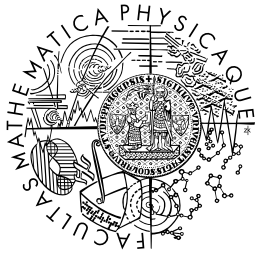

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Matějek

Framework and DSL for Ensemble-Based Access Control

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Tomáš Bureš, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor, doc. RNDr. Tomáš Bureš, Ph.D., for finding me an interesting thesis topic on a relatively short notice, providing the source code of the TCOOF-Trust prototype, and overall help with the work.

I would also like to thank my partners, Andy and Fronéma, for their support and encouragement. Without them, I would probably never even start this work.

Title: Framework and DSL for Ensemble-Based Access Control

Author: Jan Matějek

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems

Abstract: Access control policies typically take the form of a set of static rules pertaining to individual entities under control. This can be impractical in real-world scenarios: authorization invariably depends on wider situational context which often tends to be highly dynamic. This leads to increasingly complex rules, which have to change over time to reflect the evolution of the controlled system.

Ensemble-based architectures allow dynamic formation of goal-oriented groups in systems with large number of independent autonomous components. Because of the ad-hoc and situation-aware nature of group formation, ensembles offer a novel way of approaching access control.

The goal of this work is to design a Scala framework and internal DSL for describing access control related situations via ensembles. In particular, the framework will define ensemble semantics suitable for evaluating the ensembles and establishing access control at runtime.

Keywords: component ensembles; access control; domain-specific language

Contents

1. Introduction

More and more areas of human activity are becoming computerized and connected to ever-growing networks. This allows us to use these networks in novel, increasingly powerful ways, as disparate systems are now able to communicate, aggregate knowledge, and relay commands across domains. The growth and pervasiveness of Internet of Things has a great promise for building cyber-physical networks, starting at smart homes, smart buildings, and growing towards smart cities and large-scale smart grids.

At this scale, systems are necessarily heterogeneous. Devices from different manufacturers are connected via custom communication protocols on virtual subnetworks and managed by different organizations. Even if we limit the scope to a single organization wanting to computerize its physical properties, different parts of the system will usually be provided by different vendors, with separate connectivity and separate management consoles.

To further complicate things, there is a strong demand for dynamic interconnectivity. Adding a new sensor to a smart home should be seamless, and allowing house guests to, e.g., control music on the home stereo, should not be an insurmountable challenge. This is even more important in smart cities, where almost by definition users cannot be known in advance. And in enterprise environments, Bring-Your-Own-Device (BYOD) policies require that employees are able to access company systems with their own smart devices.

Taken together, we are now surrounded by interconnected systems of enormous complexity. Managing such systems is a significant challenge. Due to the nature of the requirements, systems are also more vulnerable to malicious actors. And as their importance increases, so does the value for an attacker. Therefore, security is of the utmost concern.

Our focus is on a particular aspect of system security, *access control*, both in physical and digital realms. Given a security policy, and the identities of users, it is necessary to determine whether they are authorized to perform various actions. From the physical realm, this usually means access to physical spaces, e.g., controlled by a smart lock or a card reader; but also control of building infrastructure such as heating and air conditioning, lights, etc. On the digital side, access permissions to data are common. And in a highly dynamic system, it is necessary to determine whether one party should be allowed to communicate with, and use services of, the counter-party.

Permissions are often context-dependent. In a smart city, users might only be allowed to control resources that are near them, but denied access to more distant parts of the grid. Autonomous cars coordinating on an intersection should allow communication between each other, but only those that are actually engaged in the coordination task. Emergency exits should remain locked most of the time, but allow anyone out when a fire alarm is activated — and emergency response teams should be authorized to access spaces that have strong access controls under normal circumstances.

Current state-of-the-art access control systems are not well suited for these tasks. In most approaches, a static set of rules is used for describing the security policy, basically enumerating the situations that can arise and their resolutions.

While modern approaches such as Attribute-Based Access Control can take the situation context into account, describing possibly overlapping situations causes an explosion of complexity. The resulting policy is difficult to manage and audit, resulting in possible unintended interactions between rules, or missed corner cases with undesired behavior.

This is especially true in heterogeneous, large-scale and highly dynamic systems. The sheer number of agents and their possible interactions makes it difficult to enumerate all combinations of circumstances and situations for the purpose of designing a comprehensive security policy.

This work explores a different way of specifying security policies. We build on previous research in the area of autonomic component ensembles, which provide a more natural way of describing systems consisting of dynamic relationships between many independent agents. Security situations are expressed in terms of ensembles, and the security policy attaches rules to these ensembles.

Continuing the work from [1], our aim is to present a policy specification language based on the ensemble model. We propose clear and well-defined semantics for the language, allowing its user to specify ensembles and constraints for their existence in a declarative way, with the goals of composability, readability and maintainability.

We also present an accompanying framework for identifying security situations and resolving ensembles specified in the security policy. We make sure the system is capable of observing and modeling non-computational entities and other agents that are not controlled directly, such as humans. The framework can either respond to access control queries directly, or emit a set of rules suitable for traditional systems.

1.1 Structure of this Work

We introduce a running example in chapter 2. This is a scenario which will serve as context and motivation for the rest of this work. In subsequent chapters, we will refer back to the running example for showcases of certain features, performance effects, etc.

In chapter 3 we look at problem background and related work in the fields of ensemble based systems and dynamic access control. A brief overview of many existing approaches is provided and evaluated with regard to our goals.

Chapter 4 summarizes the problem, presents a broad overview of the proposed solution, and restates our requirements in concrete terms.

Chapter 5 is intended as a user guide to the TCOOF-Trust framework and language. It contains explanations of core concepts and their semantics, provides practical examples and a full reference of the features of the DSL. It also contains a full implementation of the security policy of the running example.

Translation process from the DSL to a constraint satisfaction problem is described in chapter 6, together with implementation details for the DSL constructs.

In chapter 7 we evaluate performance of the basic operations of the framework, and then test the running example implementation in a number of synthetic and real-world-like scenarios.

Finally, chapter 8 summarizes our findings and concludes the work.

2. Running Example

For our running example, we use a modified scenario based on a real-life problem, which was first introduced in [1]. We will refer back to this scenario throughout the work, in order to showcase problems, features of the DSL, and implementation details.

2.1 Scenario

The scenario models a company that works on sensitive projects for its clients. At any given time, multiple projects can be developed in parallel. Developers working on the same project must be able to cooperate. However, to limit exposure of sensitive intellectual property, developers from different projects should not come into contact with each other; namely, developers of two different projects are not allowed to stay in the same room while in the company building. To this end, each developer has a smart device (mobile phone, smart watch, etc.) that can direct them to the appropriate rooms.

We assume that each developer is assigned to exactly one project. Only two types of rooms are considered: workrooms and lunchrooms.

Workrooms are open whenever the building is open, which is from 7:30 AM to 9 PM. Each workroom is assigned to a particular project. All developers on a project are permitted to enter all workrooms for that project, to allow for efficient collaboration. We assume that workroom assignment is fixed for the duration of our scenario and that it provides enough capacity for all developers of a given project.

Lunchrooms only open around midday, from 11:30 AM to 3 PM. Each lunchroom has a set maximum occupancy. We expect that there will not be enough lunchrooms to seat all developers, esp. with the constraint that developers from different projects must not meet in the same room. Therefore, room assignment must be dynamic. Developers will be equipped with smart devices which can send seating requests and display the current situation.

Our interest lies in access permissions. Specifically, we want to investigate which developers can enter which rooms at various times and under various conditions, and how an access control system could handle a given situation.

Figure 2.1 illustrates a possible situation. Developers on blue and red projects are moving around in the building, which has three lunchrooms and three workrooms. Two workrooms (computer symbol) are assigned to the red project and one is assigned to the blue project, as indicated by the symbol color. Lunchrooms (food symbol) are not assigned to projects. Each room has a capacity of four people.

There is a red, blue or black lock on each door. Red and blue indicates that the door is open to developers of the corresponding project. Black indicates that anyone can open the door. The lock on room D is closed, showing that nobody can enter. In this case, it is because the room is full.

Room B is also full, but the lock remains open, because our security policy does not take workroom capacity into account.

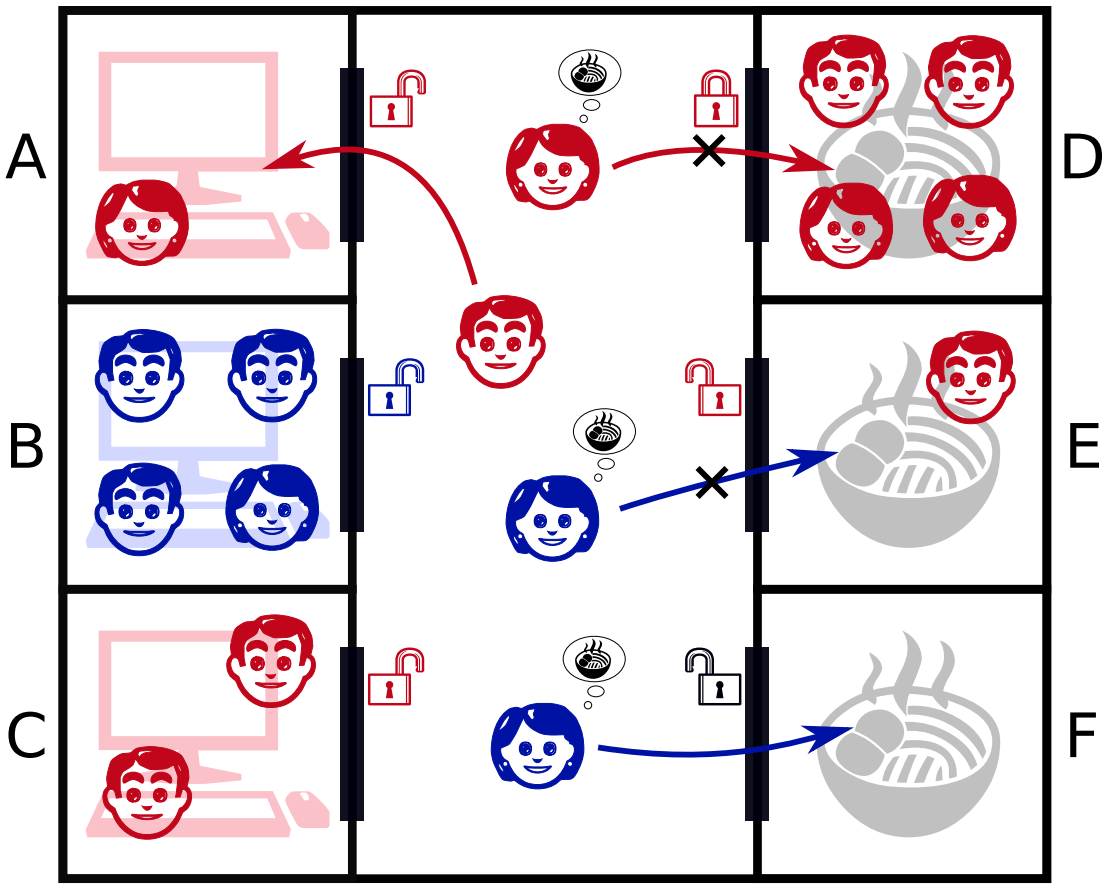

Figure 2.1: Example scenario with workrooms and lunchrooms

The red worker in the middle is allowed to enter room A, because it is a workroom assigned to a red project. The hungry red worker is not allowed to enter room D, however, because the room is full. She would be allowed to enter room E, currently open to developers of red project, but the hungry blue worker in the middle cannot do that. There is already a red worker inside, and the security policy disallows developers from different projects to meet in the same room. Hungry blue worker at the bottom can enter room F, because it is empty, so no security conflict arises.

2.2 Workroom Assignment

If we ignore lunchrooms for a moment, the problem is relatively simple. Access permissions are known in advance. At night, no rooms are open and nobody is allowed to enter. In the day, every developer assigned to a particular project is allowed to enter every workroom assigned to that same project. We can enumerate the available workrooms, and for each one, enumerate all persons allowed to enter that room. Only those people are allowed to enter, and no other access permissions exist. The rules are fixed and don't need to be updated except when scenario definition changes.

Figure 2.2 shows all possible configurations. Either it is night time, in which case nobody can enter, or it is daytime, and blue developers can enter, while red ones cannot. This scenario is easy to solve even in traditional entity-based access

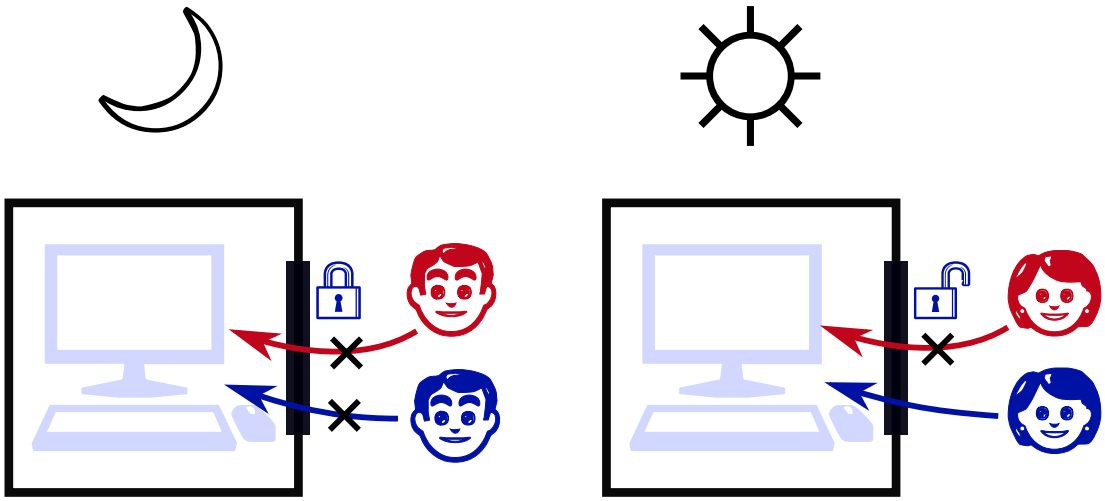

Figure 2.2: Workrooms at night and in the day

control systems. The only dynamic parameter is time of day, which is commonly supported on contemporary electronic locks.

2.3 Lunchroom Assignment

During lunch hours, a developer can request seating in a lunchroom. This introduces a dynamic and unpredictable element. It is no longer sufficient to *enforce* access control rules, we need to re-evaluate the situation and generate new access grants.

There are several possible approaches to this situation. One option is to leave the choice to the human: during lunch hours, every developer is allowed to enter every lunchroom, except when (a) the lunchroom is full, or (b) developers from a different project are already present in the lunchroom. The advantage of this approach is twofold. First of all, it gives greater freedom to the developers. The system is not trying to make decisions for them, and everyone can choose a lunchroom based on their own criteria, such as how close it is, where their friends sit, etc. Second, because the system does not need to make choices, it is computationally simpler. We need to monitor the situation and update access grants based on current seating and room capacity, but we can still describe the situation using conditional entity-based rules.

There are some drawbacks, too. When individual developers choose lunchrooms for themselves, they only take their local context into account. This can lead to a situation shown on figure 2.3. Red developers have occupied all the available lunchrooms. Even though there is more than enough total available seats, none of the hungry blue developers can use them.

There is also no way to reserve a seat in advance; a developer could head out towards their favorite lunchroom, only to find out that it is occupied and they need to go elsewhere.

On the opposite end of the spectrum, we can have the access control system make all the decisions. We are working with the assumption that developers are free to choose *when* they want lunch, so we cannot simply pre-generate fixed time

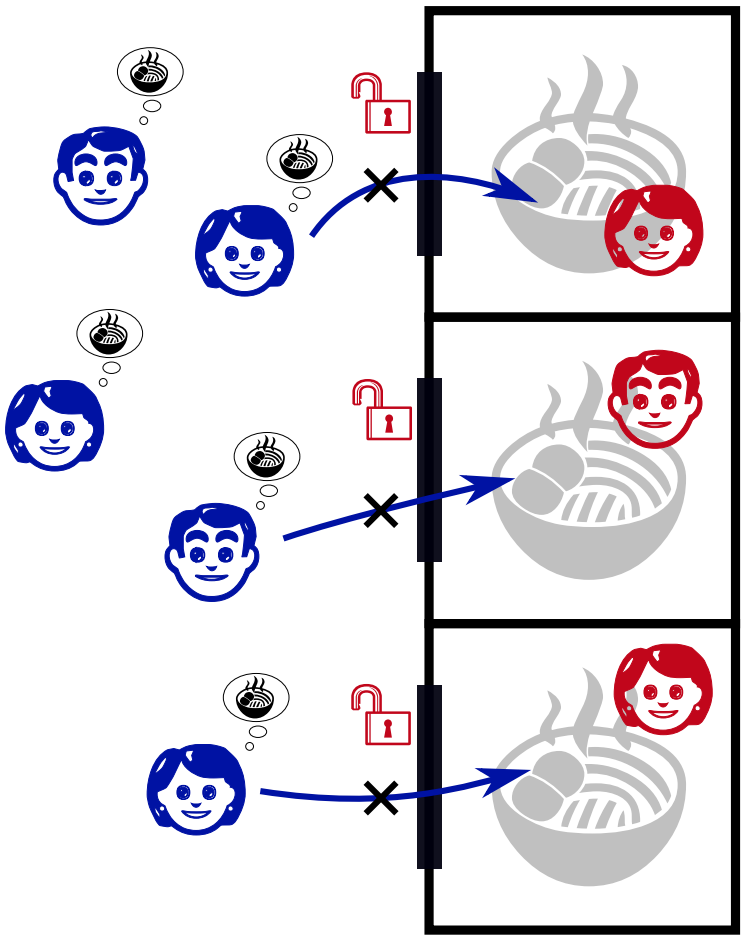

Figure 2.3: Inefficient use of available lunchrooms

slots and seats and solve a scheduling problem. However, we can still assign a particular lunchroom whenever a developer requests a seat. The rule would be as follows: during lunch hours, a developer can request a lunchroom. When a valid seat becomes available, the developer is allowed to enter the assigned lunchroom once. When they leave the lunchroom, their seat is freed for next requests.

This is the variant that we will be using in the rest of this work. It gives us the opportunity to explore behavior of an access control system with regards to solving conflicting and/or overlapping requirements.

We must note, however, that in the real world, a less strict system would be preferable. Perhaps a hybrid of the two outlined here: allow the developers to choose a lunchroom based on individual preferences, reserve a seat through a smart device, while employing heuristics to keep some free lunchrooms so that no project is starved, both in the technical and the literal sense.

2.4 Assignment Criteria

When assigning lunchrooms, we must satisfy the scenario rules: developers from different project must not be allowed to enter the same room, and we cannot assign more developers to a lunchroom than its capacity. But in addition, we might want to optimize for other criteria.

First of all, we want to achieve good utilization and availability of lunchrooms. More specifically, every lunch request should be serviced as soon as possible, and thus every developer should be able to have lunch at the time of their choosing. This means two things: (1) we need to seat as many developers as possible at the same time, and (2) at any given time, we should be able to seat a developer from any project.

While obviously limited by total capacity, there is a lot of room for different choices with regard to these criteria. Consider a situation where a group of 6 developers from project A and a group of 4 developers from project B request lunch simultaneously, and there is only one free lunchroom available. By (1), we should seat the group from project A, because it is bigger. But if all the other lunchrooms are occupied by project A, we should probably give the room to the B group to better achieve (2) — this would reserve the free space for more developers of project B, while we can expect spaces for A developers to become available in the other lunchrooms.

We could continue adding criteria, of course. Earlier requests should have priority. It might be desirable to assign lunchrooms that are physically close to the requester. We might try to keep groups together. Perhaps requests should be prioritized by employee seniority, or maybe we should even keep reserved seats for the bosses at all times. Each developer could sort the lunchrooms by preference and their selection should be taken into account.

Obviously, each additional criterion is making the problem more complex. It is not our goal to solve this complexity; the point here is to show that requirements can vary and this needs to be taken into consideration.

3. Background and Related Work

3.1 Ensemble-Based Architectures

The complexity of managing a system grows with its size. This has traditionally been handled through hierarchical decomposition, component-oriented programming, and similar techniques. With the rise of the Internet of Things, however, we are reaching the limits of these approaches. If the environment consists of many independent agents with no clear functional hierarchy, it is impractical to impose a top-down view of control. There is no single point of access to a "swarm" of sensors, smart things, and cyber-physical systems in general, especially when these are distributed over a large area with no promise of reliable connectivity and continued availability. The use-cases are new, too: clients want to access the system from different locations and perspectives, use different services in different configurations, and take advantage of the inherent dynamicity.

The problem is partially solved through *Service-Oriented Architectures* [27]. SOA revolves around modular, dynamically discovered and dynamically bound services. Unlike a typical "hard-coded" component-based system, SOA can deal with small services that are randomly appearing and disappearing from the network. There are limitations, though. SOA is still a top-down approach that assumes its parts are discrete services. This is not necessarily true with IoT. Consider a home sensor network. SOA can expose every sensor as its own service, but then it falls to the client to locate, e.g., all sensors in a single room. Alternately, the whole network could be a single service, but then it needs to have "zoom in on a single room" as a feature. This gets complicated when the query changes, e.g., when locating all temperature sensors. The issue is more pronounced when considering a heterogeneous city-wide network of smart things. Also, service discovery starts being a difficult problem when the number of services grows to tens of thousands.

A whole other issue is agent collaboration. Certain problems lend themselves to distributed approaches — e.g., self-driving cars collaborating to find parking spaces along their driving routes. While it is possible to create a centralized service for this task, it would be more practical to have the agents collaborate directly and locally. This is not a problem that SOA can solve, and indeed it is unsuitable for any kind of top-down approach. Instead, we need to look towards multi-agent systems to manage the collaboration, and possibly some other approaches that will allow the agents to locate each other, establish communication links, organize and reorganize.

To restate: we are experiencing an explosion of the size of cyber-physical systems and the number of independent computing devices, and traditional approaches fail to use these systems efficiently. There is a growing need to organize highly dynamic systems that consist of many independent agents. Not only is a different approach required, we actually need different abstractions.

One such abstraction is the *ensemble*. An ensemble is a loose coalition of *components* formed around a shared goal. A component can be any kind of moving part: in a software system, a component would be a module, object, service, any piece of code performing a particular functionality. In the world of

cyber-physical systems, a component can be a computer, an IoT device, a drone, etc. As we show in this work, we can even model humans, rooms and other non-computational entities as components.

Members of an ensemble inhabit specific *roles*, which are usually described in terms of capabilities or services a component can provide. This decouples concrete components from the ensemble. For example, an ensemble providing environmental data for a HVAC[1] system could have a "thermometer" role, a "CO_2 detector" role, and an "uplink" role which handles Internet connectivity. These would be inhabited by appropriate devices, without regard to which specific device is providing which service.

A single component can perform multiple roles in the same ensemble (e.g., an Internet-capable thermometer can function both as a sensor and as an uplink), or be a member of several different ensembles (the same uplink can serve multiple sensor suites at the same time). It is also conceptually simple to replace a component with a different one in the same role, in case the former member becomes unavailable or no longer fits the ensemble's parameters.

Ensembles are composable; an ensemble can contain any number of sub-ensembles, which in turn can be composed of more sub-ensembles. Sometimes it is useful to set up a role which can be inhabited by a sub-ensemble.

One of the key features of an ensemble-based architecture is its high level of dynamicity. Ensembles are formed and dissolved when needed. Similarly, membership in an ensemble is driven by current context and conditions, not by any sort of explicit registration. Therefore, the natural way to specify ensembles is declarative. We can use predicates over component properties and the local conditions. E.g., a smart parking lot can define an ensemble as "parking meter, plus all vehicles seeking parking within 10 minutes from it."

This meshes well with ideas of autonomic computing [12]: individual components could be able to determine ensemble membership based on their own knowledge, without a central coordinator. Alternately, when a global perspective is available, a coordinator can use all available information to optimize ensemble membership and role assignment, and can be flexible in responding to change.

Due to its ephemeral nature, it is difficult to work with an ensemble "from the outside". In an autonomic system, ensembles might not even be visible from the outside, their existence known only to the individual members. And even with a central coordinator, there are some challenges: for example, an ensemble can dissolve at any time and its constituent components can be reassigned. There is no way to maintain a persistent reference to a particular ensemble or a role within it. Instead, the ensemble system should be considered self-organizing and self-governing. A user of the ensemble system can post goals, and the system will ensure that ensembles are formed to resolve that goal.

3.1.1 Related Work

Ensembles present a perspective which differs significantly from traditional views of component systems. For this reason, new methodologies and paradigms are being designed for efficient programming of ensemble systems.

[1] Heating, Ventilation, Air Conditioning

Software Component Ensemble Language was introduced in [4] in 2013 and refined a year later in [18]. SCEL is designed as a "kernel" language, a mostly abstract grammar that is supposed to be a building block for more concrete and full-featured languages. Its paradigm is built on four concepts: *knowledge* of individual components; *behaviors* operating over knowledge bases; *aggregations* as collections of components; and finally *policies* that can control and adapt execution of behaviors. Each component is represented as an interface exposing its knowledge base, available behaviors, and governing policies.

Despite having "ensemble" in the name, there is no explicit notion of an ensemble nor a role. They exist implicitly, via the ability to control targets of actions with logical predicates.

Interestingly, the SCEL paper [18] shows an access control sub-language as an example of the policy concept. However, the language is used to protect processes on the level of one component, and its capability to consider broader situation context is limited, so it is of little interest to the topic of this work.

HELENA, from *Handling massively distributed systems with ELaborate ENsemble Architectures* [9] [13], is a rigorous formal approach to ensemble-based systems, focused primarily on the role concept. The goal of the design is to enable formal verification of ensemble behavior. Role operations are specified by a labeled transition system, and ensembles are modeled as automata. Components are considered resources for the roles they can fulfill, but are themselves passive; any active behavior is initiated through the role abstraction.

As the HELENA approach examines the problem from the role level, the process of forming an ensemble is left unexplored. Finding components to inhabit roles is assumed to be implicit.

The *Distributed Emergent Ensembles of Components* (DEECo) model [2] is a more practice-oriented approach, tailored for use in IoT applications. It defines a paradigm of *ensemble-based component systems*, where components, roles, and ensembles are all first-class objects. The DEECo paper also discusses the question of reliability of communication and presumes use of mesh networking.

Membership in an ensemble is determined by a *membership condition* over components' knowledge attributes. Component roles are modeled as interfaces, prescribing the knowledge attributes that must be available to the ensemble. Ensemble definition further specifies what knowledge is exchanged between members, when, and how often.

Ensemble formation is initiated by a *coordinator*. Each ensemble specifies a single coordinator role. The component which inhabits that role will look for other members via local or routed broadcast on the network, and facilitate the knowledge exchange processes. While the paper doesn't elaborate on this, the concept is nevertheless important, as it basically imposes locality on the search for members.

A follow-up paper [14] expands on the idea of search locality, adds discussion of performance issues, and introduces the concept of filtering as a way to limit the number of components in consideration. It also adds a concept of *fitness*, a quality score of the formed ensemble. For example, it is possible to specify that an ensemble consisting of nearby components is a better candidate than an ensemble whose components are far from each other.

In [3] an architecture definition language called *Trait-based Coalition Formation ADL* (TCOF-ADL) is presented. Its express goal is to facilitate selection of components and formation of ensembles, while ensuring that formation responsibilities are properly distributed between agents in the ensemble system. It provides some basic predicates and allows components to be extended with ensemble formation related traits such as location awareness, data prediction, and a statistical evaluator. The language is implemented as a Scala-internal DSL and employs a constraint solver to help with ensemble membership resolution.

A variant of TCOF-ADL, named TCOOF-Trust[2], is presented in [1]. The paper specifically deals with application of ensemble concepts to access control, and this work is a direct follow-up to the same research. The TCOOF-Trust language reuses some of the original constructs, as well as the ensemble formation engine, and adds commands related to access control decisions.

3.2 Access Control in Dynamic Systems

Historically, access control tended to be rule-based and local to specific resources, such as files or concrete objects. In UNIX, for example, each file has an owner and a group, and a fixed bitmap of permissions that can be granted or denied to the owner, members of the group, or everyone else. An improvement to this system is an *Access Control List* (ACL) attached to an object, which can list an arbitrary number of users, groups, and their permissions.

As system complexity grows, this is no longer a sufficient solution. Many contemporary systems use some variant of *Role-Based Access Control* [8] (RBAC). Access control rules are expressed as tuples: (*role*, *operation*, *subject*), authorizing members of *role* to perform *operation* on a *subject* resource. In addition, roles can be arranged in a hierarchy, so that a role higher in the hierarchy inherits all permissions of lower roles. This naturally fits human organizational hierarchies: e.g., a supervisor role will usually inherit all permissions of the worker role, without need to specify them again.

A single user can be assigned to any number of roles. This provides an important abstraction and simplifies user management, as it is now possible to grant or revoke user privileges simply by modifying their role assignment. A more recent *Organization-based Access Control* [11] (OrBAC) model adds another layer of abstraction: each of *role*, *operation* and *subject*s in the master policy can now be "implemented" by a corresponding object in a local environment. This enables sharing policies between organizations at design-time and collaboration at run-time, as two organizations can cross-authorize their users in compatible roles.

As described, RBAC-like policies are not aware of wider context. Role R has permission to perform operation O on subject S at all times, regardless of situation or current conditions. That is not good enough for many real-world scenarios, where context plays an important role. For instance, a "worker" should have permission to "enter" the "workspace" — but only during work hours.

Several approaches arose to close this gap. *Context-Sensitive RBAC* [16] adds context information to roles, operations, and subjects, and attaches predicates

[2]The additional "O" in the name is not explained. Presumably it comes from changing "coalition" to "coordination" in the acronym.

over the contexts to individual access control rules. Similarly, *Context-Aware RBAC* [15] introduces preconditions that can query context information before allowing the rule to apply. In addition, a "context guard" predicate can be specified, which must hold true in order for active sessions to remain open. In [26], context constraints are separated into those that can be evaluated at design-time and those that must be evaluated at run-time, and a development methodology for constrained rules is presented.

In recent years, *Attribute-based Access Control* [10] (ABAC) is gaining popularity because of its even greater flexibility. In ABAC, permissions are not attached to roles, but to arbitrary expressions over attributes of users and subjects. As an example, RBAC could be considered a special case of ABAC, with the only attribute under consideration being the list of user's roles. But every object can have any number of attributes and these can dynamically reflect current situation. Attributes of environment, such as current date and time, can also be accessed.

A popular realization of the ABAC model is OASIS XACML [19] standard, which is an XML-based policy language, and a more developer-friendly language ALFA [20] that translates directly to XACML. Developers can describe rules using a large number of built-in predicates, or they can define custom functions for more complex situations.

The main limitation of these approaches, which we will collectively call "rule-based", is the actor-action-subject perspective. ABAC is capable of querying the situation, but the query still originates from a static description of applicable actors, actions, and subjects. In other words, rules can query context, but context cannot influence rules.

This makes rule-based approaches highly impractical in scenarios where the context does in fact cause the rules to change. The implementation of such context-dependent security policy is scattered across many different rules, which makes it hard to audit, maintain and debug. In fact, it is difficult to enumerate the rules that should be affected in the first place.

One example is dynamic global state which cross-cuts ordinary rules. For instance, a "night policy" might be different from a "day policy". Affected rules would need to have two variants, and updating one of the policies couldn't be done in one place. In addition, if global states can overlap (consider a "lock-down state" when every door only opens for members of a designated responder group and nobody else), complexity can grow exponentially. This growth manifests in every affected rule, possibly to the point where it is more practical to go outside the ABAC system and simply maintain separate manually deployed policy configurations.

In highly dynamic systems, such as IoT networks, the same limitation can be seen from a different angle: it is burdensome to express every possible combination of actors, actions, and subjects beforehand. Dynamic relationships arise, form, and dissolve throughout the life of the system. In our running example, a person is not allowed to enter a room if another person working on a different project is already inside that room. Such rule can be expressed as a context query with some creative use of attributes, but it is more natural to query the context first and dynamically form the rule afterwards.

A number of so-called "adaptive security" solutions and frameworks have been

proposed to get around this limitation, each with their own pros and cons.

dynSMAUG [17] paper works with the concept of a *situation* distinct from *context*. While context is a collection of information on the actors and the environment at a specific point in time, a situation is a time frame — an interval in which some context predicates hold true, possibly including particular sequences of events. Situations are defined separately from the security policy in a SQL-like language Esper [7], and currently applicable situations are exposed as attributes queryable by XACML/ALFA languages.

While the model is sound, the need to describe the policy in two phases and in two different languages is impractical in terms of maintainability. Furthermore, the Esper language is powerful but also very complex even for simple cases.

A self-learning model was presented in [25]. Protected assets, operational goals and possible threats are modeled and then compiled to a fuzzy decision network. This network can evaluate the state of the system at run-time and configure or reconfigure applicable security policies. This solution, however, is limited to reasoning about a closed system, with no consideration for independent agents.

The solution presented in [24] uses inference rules based on logic programming to build higher-level context information from lower-level knowledge (such as sensor readings). Resulting high-level knowledge is then used as a basis for security actions. This enables the policy designer to make security decisions at the appropriate level of abstraction.

4. Solution Overview

Contemporary access control models are overwhelmingly of the rule-based variety — that is, the policy is a set of rules in the format "*actor* is permitted to *action* on *subject*". They do not cope well with highly dynamic systems, where strong situational awareness and ability to handle ad-hoc formed relationships between independent agents is required. Adaptive models have been proposed, which can handle some parts of the problem, but there is currently no consensus on the appropriate way to control systems like smart spaces, IoT sensor swarms, and other large-scale dynamic environments.

We are interested in a model that would have all of the following properties:

- *Situational awareness.* The system should be context-aware by default, able to detect situations based on low-level information, and make situation-appropriate decisions.
- *Dynamicity.* The system should cope well with environments consisting of a large number of agents that can appear, disappear, and form ad-hoc relationships at any time. It should enable decomposition of the environment into logical groups, and update immediately when availability or properties of the group members change.
- *Composability.* It should be possible to break the policy down to functional parts and build higher levels of policy from the lower-level components.

  It should also be possible to develop parts of the security policy separately, even if they relate to the same actors or subjects. The resulting policies should be applicable at the same time if their end results do not conflict, and there should be a way to resolve conflicts that arise.
- *Maintainability.* The policy specification should enable good engineering practices. Related concepts should be specified close together. Security decisions should be made at the appropriate abstraction level. It should be easy to see which parts of the policy are affected by a change, or which parts must be changed to achieve the desired effect.

The idea explored in this work is to describe security situations in terms of ensembles with attached access control decisions, have a solver element identify which components are members of which ensembles, and apply the rules on them.

Ensemble-based architectures naturally possess the first three desired properties. They are designed for large-scale systems composed of many independent agents, and are suitable for highly dynamic environments. Ensemble formation depends on current context. An ensemble is a logical group of components, so it is a decomposition of the larger system. It can also itself be constructed from sub-ensembles, allowing composability. Components are not limited to a single ensemble, so ensembles can overlap.

As for the fourth property, maintainability, that depends for the most part on the design of a specification language and choice of the right abstractions. As explained in section 3.1, declarative specification is a natural approach for ensembles. Another notable feature we are interested in is composability on the

language level, not just on the conceptual level. Work on SCEL and DEECo is a good inspiration here.

To make ensemble-based architectures applicable to the problem of dynamic access control, we will need to tweak some of their properties.

First of all, ensemble-based systems are usually distributed and components self-organize based on their knowledge. That approach is obviously not applicable to access control. We will require a trusted supervisor with complete information to form the ensembles and enforce the rules.

Another issue is beyond-control entities. Access control systems routinely deal with independent agents beyond their control, such as humans; after all, the whole point of an access control system is to make decisions that limit actions of beyond-control agents. Ensemble systems, on the other hand, tend to require agent cooperation in order to achieve their goals.

The ensembles that we use are not goal-oriented, however. There is no reason to control actions of individual components or direct them imperatively. From the access control point of view, component knowledge — and indeed the ensemble structure as such — is purely descriptive. We can represent beyond-control entities as virtual components and gather knowledge about them indirectly via sensors. E.g., the system can track a human's position with an indoor positioning system and their smartphone, and can monitor room capacity by counting incoming and outgoing door openings.

Using a central supervisor happens to help us here, too. It is a good place to store the collected knowledge, and to perform computations that would be done on the components in a more traditional ensemble system.

4.1 Practical Example

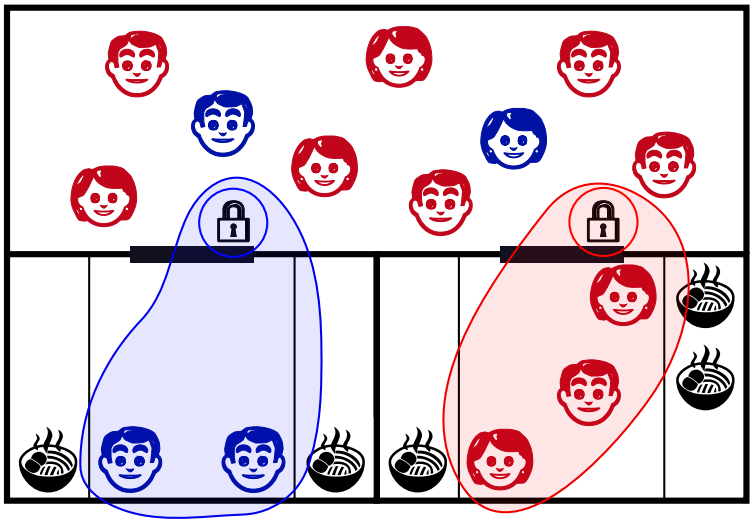

Figure 4.1: Example situation: no outstanding lunch requests

Figure 4.1 illustrates a part of our running example. There are two lunchrooms, each with six available spaces. Two people from a blue project are already eating in the left one, and three people from a red project are eating in the right one. The translucent shapes indicate two existing ensembles, each of which is

formed around a door lock. Membership in an ensemble grants the person a permission to open the corresponding lock.

In the first picture, nobody is requesting access to a lunchroom. The only people allowed to unlock the doors are those already inside, who should obviously be allowed to leave.

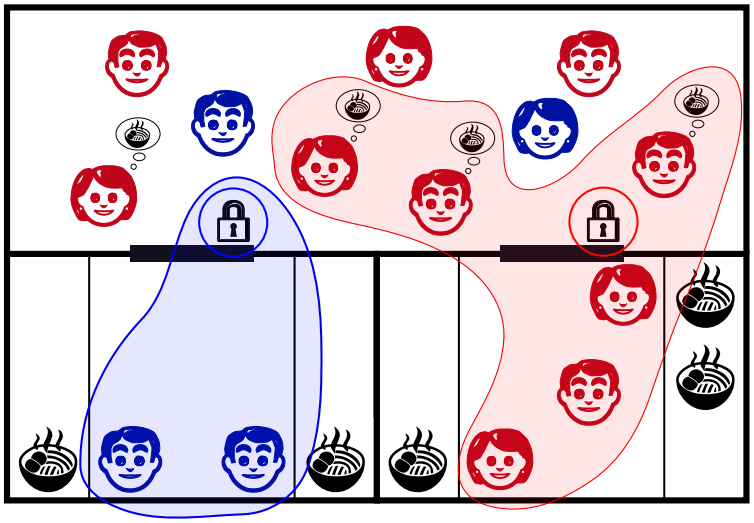

Figure 4.2: Example situation: new lunch requests

In picture 4.2, four more people from the red project are hungry, as indicated by thought bubbles. When the system is informed of this, it should allocate lunchroom seats to the hungry people. The red ensemble extends to include three of the hungry people — but not the fourth, because there is no more space for them in the lunchroom. The blue ensemble doesn't include them, because they are working on a red project, and this would violate the constraint that people from different projects cannot eat together.

The example in figure 4.3 uses a pseudo-language to concisely express the ensemble configuration, which roles exist, and what constraints limit the available solutions. The `allow` verbs also specify access control rules to apply on the ensemble.

```
ensemble Lunchroom {
    role Room = select 1 of all rooms
    role Eaters = select all of Room inhabitants
    role Requesters = select some of all hungry

    constraint {(Eaters + Requesters) must not exceed Room capacity}
    constraint {all Eaters and Requesters must have the same project}

    allow Requesters to enter Room
    allow Eaters to leave Room
}

constraint {Room from every Lunchroom must be distinct}
constraint {Requesters from every Lunchroom must be disjoint}
```

Figure 4.3: Pseudo-code for specifying a small ensemble configuration

The code defines an ensemble type for a single lunchroom. Three roles are defined: `Room` is the selected lunchroom, `Eaters` are people already inside, and `Requesters` are those who are waiting for a seat. All eaters are selected, but only some requesters. This prevents a requester from displacing someone who is already in the room. The ensemble-local constraints ensure that we do not pick more people than we can (which effectively only limits the number of new picks) and that all selected people are from the same project.

Global constraints, applied over all instances of `Lunchroom`, ensure that no more than one ensemble is formed per room and that a single requester is not picked for more than one lunchroom. It is important to note that ensembles are generally allowed to overlap; without this constraint, situation on picture 4.4 would be perfectly valid. The same requester would take up space in both lunchrooms, pushing out the hungry person in the top left.

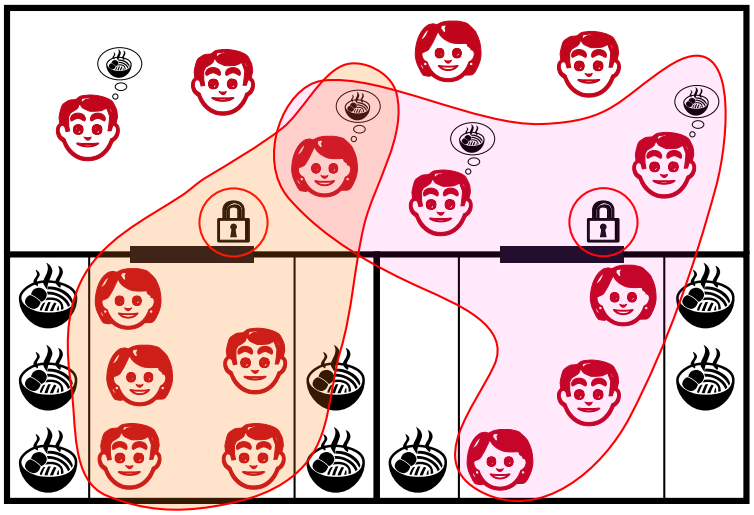

Figure 4.4: Example situation: same person is selected for both rooms

Given a definition similar to the example above, the supervisor should analyze the current situation, choose an appropriate assignment of components to roles, and grant the specified permissions.

4.2 Goals Revisited

There are now several open problems that must be solved in order to make the above example actually work. There are also several more necessary features that are not shown in the example. We can divide the goals into two distinct areas: the specification language, and an implementation of the supervisor element. In the rest of the text, we will call this supervisor element the *solver*.

4.2.1 Specification Language

Our example uses pseudo-code to outline the desired semantics. We need a language that is parsable by the solver and expressive enough to support all the desired features. Our goal is to design such language and specify its semantics in detail.

Language design poses a problem on its own. It would be ideal to reuse an existing language and only extend it to support our use-case. For this reason, we decided to implement an internal domain-specific language (DSL). Scala [28] was chosen as the host language for its flexible syntax and strong typing system. Scala gives us a repertoire of powerful language features, operators, and primitives readily available for the DSL. We are thus free to concentrate on the specification semantics.

At its most basic, our language must allow definition of ensembles, roles within them, and membership predicates for role inhabitants. As shown in the example, it must also be possible to specify arbitrary constraints, applicable both locally within the ensemble and globally over multiple ensembles.

Next, the language must provide a way to specify when an ensemble is applicable. In our running example, lunchrooms should only open during lunch hours. This is partially possible with arbitrary constraints on the ensemble formation: if one of the constraints is "it must be lunch time", the ensemble will simply not form at other times. However, an ability to *enforce* that a particular ensemble is formed might be useful in certain use cases.

Many ensemble configurations can have multiple solutions. Looking back at picture 4.2, there is nothing in the pseudo-code specification telling the red ensemble to find as many hungry people as it can. If the ensemble remained the same as on picture 4.1, it would still fit all the constraints. The specification language should thus have a way of indicating "solution quality", or scoring a given solution.

Finally, as we are interested in access control, it must be possible to attach security decisions to the ensembles and specify when and how they apply. A reasonable format is RBAC-like *actor-action-subject* — it is familiar, and at the point when the rule is applied, we have already expressed all contextual awareness by forming the ensemble.

4.2.2 Solver Implementation

The second goal is to have a working implementation of the supervisor element. It should process a policy specification and determine which ensembles are applicable and which components will inhabit which roles. When the specification is ambiguous, the solver should choose a solution that satisfies all constraints.

Given a set of components and arbitrary constraints, finding a valid role assignment can be transformed into a constraint satisfaction problem. This would allow us to reuse an existing constraint solver library instead of creating our own. Therefore, the solver should be capable of converting the relevant parts of policy description into a representation appropriate for a constraint solver.

As previously stated, some scenarios can have a method of scoring the solution quality. The solver must be able to configure itself to either find any viable solution, or to find a solution optimal in some variable. Furthermore, certain problems can be described in multiple different ways, and different descriptions can have different performance characteristics. The solver must be amenable to such tweaks, possibly even configurable in how it explores the solution space.

And of course, after a solution is found, the solver must allow users to inspect the solution and execute access control queries.

4.2.3 Solution Novelty

This work is a direct follow-up to the paper [1], which introduced both the idea of using ensembles for access control, and the Scala DSL implementation which is the basis for our framework. That implementation was presented as a proof of concept, mainly intended for rapid prototyping. Our aim is to improve upon it in several notable areas.

On the DSL side, our main goal is to extend, refine, and stabilize the set of available language constructs. Each function, method, or construct should have well-defined semantics and a clearly stated purpose.

Another goal is to ensure that using the DSL is user-friendly and free of pitfalls. The original code already makes good use of Scala's type system to catch many problems at design time. We review and extend this system and clearly document the limitations of the DSL embedding.

We ensure that the framework codebase is using good engineering practices and that it is readable, reusable, and extendable. We provide a straightforward and efficient API for embedding in other projects, and add a comprehensive test suite with unit tests for each feature of the framework.

Finally, we perform detailed performance evaluation of various features of the solver and the framework as a whole, and provide infrastructure for further measurements.

5. Ensemble Framework

We have designed and developed a framework for managing access control with ensembles. It consists of a *domain-specific language* (DSL) for describing the ensembles, and a *runtime environment* which can analyze the ensemble description and generate the specified ensembles from available components.

The DSL is internal within the Scala language. That means that it uses the basic syntax of Scala, is compiled with the Scala compiler and runs on the Java virtual machine. We have implemented a number of functions and language constructs that enable the user to express ensemble-related concepts in a succinct and readable way. We also leverage Scala's strong type system to enforce typing checks and catch problems at compile-time.

This chapter serves as a user guide to the framework. Section 5.3 shows a very simple example of usage. Section 5.4 introduces each of the main concepts in the framework, their function and semantics. Section 5.5 translates our running example from chapter 2 to the DSL. Finally, section 5.6 lists all features available in the DSL and their usage.

5.1 Typographical Conventions

In the following text, *italic type* is used to emphasize newly introduced terms. After the term is explained, its further occurrences are set in normal type.

To highlight source code elements, such as functions, types, variable names, and code snippets, we use `monospace font`.

5.2 Overview

The purpose of the framework is to analyze the DSL-specified ensemble structure and assign components to appropriate roles in a way that fits all the constraints, and/or maximizes values of some variables. This can be understood as a constraint satisfaction problem (CSP). Therefore, the main component of the framework is a CSP solver. We use the word *solving* for the process of determining ensemble membership, and a *solution* is a particular assignment of components to roles in the ensembles. Only when the scenario is solved, we can generate access control rules based on the solution.

Input of the framework consists of two parts. First is a description of the ensemble structure, expressed with the DSL. Second is a collection of components, which are supposed to be assigned to roles.

When the computation is done, the framework outputs a reference to the solution. It is possible to examine which ensembles were activated and which components were selected for which roles. The framework also emits access control rules and notifications.

5.3 Hello, World!

The following code snippet describes a simple ensemble with one member — the `greeter`, who is granted the `"greet"` permission for each of `people`.

```
1  class Person(name: String) extends Component {
2    name(name)
3  }
4
5  class SimpleScenario(val people: Seq[Person]) {
6
7    class HelloWorld extends Ensemble {
8      val greeter = oneOf(people)
9
10     allow(greeter, "greet", people)
11   }
12
13   val policy = Policy.root(new HelloWorld)
14 }
```

Line 1 defines a very simple component — a person with a name.

Line 5 defines a scenario class. We use scenario classes to enclose the ensemble definitions and related data. In this case, it's the list of `people` on which our ensemble will be operating.

Line 7 defines a root ensemble, and line 8 specifies a `greeter` role, which is `oneOf` the `people` in this scenario.

With line 10 we grant the selected `greeter` the permission to perform action `"greet"` on any of the `people`. Actions are specified as strings.

Finally, line 13 sets up a new instance of the `HelloWorld` ensemble as a root. This tells the solver where to start.

We have specified our scenario, but the code above doesn't actually *do* anything. We need a way to execute it and provide the list of people. Let's create a companion object:

```
15 object SimpleScenario {
16   val Names = Seq("Roland", "Lilith", "Mordecai", "Brick")
17
18   def main(args: Array[String]): Unit = {
19     val people = for (name <- Names) yield new Person(name)
20     val scenario = new SimpleScenario(people)
21
22     scenario.policy.resolve()
23     for (action <- scenario.policy.actions) println(action)
24   }
25 }
```

The `main` function generates a list of `Person` instances, which is then used to instantiate the scenario on line 20. The `resolve` call on line 22 instructs the framework to find and apply the first solution. Line 23 prints out all permission grants.

When the program is executed, its output will look like this:

```
AllowAction(<Component:Roland>,greet,<Component:Roland>)
AllowAction(<Component:Roland>,greet,<Component:Lilith>)
AllowAction(<Component:Roland>,greet,<Component:Mordecai>)
AllowAction(<Component:Roland>,greet,<Component:Brick>)
```

We can see that the solver selected the first `Person` to be a greeter, and granted them permission to perform the `"greet"` action on all the other `Person`s.

5.4 Core Concepts

5.4.1 Components

Every entity that the runtime knows about needs to be represented as an object derived from `Component`. That means connected devices, locks, and even people, are treated as components. They can be members of ensembles and inhabit roles.

A component represents the system's knowledge of an entity, but the framework has no control over it. From its point of view, a component and its data are purely inputs. This is also how we can represent people as components: as far as the runtime is concerned, they are view-only.

Component instances don't need any methods, and in fact they should not have any. The possible exception to this recommendation is "computed knowledge". Sometimes it is useful to have a helper method that returns the result of some computation over the component's attributes. For example, a worker component can have knowledge of its location, and a method `isInLunchRoom` that returns true if the location is a lunchroom. The result of this method is still component knowledge, i.e., something we know about the component, but does not need to be stored as a value.

Components should come from outside the ensemble definitions, preferably from outside the scenario class. Component instances should never be created within an ensemble. All knowledge about a component should be represented in its attributes.

5.4.2 Ensembles

Every ensemble is represented as an instance of a subclass of `Ensemble`. The instance holds references to roles and their assigned components, sub-ensembles, situation predicates and constraints.

If there is just one ensemble or sub-ensemble for a particular purpose, it can be specified as a singleton with Scala's `object` keyword. If more ensembles of the same type are needed, a class is more appropriate. All instances of the class must be explicitly created in the parent ensemble body. Every instance must also be registered, using either `rules` or `ensembles` call:

```
1  class Root extends Ensemble {
2    object SubEnsembleObject extends Ensemble {
3      // ...
4    }
5    class SubEnsemble(id: Int) extends Ensemble {
6      // ...
7    }
8
9    val subEnsembles = for (i <- 1 to 5) yield new SubEnsemble(i)
10   rules(subEnsembles)
11   ensembles(SubEnsembleObject)
12 }
```

Section 5.4.3 explains the difference between `rules` and `ensembles`.

The class body can specify role membership, sub-ensembles, constraints, and situation predicates. Each of these features is described in its own subsection.

5.4.3 Ensemble Activation and Situations

An ensemble can be active or inactive. We also use the term *selected* — as in, the ensemble is selected as a member of the parent ensemble. Ensembles registered with the `rules` function are active by default. When registered with `ensembles`, the framework dynamically determines whether the ensemble should be active or inactive, based on constraints in its parents.

An active ensemble takes part in the computation and all constraints specified in the ensemble must be satisfied. Inactive ensembles are not considered, their roles have no members, and all their sub-ensembles are also inactive — i.e., an ensemble can only be active if all its parents are active.

It is possible to specify a boolean *situation predicate*. If the predicate evaluates to false, the ensemble and all of its children are deactivated. Note that an ensemble registered with `ensembles` can still be inactive if its situation predicate is true; it is a necessary condition, not a sufficient one.

The situation predicate is specified via the `situation` construct. At most one situation predicate can be used per ensemble; if more than one `situation` is defined, the last one is used and all others are ignored.

5.4.4 Roles

Fundamentally, an ensemble is a collection of components assigned to distinct roles.

Some role assignments can be known in advance. Such roles don't need to be explicitly created. It is a good practice to assign the component to a member variable of the ensemble, but it is perfectly legal to use a variable from an outer scope directly.

Other roles are assigned dynamically by the framework. For these, it is necessary to describe parameters of the assignment and a collection of candidates.

The function `subsetOf` specifies that a particular role is a subset of a given collection of components, or a subset of inhabitants of a different role. An optional argument allows in-line specification of a constraint on the subset's cardinality, e.g., that no more than N components must be selected.

As a shortcut, the `oneOf` function defines that exactly one of the candidates will be selected for the role. The `allOf` function defines that all of the candidates will be selected, effectively converting components directly to roles. This can be useful in some cases, but it is usually not necessary to do it explicitly.

The function `unionOf` can link several roles into a single group. This is useful in situations where a constraint applies over many roles together and it would be difficult to express it in terms of the individual roles.

Dynamically assigned roles are sets of components. A component can be a member of multiple roles, but cannot be a member of the same role in the same ensemble more than once.

The result of each of these functions is a *role object*. After a solution is found, this object can be used to access a list of the selected elements. At solving time, however, the results are not yet available, and instead specialized methods of the role object must be used. See section 5.6.5 for a complete list of available operations.

5.4.5 Constraints

A *constraint* places arbitrary limits on the solution. Unlike situation predicates, which are evaluated beforehand, constraints are applied during the search for a solution. It is thus a good place to specify requirements that the solution must fulfill. Constraints can be used to specify that membership of some role must be disjoint with another role, mark one role's cardinality as strictly smaller than that of another role, etc.

Constraint predicates are specified with a `constraint` call. There can be multiple `constraint` calls in an ensemble, and all specified constraints must be fulfilled in the solution. If a constraint in an active ensemble cannot be fulfilled, no solution will be found.

Constraints in inactive ensembles are ignored.

Care must be taken when creating constraints over the results of `subsetOf` and the other role functions. Role objects define several methods, such as `all`, `sum`, and others, whose results are valid constraint objects. However, if the list of selected members is accessed directly while specifying a constraint, an exception will be thrown. See section 5.6.5 for details.

5.4.6 Solution Utility

It is possible to attach a utility expression to an ensemble. If present, the framework will by default try to maximize the value of this expression when looking for solutions. In other words, solutions with higher utility are preferred.

Utility expression can be specified with the `utility` construct. Only one utility expression can be specified per ensemble. If more than one `utility` is used, only the last one takes effect.

If multiple ensembles specify an utility expression, the total utility of the solution is a simple sum of all the individual utilities. This is usually the desired behavior. E.g., when calculating utilities for individual lunchrooms from the running example, each room has its own utility, and we are interested in the sum of all rooms.

Sometimes this solution is insufficient, however. Maybe the total utility of an ensemble is the average utility of each of its sub-ensembles. If that's the case, `utility` should not be used in the sub-ensembles, only in the parent. The sub-ensembles can instead specify a method calculating the partial utility, which will then be used in parent's utility expression:

```
1  class Root extends Ensemble {
2    class SubEnsemble extends Ensemble {
3      val someRole = subsetOf(members, _ < 4)
4      def subUtility = someRole.cardinality * someRole.sum(_.weight)
5    }
6
7    val subEnsembles = for (_ <- 1 to 5) yield new SubEnsemble
8    utility {
9      subEnsembles.map(_.subUtility).reduce(_ + _) / subEnsembles.size
10   }
11 }
```

Only active ensembles are counted towards the utility total.

Similar to role objects, utilities are not actual numbers. Methods like `sum` are not available and more elaborate calculations must be built from basic operators.

5.4.7 Root Ensemble

One ensemble must be designated as top-level, or root, in the scenario. This tells the framework where to start with solving. The root ensemble is always active, and `situation` specified in the root has no effect.

It would be natural to specify the root ensemble as an `object`, but due to limitations of the DSL embedding, that is not possible. The root ensemble **must** be a class, instantiated in a call to `Policy.root`:

```
1 class Example {
2   class Root extends Ensemble {
3     // ...
4   }
5   val policy = Policy.root(new Root)
6 }
```

See section 6.2.7 for explanation of this requirement.

Result of the `Policy.root` call is a `Policy` object. By convention, we always store it in a member variable named `policy` in a scenario class.

5.4.8 Scenarios and Solving

Once a policy is specified, it is necessary to supply the actual `Component` instances and other contextual data. These are usually enclosed in a so-called *scenario class*.

In terms of code, the scenario class is a plain class and does not need to implement any particular traits. Its purpose is code organization: it is a good practice to keep the policy definition (i.e., the hierarchy of `Ensemble`s) in the same scope with the required context information. Having the context encapsulated within a class enables use-cases such as multiple instances of the same policy for different buildings.

By convention, a *policy instance* is stored in an attribute named `policy`.

The policy instance, created with `Policy.root` call, provides access to the framework's solving functionality. The most basic feature is the `resolve()` method, which performs the computation, assigns members to roles, and generates access control rules. The solution can be examined through the `instance` member, which is a reference to the instance of the root ensemble. The rules can be accessed through the `actions` member or queried via the `allows()` method.

The framework does not monitor the situation continuously. The solution generated with `resolve()` is valid for the situation at the time it was called. If the situation changes, component knowledge updates, etc., the user of the framework is responsible for calling `resolve()` again.

Every call to `resolve()` runs from scratch. Previously established ensembles are dissolved and members are assigned to roles without consideration for previous assignments. This matches the operation of ensemble-based systems: ensembles are loose coalitions that form and re-form based on current conditions. For handling of persistence, refer to subsection 5.4.11.

If no utility functions are specified, `resolve()` will find and apply the first solution that satisfies all the constraints. When utility functions are present, it will find and apply a solution with the highest utility.

It is also possible to iterate over possible solutions manually. The `init()` method will reset solver state. After that, every call to `solve()` will find a new solution, viewable through `instance`, or return `false` if no more solutions are found. Finally, a call to `commit()` will apply access control rules from the current solution.

The behavior is slightly different with utility functions; there is an added constraint that each new solution must have higher utility than the one before it. That means that it will not be possible to examine every valid solution, and the computation will stop at the first solution that cannot be improved. Of course, finding out that a particular solution cannot be improved might take a long time.

5.4.9 Time Limits

Some security policies can have an exponential number of solutions. Absent other constraints, there are 2^N possible results of `subsetOf` on N members. Finding a solution with maximum utility could therefore take a very long time.

To manage this issue, it is possible to configure a time limit after which the search stops. As explained in the previous section, an optimizing solver iterates through valid solutions, searching for those with increasing utility. By setting a time limit, we are effectively declaring that we are interested not in the optimal solution, but the best that can be found inside the limit.

Time limits are most useful in scenarios with utility expressions. They provide little benefit when iterating over all valid solutions, as the search can usually be stopped simply by not requesting another solution.

An optional parameter to `init()` or `resolve()` specifies a time limit in milliseconds. The countdown starts at the time the method is called, and carries across calls to `solve()`. If the time limit expires while `solve()` is running, it will stop the computation and return false, and all subsequent calls will also return false — same as if no more solutions can be found.

5.4.10 Access Control

When the scenario is solved and component assignments are known, the runtime emits specified access control rules. There are two available functions: `allow` and `deny`. Both take three arguments: actor, action and subject. Actors and subjects can be components, collections of components, or roles. Action must be a string.

Specifying a collection of actors or subjects is the same as specifying each actor and each subject one by one. Specifying a role applies the rule to selected members of that role.

If the ensemble is active, access control directives will be emitted. The framework takes a default-deny approach. If a triplet does not exist in emitted directives, the permission is denied. If an `allow` triplet exists, the permission is granted, unless a `deny` triplet also exists. This way, it is possible to grant a wide permission in an ensemble, but refine it in a sub-ensemble or a different situation-specific ensemble.

5.4.11 Notifications and Persistence

Using the function `notify`, it is possible to attach messages to components. This is the only way the framework can affect components (which are usually autonomous entities beyond our control). The notification feature serves two purposes.

First, it is possible to query the notifications inside an ensemble. This way it is possible to persist earlier configurations. Consider this example:

```
1  case class Reservation(room: Room) extends Notification
2
3  class SeatReservations(room: Room) extends Ensemble {
4    val alreadyReserved = workers.filter(_.notified(Reservation(room)))
5    val newlyReserved = oneOf(workers.filter(_.askingForReservation))
6
7    allow(alreadyReserved, "enter", room)
8    allow(newlyReserved, "enter", room)
9    notify(newlyReserved, Reservation(room))
10 }
```

Workers that have reserved seats in previous runs will still be granted the `"enter"` permission on subsequent runs. If we didn't attach the notification, they would lose the permission when the solution is rerun.

Second, a notification action is recorded as part of the generated access control rules. Users of the system can listen for these notification actions and forward them to components. This can be useful to, e.g., send a message to a worker's smartphone, to inform them about their seat reservation.

Notification messages must implement trait `Notification`. It is useful to define them as case classes, so that it is possible to filter them by value, as demonstrated in the example.

5.5 Implementing the Running Example

5.5.1 Overview

Our scenario consists of workers, which are assigned to projects; workrooms, which are also assigned to projects; and lunchrooms, which are unassigned. Workers, workrooms, and lunchrooms will be represented as components. Two distinct sub-problems exist, each with its own parameters: assignment of workrooms and assignment of lunchrooms. These can be naturally described as separate ensembles.

When the building is open, workers are allowed to enter all workrooms assigned to their project. We will create an ensemble for every project, and this ensemble will have the following roles:

- *project workers*, inhabited by all workers for that project
- *project rooms*, inhabited by all workrooms for that project

The ensemble will grant all project workers access to all project rooms.

Lunchrooms open at lunch time. Workers can indicate that they are hungry, which we represent as a knowledge field on the worker component. We would like to collect hungry workers in small ensembles, each granting access to a single

lunchroom. To accomplish that, we will create an ensemble for every lunchroom, with the following roles:

- *occupants*, inhabited by all workers currently in the room, plus all workers that have previously been assigned to the room
- *assignees*, inhabited by hungry workers who are not yet assigned

The ensemble will attempt to collect assignees from the pool of all hungry workers, up until room capacity is filled. An additional constraint is that every member of this ensemble must be assigned to the same project. That means that if there are existing occupants, new assignees must have the same project as them. If the room has no current occupants, new assignees can be selected from any project.

In addition, not all seatings are equally good. We want to use lunchrooms sparingly: given the choice between putting a worker into an empty lunchroom and an occupied one, the occupied should be picked, so that we keep the empty lunchroom available for other projects.

Once a satisfactory solution has been found, the ensemble will allow both occupants and assignees to enter the lunchroom, and notify new assignees that a seat was found for them.

5.5.2 Implementation

Following this description is a full listing of the policy for the running example, including definitions of components.

Lines 1–18 define component types. All rooms are of common supertype `Room`. Apart from name, the `LunchRoom` has a knowledge field `capacity`, stating its maximum occupancy.

`Worker`s have three knowledge fields: their assigned `project` (specified in constructor), their `hungry` status, and their current `location`. A helper method `isInLunchRoom` returns `true` if the worker's location is a lunchroom, as an example of "computed knowledge" from section 5.4.1

Next, case classes are defined on lines 20–24. Projects do not need to be components, but we still need structured information about them, particularly their list of assigned workrooms. We also create a case class for lunchroom assignment notifications.

Scenario class definition starts at line 26. Its inputs are lists of projects, workers, workrooms, and lunchrooms. An instance of this scenario class could represent a workday in a single building. Opening and closing times are part of the security policy and are hard-coded at lines 31–34.

Line 37 defines a variable to represent current time. In a real-life deployment, this might be represented by a reference to system clock; however, for our testing, it is useful to set the current time explicitly.

We also define a helper attribute `workersByProject` for easier access to lists of workers from the same project.

Class `RoomAssignment` at line 42 represents the policy root. It defines one pseudo-role, `hungryWorkers`, which is inhabited by all workers who are (1) hungry, (2) not currently in a lunchroom, and (3) not already assigned to a lunchroom. This will be the pool from which the lunchroom ensembles select candidates.

Furthermore, the root ensemble contains definitions of the two sub-ensembles for our two sub-problems.

Workroom ensemble is described by the class `WorkroomAssignment` on line 54. It takes a project definition as a parameter. Its situation predicate specifies that it is only active between `BuildingOpenTime` and `BuildingCloseTime`. The statically-assigned role `projectWorkers` is simply the list of workers for the ensemble's project. The grant at line 61 allows all `projectWorkers` to enter all `workrooms` of the project.

Lunchroom ensemble is represented by the `LunchroomAssignment` class at line 67, and takes a lunchroom as an argument. Like the workroom ensemble, it also has a situation predicate; this time specifying that it applies between `LunchOpenTime` and `LunchCloseTime`.

The first role, `occupants` at line 75, is a statically determined list of workers that are either (a) already assigned to the room, as determined by the appropriate `LunchRoomAssigned` notification, or (b) physically present in the room, as indicated by their `location` attribute.

We use `occupants` to calculate `freeSpaces`, the number of remaining seats in the room. Then, at line 82, we define the `assignees` role as a dynamically selected subset of `hungryWorkers`, limiting its size to the number of free seats.

The `eaters` role at line 84 is a union of `occupants` and `assignees`. It is not a meaningful role in the ensemble, but implementation-wise, it is needed to specify the constraint on the next line: all `eaters` must have the same value of their `project` attribute.

Lines 89–92 define the utility expression. Fuller rooms are preferred, which is expressed by setting the utility to the square of total number of used seats. This way, a solution that places two workers in the same lunchroom is measured as better than a solution that places each of them in a separate room.

Finally, `assignees` are notified of their assignment at line 95, and line 96 allows all `eaters` to enter the room.

Back at the `RoomAssignment` level, lines 99–102 create instances of the sub-ensembles. An instance of `WorkroomAssignment` is generated for every project, and an instance `LunchroomAssignment` is generated for every lunchroom. The `rules` call configures the sub-ensembles to be selected whenever their situation predicate is true, i.e., they are always active in their specified time-frames.

The constraint at line 105 ensures that all instances of the `assignees` role are disjoint, or in other words, that no worker can get a seat reservation in more than one lunchroom at the same time.

Line 108 instantiates the policy object and makes it available as an attribute.

5.5.3 Source Code

```
// Different types of rooms
abstract class Room(name: String) extends Component {
  name(s"Room:$name")
}
class LunchRoom(name: String, val capacity: Int)
  extends Room("Lunch" + name)
class WorkRoom(name: String)
  extends Room("Work" + name)

// Worker assigned to a project, can be hungry or not
class Worker(id: Int, val project: Project) extends Component {
  name(s"Worker:$id:${project.name}")
  var hungry = false
  var location: Option[Room] = None

  def isInLunchRoom: Boolean =
    location.map(_.isInstanceOf[LunchRoom]).getOrElse(false)
}

// Project with pre-assigned workrooms
case class Project(name: String, workrooms: Seq[WorkRoom])

// Notification for lunchroom assignment
case class LunchRoomAssigned(room: LunchRoom) extends Notification

class LunchScenario(val projects: Seq[Project],
                    val workers: Seq[Worker],
                    val workrooms: Seq[WorkRoom],
                    val lunchrooms: Seq[LunchRoom]) {
  // Opening times of the building and of the lunchrooms
  val BuildingOpenTime  = LocalTime.of( 7, 30)
  val BuildingCloseTime = LocalTime.of(21,  0)
  val LunchOpenTime     = LocalTime.of(11, 30)
  val LunchCloseTime    = LocalTime.of(15,  0)

  val DefaultNow = LocalTime.of(8, 42)
  var now = DefaultNow

  // mapping projects to lists of workers
  val workersByProject = workers.groupBy(_.project)

  class RoomAssignment extends Ensemble {
    name("assign workers to projects and rooms")

    // list of all hungry workers waiting for a lunchroom
    val hungryWorkers = workers.filter { w =>
      w.hungry &&
      !w.isInLunchRoom &&
      !w.notified[LunchRoomAssigned]
    }

    // Each worker assigned to a project can access all workrooms
    // assigned to that project when the building is open.
    class WorkroomAssignment(project: Project) extends Ensemble {
      name(s"assign workrooms to workers on project ${project.name}")

      situation { (now isAfter BuildingOpenTime) &&
                  (now isBefore BuildingCloseTime) }

      val projectWorkers = workersByProject.getOrElse(project, Seq.empty)
      allow(projectWorkers, "enter", project.workrooms)
    }
```

```
63
64      // Each hungry worker will get an assigned lunchroom so that
65      // no lunchroom is over capacity and workers from different
66      // projects do not meet in the same lunchroom.
67      class LunchroomAssignment(room: LunchRoom) extends Ensemble {
68        name(s"assign workers to lunchroom ${room.name}")
69
70        // Only activate when lunchrooms are open
71        situation { (now isAfter LunchOpenTime) &&
72                    (now isBefore LunchCloseTime) }
73
74        // list of previously assigned workers
75        val occupants = workers.filter { w =>
76          w.notified(LunchRoomAssigned(room)) ||
77          w.location.contains(room)
78        }
79
80        // newly-assigned hungry workers must fit into free space
81        val freeSpaces = room.capacity - occupants.size
82        val assignees = subsetOf(hungryWorkers, _ <= freeSpaces)
83
84        val eaters = unionOf(occupants, assignees)
85        constraint { eaters.allEqual(_.project) }
86
87        // Set the solution utility to square of the number of occupants,
88        // i.e., prefer many workers in one room over few workers in many rooms
89        utility {
90          val occupied = assignees.cardinality + occupants.size
91          occupied * occupied
92        }
93
94        // grant access rights and notify newly selected hungry workers
95        notify(assignees, LunchRoomAssigned(room))
96        allow(eaters, "enter", room)
97      }
98
99      val workroomAssignments =
100       rules(projects.map(new WorkroomAssignment(_)))
101     val lunchroomAssignments =
102       rules(lunchrooms.map(new LunchroomAssignment(_)))
103
104     // ensure that a worker is not assigned to more than one lunchroom
105     constraint(lunchroomAssignments.map(_.assignees).allDisjoint)
106   }
107
108   val policy = Policy.root(new RoomAssignment)
109 }
```

5.6 Reference

Scala is a strongly typed language, so a Scala-internal DSL is also strongly typed. In order to make this section more concise, however, we will be using simplified type signatures. The following simplifications are used:

- Whenever a function has bounded type parameters, we omit them in favor of the most general type.
- If a function does not return a value, the return type `Unit` is omitted.
- Whenever a function has a variadic argument (denoted with an asterisk), at least one argument must be provided.
- For every function with a variadic argument of type `T`, denoted as "`T*`", an overload exists that takes a single `Iterable[T]` argument instead.
- We use type name `Role` for brevity, but that type does not exist. The actual type is `MemberGroup[Component]`.

For readers not deeply familiar with Scala, we point out the *by-name parameters* feature. Whenever an argument type is prefixed with "`=>`", it is not evaluated immediately, but only when it is used. This allows us to write expressions that would fail at ensemble definition time, but work fine when the framework executes them.

5.6.1 `Component` class

`Component` must be used as a superclass of every component type. Only one function is available at declaration time:

`name(nm: String)`
: Set a name of the component. This is useful for debugging purposes, when printing out ensemble memberships.

Component instances also have several methods from trait `Notifiable` for querying received notifications:

`notifications: Iterable[Notification]`
: Return a collection of all `Notification` instances received by this component.

`notified(notification: Notification): Boolean`
: Query a specific notification. Return `true` if the exact specified notification was received by this component.

`notified[N <: Notification]: Boolean`
: Query a notification class. Return `true` if any notification of type `N` was received by this component.

5.6.2 `Integer` type

`Integer` is a generic reference to an integer number whose value might not be known until a solution is found. It is usually a result of operations on member group objects. Basic arithmetic operators on `Integer`s are overloaded to return `Integer`s and basic comparison operators are overloaded to return `Logical`s. Implicit conversion from `Int` is available, so that it is possible to mix `Integer` calculations with standard Scala math.

One notable imperfection is that the equality operator "`==`" cannot be overloaded in Scala. For comparing values of `Integer`s, use the triple-equals "`===`" operator instead. The operator "`==`" will compare object identities and return a boolean.

5.6.3 `Logical` type

`Logical` is a generic reference to a truth value which might not be known until a solution is found. It is usually the type of constraint operations. Basic boolean operators on `Logical`s are overloaded to return `Logical`s. Implicit conversion from `Boolean` is available, so that it is possible to mix `Logical` expressions with statically evaluated booleans.

5.6.4 `Ensemble` class

The security policy consists of a nested series of classes deriving from `Ensemble`. Most of the ensemble definition happens in the body of `Ensemble`, so this class provides most of the available functions.

`name(nm: String)`
: Set a descriptive name of the ensemble. This is useful for code documentation and for debugging purposes, when printing out ensemble memberships.

`utility(util: => Integer)`
: Assign an utility function to the ensemble. `util` is an `Integer` expression that is evaluated for each solution being tested. Refer to subsection 5.4.6 for detailed semantics.

`rules(ensembles: Ensemble*): EnsembleGroup`
: Register sub-ensemble(s) with static activation. Sub-ensembles registered via this function *must* be activated if possible.

  Each sub-ensemble must be registered with `rules` or `ensembles` to take part in the computation.

`ensembles(ensembles: Ensemble*): EnsembleGroup`
: Register sub-ensemble(s) with dynamic activation. Sub-ensembles that are registered via this function *can* be activated by the solver, if that leads to a good solution.

  Each sub-ensemble must be registered with `rules` or `ensembles` to take part in the computation.

oneOf(items: Component*): Role
oneOf(role: Role): Role

Define a role inhabited by *exactly one* of the specified components or inhabitants of the specified role.

allOf(items: Component*): Role

Define a role inhabited by *all* of the specified components.

This function is useful for explicit conversion of components to role objects. However, in most cases, components and collections of components are implicitly converted to roles as needed. E.g., the following two ensemble definitions are equivalent:

```
1  val members = for (_ <- 1 to 5) yield new Member
2
3  object WithAllOf extends Ensemble {
4      val role = allOf(members)
5      allow(role, "open", door)
6  }
7
8  object WithoutAllOf extends Ensemble {
9      allow(members, "open", door)
10 }
```

subsetOf(items: Component*): Role
subsetOf(role: Role): Role
subsetOf(role: Role, cardinality: Integer => Logical): Role

Define a role inhabited by a *subset* of the specified components or inhabitants of the specified role.

The optional argument `cardinality` specifies a constraint on the subset's cardinality. It is a function that takes an `Integer` argument, representing the subset's cardinality, and returns a `Logical` result configuring whether the cardinality is valid. It is possible to use Scala's placeholder underscore as a shortcut, i.e.:

```
val role = subsetOf(components, _ < 10)
```

unionOf(roles: Role*): Role

Defines a role whose members are a *union of specified roles*. Specifically, any component that inhabits one of the `roles` also inhabits the union role, and if a component inhabits the union role, then there exists at least one role in `roles` which the component also inhabits. This is mainly useful for specifying constraints over collections of roles that would otherwise be difficult to express individually; e.g., total size of the union must not exceed a specified number.

allow(actors: Role, action: String, subjects: Role)

Grant permission to each inhabitant of `actors` role to perform `action` on each inhabitant of the `subjects` role.

Through implicit conversions, it is possible to use a component or an iterable of components in place of any of the `Role` arguments.

`deny(actors: Role, action: String, subjects: Role)`

Deny permission to each inhabitant of `actors` role to perform `action` on each inhabitant of the `subjects` role.

Through implicit conversions, it is possible to use a component or an iterable of components in place of any of the `Role` arguments.

`notify(targets: Role, message: Notification)`

Send a `message` to each of `targets`. The message is persisted across solver runs, and its presence can be queried when forming ensembles. See subsection 5.4.11 for detailed semantics and subsection 5.6.1 for query methods.

Through implicit conversions, it is possible to use a component or an iterable of components in place of the `Role` argument.

`constraint(clause: => Logical)`

Set up a constraint that must be satisfied in every solution. The clause must be of type `Logical`, because it is propagated to the constraint programming engine, where it limits the search space. Therefore, it is possible to use role object expressions as constraints.

Multiple constraints can be specified in an ensemble and each one must be satisfied in a valid solution.

`situation(predicate: => Boolean)`

Set up a situation predicate. The predicate is evaluated *before* the solver starts processing the ensemble. If it evaluates to `false`, the ensemble is excluded from the solution.

`situation` has no effect in the root ensemble.

5.6.5 Member Groups

The base class `MemberGroup[T]` maintains a collection of components or ensembles. A so-called "role object" is in fact a `MemberGroup[C <: Component]`. Ensemble groups are represented by a subclass `EnsembleGroup`, which performs additional handling related to ensemble hierarchies. However, all functionality relevant to the DSL is defined in the base class, and thus identical for roles and ensemble groups.

For simplicity, we will use the type name "`Member`" to stand in for the member type of a `MemberGroup`.

The member group provides a notion of *selected members* — a subset of the member collection which is considered part of the solution. For instance, a role created with the `oneOf` function will have exactly one selected member.

All `Integer` methods can be used to build constraints with arithmetic and comparison operators. All `Logical` methods can be used as constraints directly, or combined with other constraints using boolean operators. For simplicity, the descriptions use terms like "true if" or "false if", but keep in mind that the truth value of `Logical` is only relative to a candidate solution.

`selectedMembers: Iterable[Member]`

List of `Member` instances that are selected for the solution.

Throws an exception if no solution has been generated.

`cardinality: Integer`

Cardinality of the group, i.e., the number of selected members.

`contains(member: Any): Logical`

True if the specified member is selected.

`containsOtherThan(member: Any): Logical`

True if at least one member other than `member` is selected.

`containsOnly(member: Any): Logical`

True if `member` is the only selected member.

`sum(func: Member => Integer): Integer`

Sum of values obtained by applying `func` on each selected member.

Typically used to sum the value of some knowledge field of the selected components. Scala's placeholder underscore is useful here:

```
val swarm = subsetOf(robots)
constraint { swarm.sum(_.arms) > 7 }
```

`all(func: Member => Logical): Logical`

True if predicate `func` holds for all selected members.

`some(func: Member => Logical): Logical`

True if predicate `func` holds for at least one selected member.

`allEqual(func: Member => Any): Logical`

True if result of `func` is the same for every selected member. In other words, the set of values yielded by `func` for each selected member has at most one element.

Typically used to ensure that components selected for a role all have the same value of some knowledge field, e.g., belong to the same team, have the same rank, etc.

`allDifferent(func: Member => Any): Logical`

True if result of `func` is different for every selected member. In other words, the set of values yielded by `func` for each selected member is the same size as the set of selected members.

Typically used to ensure that components selected for a role all differ in some knowledge field, e.g., no two components from the same team are picked.

`disjointAfterMap[T,M](funcThis: Member => T,`
`other: MemberGroup[M],`
`funcOther: M => T): Logical`

True if, after converting the selected members of this and the other group to a common type `T`, the resulting sets are disjoint.

There are two types of usage for this function. One type is ensuring that two groups of same or similar types of components are partitioned according to some variable — e.g., groups of workers disjoint over the projects they are working on.

The other type is ensuring that two groups of different types of components

do not mix with regard to some property. An example of this would be a sort of "wolf, goat, cabbage" scenario: the `eaters` role in a `Shore` ensemble must be disjoint with the `foods` role, so that none of the eaters will eat any of the food.

5.6.6 Collections of Member Groups

Special behavior is defined for collections of `MemberGroups`, which are typically obtained by mapping a collection of ensembles to one of their roles. The purpose of this behavior is to support a common idiom: ensuring that membership of a role in multiple ensembles does not overlap.

The following example assigns workers to teams in a way that no worker is assigned to two teams:

```
1  val workers: List[Worker] = /* ... */
2
3  class Team(val id: Int) extends Ensemble {
4    val teamMembers = subsetOf(workers, _ > 0)
5  }
6
7  val teams = rules {
8      for (i <- 1 to 4) yield new Team(i)
9  }
10
11 constraint { teams.map(_.teamMembers).allDisjoint }
12 constraint { teams.map(_.teamMembers).cardinality === workers.size }
```

A collection of member groups has the following methods:

`cardinality: Integer`
: Cardinality of the collection, i.e., the total number of selected members across all groups in the collection.

`allDisjoint: Logical`
: True if all groups in the collection are disjoint, i.e., no member is selected in more than one group.

5.6.7 `Policy` class

The `Policy` class represents the security policy in a single object, and provides methods to initiate solving and examine its results.

The `resolve()` method is the most straightforward way to interact with the policy object:

`resolve(): Boolean`
`resolve(limit: Long): Boolean`
: Find a valid solution and record security actions. If no utility expression is defined, returns the first solution that satisfies all constraints. If an utility expression is defined, returns the maximum utility solution, or the best solution that could be found within a time limit. Returns `true` if a solution was found, or `false` if not.

  When `limit` is specified, it is used as a time limit for the solving process, in milliseconds. If the method does not return before the time limit expires, the best solution found so far is recorded.

This is an all-in-one method that performs all solving steps automatically. To customize the solving process, it is necessary to use the methods below.

The following attributes are available as soon as a solution is attempted:

`exists: Boolean`
: True if a solution was found.

`instance: Ensemble`
: Reference to an instance of the root ensemble class. Through `instance`, it is possible to examine role and sub-ensemble assignment.

`actions: Iterable[Action]`
: Generated list of security actions, collected from all sub-ensembles. Contains objects of type `AllowAction`, `DenyAction` and `NotifyAction`.

`solutionUtility: Int`
: Total utility of the solution, if one exists. If the solution exists but has no utility expressions, this will return zero.

`allows(actor: Component,`
`action: String,`
`subject: Component): Boolean`
: Return true if `actor` is permitted to `action` on `subject`.

  See section 5.4.10 for semantics of access control queries.

In case a customization of the solving process is needed, the following methods are available to run the solving step-by-step:

`init()`
`init(limit: Long)`
: Reset the solver, configure time limit, delete all solutions and recorded security actions, and prepare for finding a solution.

  Time limit is in milliseconds and applies across all subsequent solving runs. If the time limit expires while a call to `solve()` is in progress, the solving process stops and `solve()` returns `false`. See section 5.4.9 for details.

  Must be called whenever the situation changes, otherwise the policy will reflect the previous state of ensembles, components, and the environment. In particular, must be called before the first call to `solve()`.

`solve(): Boolean`
: Find one solution. Return `true` if a valid solution is found, `false` otherwise.

  This method can be called repeatedly to iterate over all solutions. If no solution is found, and a previously-found solution exists, it will still be accessible.

  If no utility expression is specified, repeated calls will yield successive solutions. With a utility expression, each successive call will find a solution with higher utility than the previous one. If no such solution exists, `solve()` will return `false`, even if other solutions exist with equal utility. To find the solution with maximum utility, the following idiom is used:

  ```
  while (policy.solve()) {}
  ```

`commit()`

Commit current solution, generate security rules, and send notifications to components. Must be called before accessing `actions` for a new solution.

6. Implementation Details

6.1 Modeling for Constraint Solver

As explained in previous chapters, our framework employs a general solver for constraint satisfaction problems (CSPs). We have chosen Choco solver [23], a free and open-source constraint programming library for Java. Much of the work the framework does is converting the ensemble configuration into a problem input for Choco solver. This section explains the process in detail.

6.1.1 Overview of Choco Solver

Choco is a very fast constraint solver, designed with research applications in mind. It is written in Java 8, which allows us to integrate it easily with our Scala framework. Out of the box, Choco supports integer, boolean, and real values, and sets of integers. It allows the users to constrain domains of variables with basic equalities and inequalities, simple arithmetic expressions, boolean operations, automatons, if/then/else expressions, membership conditions, etc. It is also possible to implement custom constraints and their propagators.

Choco's key component is a `Model` object, which represents the problem currently being solved. Methods of this objects can be used to create *variables* of different types. For our framework, we make use of `BoolVar`, `IntVar` and `SetVar`.

Domain of a `IntVar` is $[-2^{32}, 2^{32} - 1]$. It can be represented in two distinct ways: a *bounded* domain is a contiguous interval between the upper and lower bound, while an *enumerated* domain can be any set of integers. The latter is obviously less performant and memory-efficient. However, for our use-case, we only use enumerated domains.

Domain of a `BoolVar` is $\{0, 1\}$, i.e., it is implemented as an integer. The important feature of `BoolVar` is that it can be used for *reification* of constraints; i.e., the resulting value of the variable represents whether a particular constraint was satisfied or not.

Domain of a `SetVar` is any set of integers. The default internal representation is a bit set, through Java's `BitSet` data type. We use set variables to represent ensemble or role membership. Integers in the set represent indices into an array that holds the member objects.

Each constraint is also represented as a Java object, generated by a factory method on the `Model` instance. Every constraint can either be *posted* or *reified*.

Posting a constraint means that it will represent a rule in the model. Every solution *must* satisfy that constraint.

Reifying a constraint associates it with a boolean variable. That variable resolves to true if the constraint is satisfied, or false if not. This allows expressing constraints as boolean expressions or if-then constructs based on the status of other constraints.

Choco solver provides a rich library of built-in constraint constructors. Arbitrary arithmetic operations on `IntVar`s are available, as well as tests of set membership, intersections, union, subset relation, etc. More complex construc-

tions can be built with boolean operators, and it is possible to set up constraint dependencies with `ifThen` and `ifOnlyIf` methods.

6.1.2 Modeling Basics

At its heart, ensemble assignment is a set membership problem. It makes sense that the basic building block would be a `SetVar`. The framework assembles components and ensembles into *groups*, and each group is represented by a `SetVar` over the entities' indices within the group: if an index is included in the `SetVar`, the corresponding entity is considered part of the solution, and we say it is *selected*.

Per-group sets mean that the same entity can have a different index in different groups. That makes certain operations awkward. Namely, in many cases we cannot use the built-in set operations, because for a given index n, $n \in A$ usually represents a different entity than the same $n \in B$. This makes operations like set union meaningless and we need to reconstruct them from primitives.

The alternative would be to assign an unique index to every entity and represent membership through these unique indices. However, that would lead to sparse sets, which would make their in-memory representations inefficient.

In case of ensembles, there is an implicit dependency relationship: a child ensemble can be selected if and only if its parent is also selected. Similarly, if an ensemble is not selected, no components are selected for any of its roles. These dependencies must be expressed as constraints on group membership.

6.1.3 Ensemble Selection and Situations

Several things must be true if an ensemble is selected: its parent must also be selected, its situation predicate must not be false, and all of its constraints must be satisfied. These requirements can be expressed in simple implications:

1. Root ensemble is always selected.
2. **IF** parent ensemble is not selected, **THEN** none of its children are selected.
3. **IF** ensemble is selected, **THEN** its situation predicate must be true, and its constraints must be satisfied.

No rule specifies when an ensemble *should* be selected, though. With the exception of the root ensemble, this means that the simplest solution is to not select any ensembles.

There are steps the user can take to avoid this problem. They can specify constraints in the root ensemble that enforce existence of some or all sub-ensembles, or they can configure utility functions in a way that maximizes the number of selected ensembles, etc. The drawback of these approaches is their obscurity. The problem itself is non-obvious, why should the policy designer think about resolving it in the first place?

Furthermore, the natural way of writing ensemble selection constraints is indirect, through the ensemble's properties or membership of its roles. But in such case, the variable representing ensemble selection remains free. This is a performance hit, in the form of larger search space: the solver will still attempt to

find solutions that satisfy the constraint with various configurations of selected ensembles. Using an utility function is similar: even though a solution with more ensembles is better, other solutions are not *invalid* and might still need to be examined.

In some cases, this behavior is desirable. In access control scenarios, however, it is rarely needed. Most cases can be solved by keying ensemble selection either to the resource being accessed, or to privileged actors — even if the corresponding set of actors or subjects ends up being empty.

To facilitate that, ensembles registered via the `rules` call have mandatory selection based on the situation predicate. The following rule is added:

4. **IF** parent ensemble is selected, **AND** child's situation predicate is true, **THEN** child ensemble is also selected.

Using (1) together with (4), it is easy to derive membership status of every ensemble registered with `rules`.

6.1.4 Constraint Propagation

To make use of Choco solver's constraint programming machinery, all constraints must be posted as instances of the `Constraint` class. To do otherwise would be highly inefficient; the solver would present us with an exhaustive list of solutions, exponential in the number of elements, and the framework would check constraint applicability on each one. This is the work we are trying to avoid by using a CSP solver in the first place.

As designers of the DSL, we want to allow the user to write constraint expressions with the language's operators. If we were designing an external DSL with custom parsing, we could convert the expressions to solver concepts. In an internal DSL, however, there is already a well-defined syntax and semantics for integer and boolean operations.

There are two kinds of constraints. Logical constraints place requirements on truth values of statements: a particular component is or is not a member of a particular role; a predicate holds for all or some members of the role, and similar. Arithmetic constraints place requirements on results of calculations and establish equalities or inequalities between integer values. Of course, an arithmetic constraint still boils down to a truth value of a statement. However, calculations with `Int`s in the host language must be properly converted to Choco's `IntVar`s and the arithmetic predicates on them.

6.1.5 Logical Constraints

Logical constraints are represented by the trait `Logical`, which emulates the built-in `Boolean` type. The trait defines the following operators: "`&&`" (conjunction), "`||`" (disjunction), "`->`" (implication), "`<->`" (equivalence), and unary "`!`" (negation). Implication is defined in terms of disjunction and negation, and equivalence is defined as an implication in both directions. The remaining operators must be defined in implementations of `Logical`.

There are three concrete implementations: `LogicalBoolean`, `LogicalBoolVar` and `LogicalLogOp`.

`LogicalBoolean` simply boxes the native `Boolean` type in a `Logical` interface. Any time a `Boolean` value comes into contact with a `Logical`, it is implicitly converted to `LogicalBoolean`. This allows us to define all operators with `Logical` operands.

Having a constraint type whose value is statically known is also useful for short-circuiting. A complicated boolean expression needs to be converted to Choco constraints if the values of variables are unavailable, but can be evaluated directly when they are known. The framework makes use of short-circuiting in several places; most notably, when generating a list of constraints for an ensemble. All ensemble constraints must be satisfied, so if a `LogicalBoolean(false)` is found, the whole set of constraints evaluates to false.

`LogicalBoolVar` is backed by Choco's `BoolVar`, and `LogicalLogOp` is backed by a `LogOp`. The difference is a matter of Choco implementation: `BoolVar` represents a *variable* in the constraint problem, while a `LogOp` represents a *constraint*, namely, a tree of boolean clauses in Choco's SAT constraint propagator. Both share a common interface `ILogic`. The framework defines a common superclass `LogicalWithILogic`, which implements the "`&&`" and "`||`" operators. Short-circuiting is used here too: if one of the operands is a `LogicalBoolean`, the result is either a fixed truth value or the value of the other operand. Only when both operands are `LogicalWithILogic`s, a new `LogOp` is created.

The only difference between a `BoolVar` and a `LogOp` is negation. A `BoolVar` class has a `not()` method; for `LogOp`, no such method is available and the negation must be constructed from a `nand` operation.

6.1.6 Arithmetic Constraints

Arithmetic constraints are represented by the trait `Integer`, which emulates the built-in `Int` type. The trait defines the basic arithmetic operators "`+`", "`-`" (including unary minus), "`*`", and "`/`"; and comparison operators "`===`", "`!=`", "`<`", "`>`", "`<=`", and "`>=`". All listed operators take `Integer`s as operands. Arithmetic operators return `Integer`s, while comparison operators return `Logical`s.

Note that the normal equality operator "`==`" cannot be used. See section 6.2.4 for technical details.

Only two concrete implementations of `Integer` exist.

Similar to the `LogicalBoolean` type, an `IntegerInt` boxes the native `Int` type, and an implicit conversion is available that ensures any `Int`s interacting with an `Integer` are automatically boxed.

`IntegerIntVar` encapsulates Choco's `IntVar`, which always represents a variable in the constraint problem. Unlike the logical constraints, there is no "tree" type to represent arithmetic. For every operation involving an `IntVar`, a new variable representing the result must be generated, and a constraint is posted that links the result variable to the operands. This means that unlike `Logical` types, `Integer`s need access to the solver instance to implement the operations. Because of that, the concrete implementations are defined inside the `SolverModel` class.

6.1.7 `MemberGroup` class

The `MemberGroup` lies at the core of the framework. It is a collection of elements of the specified member type, underpinned by a `SetVar`. Instances of `MemberGroup[Component]` represent roles, and a subclass `EnsembleGroup` represents ensemble groups (see section 6.1.8 for details). The purpose of `MemberGroup` is to provide mapping between `SetVar` values and member instances, and to wrap constraints on the `SetVar` in `Logical`s.

The class takes an iterable of candidate member instances as a constructor argument, converts it to a `Set`, so that no value is saved more than once, and then to an `IndexedSeq`, so that each member has a fixed index. The original order is lost in this process. An `allMembersVar: SetVar` is created, with a lower bound of empty set and an upper bound of all indices of the member set.

In addition, a `isActiveVar: BoolVar` is created that represents the group's active state. A `MemberGroup` can be active or inactive. If inactive, no members can be selected. This is enforced by the following constraint: **IF** `isActiveVar` is false, **THEN** `allMembersVar` must be empty.

Choco provides a number of built-in constraints on `SetVar`s through the `ISetConstraintFactory` interface. It is possible to define unions, intersections, and differences of collections of sets, specify that all sets are disjoint, find their minimum or maximum elements, and several other operations. Unfortunately, support for functional features, such as mapping elements to other values, is limited — e.g., it is not possible to specify a constraint that at least one member must satisfy some predicate, because there is no built-in way to map set members to predicates.

Constraints are often expressed in terms of universal or existential statements, so we need to implement these features using set primitives.

The universal predicate `all(func: Member => Logical)` is implemented as follows. First, the mapping function `func` is applied to all members, returning a per-member predicate. Then a constraint is generated for each member: **IF** x is selected, **THEN** predicate must apply. Finally, all member constraints are merged together with an **AND** operator.

It would be possible to implement the existential predicate `some(func)` in a similar way: the member constraint would be "x is selected **AND** predicate applies", and the constraints would be joined with an **OR** operator.

However, as it turns out, Choco internally represents logical trees in a Conjunctive Normal Form (CNF), and the generated expression is in a Disjunctive Normal Form (DNF). Converting from DNF to CNF exponentially increases the size of the formula. This is a problem, because there are as many DNF clauses as there are members in the group.

To avoid the issue, a different approach is used. One of the built-in constraints allows "channeling" a `SetVar` to an array of `BoolVar`s: `bools[i]` is true if and only if i is selected in the set. We construct an array of `BoolVar`s corresponding to results of `func` for each candidate, and create a new `SetVar` channeling this array. This gives us a set of "applicable members" for which `func` is true.

We then generate a constraint requiring that this set and the set of selected members are not disjoint. I.e., at least one selected member must also be an applicable member.

If required, it would be possible to use the same approach for the `all` predicate. The final constraint would need to specify that the set of selected members is a subset of applicable members.

Predicates `allEqual(func: Member => Any)` and `allDifferent(func)` are using a similar technique. We set up channeling between the set of selected members and set of result values: each unique result of `func` is assigned an index, and that index is selected in the value set if there is at least one selected member with that result value.

Given this channeling, the `allEqual` constraint specifies that cardinality of the value set must be 1 (or 0 if the selected member set is empty), and `allDifferent` specifies that cardinality of the value set must be equal to cardinality of the selected member set.

`disjointAfterMap` is a predicate that applies a function to the current member set, and a different function to another member set. The predicate is satisfied if the results of the mappings are disjoint. Value channeling is used in this predicate too, except the indices for values are taken from a common set. It is then straightforward to set up a disjoint constraint between the two channeled value sets.

The method `sum(func: Member => Integer)` returns an `Integer` representing the sum based on membership. Choco has a built-in function that takes the sum of selected members, given a statically-known set of "weights", or values, for each member. If all results of `func` are of type `IntegerInt`, meaning that their values are statically known, we use Choco's builtin. Otherwise it is necessary to construct the sum from `IntVar`s.

An array of `IntVar`s is created and each is conditionally set either to zero (if the corresponding member is not selected) or the result of `func`. The resulting sum then adds together all such `IntVar`s.

The membership predicate `contains(member)` can be converted to Choco's membership constraint directly. Similarly, `containsOnly(member)` specifies the membership and enforces that cardinality of the set is exactly 1.

`containsOtherThan(member)` is only slightly more complicated. If the `member` is not selected (or not in the set of candidates at all), it is enough to ensure that the set cardinality is at least 1. Otherwise the cardinality must be at least 2 — one for the selected member, one for the required other element.

6.1.8 Ensemble Hierarchy

Apart from constraints, situation predicates, utility expressions and other miscellaneous data, ensembles hold sub-ensembles, and roles and their members. Figure 6.1 illustrates this structure.

One notable feature is that an ensemble does not hold its members directly. Instead, it holds `MemberGroup`s. Selection parameters in each group are independent from other groups; that way it is possible to have dynamically selected `ensembles` and selection enforcement by `rules` in the same ensemble.

Ensembles also keep track of their own selection status. If an ensemble is selected in its parent group, it can select its own members; more precisely, its groups are allowed to select their members.

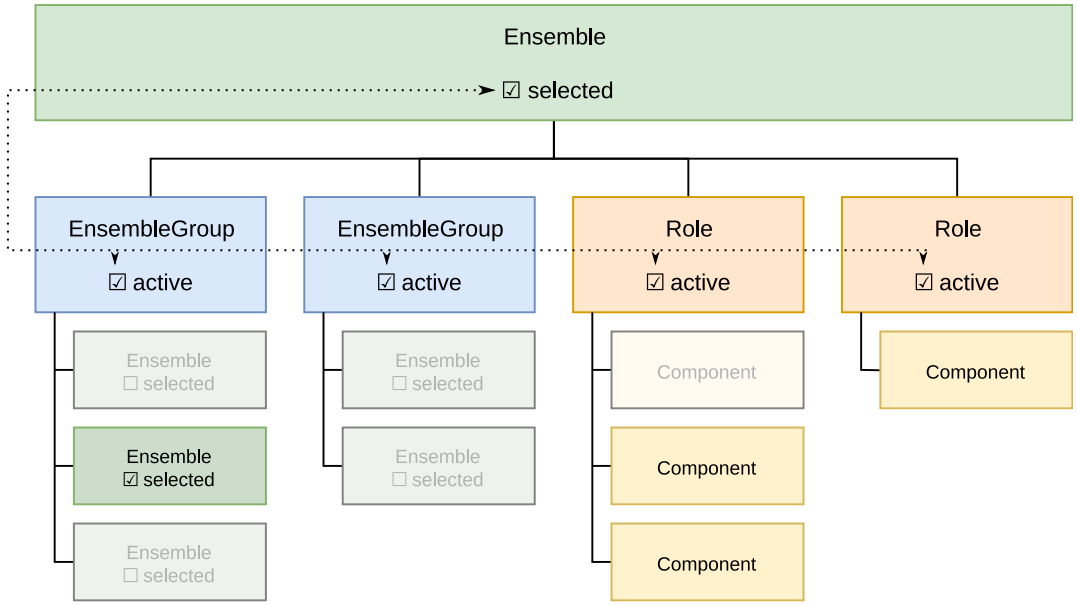

Figure 6.1: Ensemble object diagram

To facilitate this, ensemble has an `isSelectedVar: BoolVar` representing the selection status. When adding a role or an ensemble group, its `isActiveVar` is set to equal the parent ensemble's `isSelectedVar`, as indicated on figure 6.1 by the dotted line. If an ensemble is selected, all its groups become active and can select members. If it is not selected, its groups are inactive and the whole branch of the hierarchy is effectively turned off.

If an ensemble is part of an ensemble group, its `isSelectedVar` is reified with the corresponding membership constraint in the group's `allMembersVar` (plus the result of its situation predicate). The root ensemble is not a member of any groups, so its `isSelectedVar` is bound to true by the policy object.

6.2 From Scala to DSL

The Scala language has many features which make it suitable as a host language for an internal DSL. The ability to use blocks of code as expressions allows us to define custom language constructs. By-name parameters enable delayed evaluation, a necessary feature for writing declarative code in otherwise mostly imperative language. Arbitrary nesting, together with code in class bodies, makes the resulting language flow feel natural. Scala's type system allows us to retain advantages of strong typing and build generic containers that do the right thing — and at the same time, type inference is powerful enough so that users of the DSL almost never come into contact with type annotations. Finally, operator overloading lets us emulate primitive types with custom classes, and implicit conversions ensure that the emulated types interoperate with the built-in ones.

This section describes the overall architecture of the framework, and technical details of all language constructs used in the DSL. Although we give a brief overview of each language feature that we are using, this section is mostly intended for readers already familiar with the Scala language.

6.2.1 DSL Constructs

The DSL uses a number of distinct language constructs. An ensemble definition is, in Scala terms, implementing a child class:

```
class MyEnsemble extends Ensemble { /* ... */ }
```

Creating a dynamic role is realized as creating a member field from the result of a function call:

```
val role = subsetOf(members)
```

The call runs directly in the class body. In Scala, this means that it is executed as part of the constructor.

Scala does have first-class functions, but it does not have *top-level* functions. Each function must be defined within a class. Functions that are available at ensemble definition time are in fact methods of the class `Ensemble`. Their results are tied to the enclosing `Ensemble` instance, which is also where the registration actually happens; if the result of `subsetOf` in the above snippet was not saved to a member variable, the selection from `members` would still happen behind the scenes.

This is also one of the reasons ensemble instances must be explicitly registered with `rules` or `ensembles`: without the registration call, the enclosing instance would not know about the nested instances. While it would technically be possible to examine the class through reflection and find all `Ensemble` instances in member attributes, this would be needlessly fragile in practice.

Apart from "functions", the DSL also makes use of "constructs". The following code looks distinctly different from the above:

```
utility {
    val occupied = occupants.size + assignees.cardinality
    occupied * occupied
}
```

Although `utility` looks more like a keyword or a flow-control statement, it is in fact a normal method. Its definition could look like this:

```
def utility(util: Integer)
```

I.e., it is a method that takes a single argument of type `Integer`. The special syntax is enabled by two useful properties of Scala.

One, code blocks are expressions whose value is the last expression in the block. In this case, the value of the code block is "`occupied * occupied`". Given that a code block is an expression like any other, it is possible to use code blocks in unusual places, such as function arguments. The following is a contrived but perfectly valid example of Scala code:

```
1 val maximum = math.max(
2   {
3     val a = someFunc()
4     if (a > 5) a
5     else {
6         val b = someOtherFunc()
7         a + b
8     }
9   }, 0)
```

Two, Scala allows omitting parentheses in certain conditions. In particular, when calling a function with one argument, it is possible to omit the enclosing

parentheses if the argument is a code block enclosed in curly braces. The following three statements are equivalent:

```
utility(3)
utility({ 3 })
utility { 3 }
```

This way it is possible to define methods that can behave as custom language constructs. Note that this feature is particularly useful in combination with by-name parameters. See section 6.2.7 for detailed explanation.

6.2.2 Ensemble Structure

Each `Ensemble` instance contains the following data:

- Collection of role objects, representing dynamically assigned roles
- Collection of `EnsembleGroup` objects for each `rules` or `ensembles` call
- Collection of constraints, in the form of callables that return a `Logical`
- Collection of associated security actions and notifications
- Situation predicate, if defined
- Utility function, if defined

For each type of data, methods are available for configuring this data at ensemble definition time.

As the total number of methods is rather large, and the related functionalities are mostly independent, each is implemented in a separate trait. We maintain separation of concerns between traits and declare their dependencies through inheritance and self-type annotations.

Trait inheritance works the same as normal class inheritance. Most of ensemble traits inherit from `Initializable`, allowing them to override the `_init` method and access the solver instance; see section 6.2.6 for details. Inheritance also allows us to inject implicit conversions from `CommonImplicits` trait.

Self-type annotation in a trait enforces that any concrete class implementing that trait must also conform to the declared self-type. This usually means that it must implement some other trait. For example, the following self-type in trait `WithRoles` informs the compiler of a dependency on two other traits:

```
trait WithRoles {
  this: WithConstraints with WithSelectionStatus =>
  /* ... */
}
```

The `WithRoles` trait is then allowed to access members of the other traits on self.

Functional difference between inheritance and self-type annotation is subtle, but the takeaway is that self-types define a looser coupling, because they do not lock-in the inheritance hierarchy. We use self-types to declare that a trait *uses* a functionality from another trait, but is not an extended version of it.

The following list describes the individual traits of `Ensemble`.

`Initializable`

Provides an initialization hook, plus access to the solver instance through the `_solverModel` property.

`CommonImplicits`

Defines implicit conversion from `Int` to `Integer`, from `Boolean` to `Logical`, and adds methods to collections of `MemberGroup`s. This makes those implicit conversions available in ensemble definition context.

`WithName`

Provides a `name` property. Mainly for debugging purposes.

`WithActions`

Provides `allow`, `deny` and `notify` functions, and a method to collect results of these functions from sub-ensembles. To achieve that, it has a dependency on `WithEnsembleGroups`.

`WithConstraints`

Provides the `constraint` function, storage of registered constraints, and functionality related to converting constraints to solver objects.

`WithEnsembleGroups`

Provides the ability to register groups of sub-ensembles, and the `rules` and `ensembles` methods. Because groups must activate themselves based on the parent ensemble selection status, this trait requires `WithSelectionStatus`.

`WithRoles`

Provides the ability to register roles via role functions. Because roles implicitly use constraints, and the `subsetOf` function allows adding a constraint on the subset cardinality, this requires the `WithConstraints` trait. In addition, `WithSelectionStatus` is required to support group activation based on ensemble selection status.

`WithSelectionStatus`

Provides a `BoolVar` indicating the ensemble's selection status.

`WithUtility`

Provides methods to define and query the utility expression of the ensemble, and collects utility value from sub-ensembles. To access sub-ensembles, this trait requires the `WithEnsembleGroups` trait.

The `Ensemble` class itself defines the `situation` function and handling of the situation predicate. While it would be straightforward to extract this functionality to a separate trait, we opted to keep it in `Ensemble`. The functionality is small enough that it does not clutter the class, esp. given that `Ensemble` is otherwise empty; and it does not figure in the dependency graph, so extracting it would have little benefit for the rest of the code (unlike `WithSelectionStatus`, which is also small and uncomplicated, but is a dependency of two other traits).

6.2.3 Implicit Conversions

Scala's implicit conversions allow values of one type to be automatically converted to a different type, if they appear in a context where the target type is required. Two kinds of implicit conversions are available. With `implicit def`, a function returning the target type is applied to the value in question. An `implicit class` creates an instance of a new class, effectively allowing to add new methods to existing types.

Code that uses implicit conversions is less readable, because there are non-local hidden calls. In the following example, an implicit conversion adds a method named `foo` to a value of type `Int`. It is impossible to know what is `foo` just from this snippet; one would need to locate all methods named `foo` in the codebase and then figure out which implicit conversion is in play.

```
val x: Int = 7
val y = x.foo()
```

For this reason, it is usually better to avoid implicit conversions in normal code. However, they can be extremely helpful when creating a DSL.

For an implicit conversion to take effect, its function or class must be in scope: either defined in the same class (or a parent class), or explicitly imported.

We use implicit conversions for several different purposes. The most common one is upgrading `Int`s to `Integer`s and `Boolean`s to `Logical`s. This is important for seamless interoperability of native integer or boolean expressions with solver constraints, as described in section 6.1.4. The appropriate conversion functions are defined in trait `CommonImplicits`, which is mixed in where required. Importantly, it is mixed into the main `Ensemble` class, thus ensuring that user code will have these conversions in scope whenever defining an ensemble.

Implicit conversions also help cut down on the number of different method overloads. The method `unionOf` accepts either an iterable of `MemberGroup`s, or a variable number of `MemberGroup` arguments. However, we want to support using `Component`s for `unionOf` directly. Otherwise, every time the user wanted to make an union of a role and a fixed list of components, they would need to specify a role for the component list. Scala has no support for union types, so it would be impossible to specify a function that takes a variable number of "`MemberGroup` or `Component` or `Iterable[Component]`" arguments.

Instead, an implicit conversion from `Component` or `Iterable[Component]` to `MemberGroup[Component]` is defined, allowing the users to use either roles or fixed sets of components interchangeably.

Similarly, the `allow`, `deny` and `notify` verbs should accept components, list of components, or roles as their arguments. In the `notify` case, it is simple enough to provide three overloads for three acceptable argument types. However, `allow` and `deny` have two arguments of this type. A full set of overloads would need 9 variants of each verb. Instead, each of these verbs only accepts `MemberGroup`s as arguments, and implicit conversions ensure that components and lists of components are usable as well.

Two distinct implicit conversions are used with ensemble collections. First, an `EnsembleGroup` converts to a list of all its members, so that methods like `map` can be used on results of `rules` call directly. Second, an implicit class

`WithMembersIterable` adds methods `cardinality` and `allDisjoint` to collections of `MemberGroup`s. This enables the following common idiom:

```
val r = rules(/* list of ensembles */)
constraint { r.map(_.someRole).allDisjoint }
```

The variable `r` of type `EnsembleGroup` is converted to `Iterable[E <: Ensemble]`, so that `map` can be applied on it. This iterable is a plain sequence of objects of the appropriate ensemble type, which has a member role `someRole`. The result of the map is therefore a plain sequence of `MemberGroup` objects.

This sequence is then implicitly converted to a class `WithMembersIterable`, which has the method `allDisjoint`, ensuring that no member is selected more than once for `someRole`.

Note that the implicit conversion turns the `EnsembleGroup` into a list of *all* its members, not just the selected ones. This is because at the time the conversion is used, the list of selected members is not yet known.

The provided methods work due to the fact that inactive (not selected) ensembles do not have any members. Empty roles add zero to `cardinality`, and are guaranteed to be disjoint with non-empty ones.

An implicit conversion from an `EnsembleGroup` to its `allMembers` carries some risk of confusion when used in an inappropriate context. E.g., when inspecting a solution, the user might inadvertently invoke the implicit conversion and their code will visit even ensembles that are not part of the solution. We consider this risk acceptable because, again, ensembles that are not selected do not have any members.

6.2.4 Operator Overloading

Method names in Scala are allowed to use special characters, and operators are defined as regular methods with the appropriate name. For instance, overloading the "`+`" operator on an `IntegerInt` type could look like this:

```
def +(other: Integer): Integer = other match {
    case IntegerInt(value) => new IntegerInt(value + this.value)
    // ...
}
```

Combined with implicit conversions, both of the following will work:

```
val leftHand: Integer = someInteger + 5
val rightHand: Integer = 5 + someInteger
```

The first case is simple. The expression "`someInteger + 5`" is interpreted as "`someInteger.+(5)`". Because `someInteger` has a "`+`" operator, Scala tries to use it. The operator is defined for an `Integer`, and there is an implicit conversion available that upgrades `5` to `Integer`. The resulting instance is passed to the "`+`" method.

The second case is slightly more complicated. As before, the expression translates to "`5.+(someInteger)`". In this case, the built-in `Int` type does not have an appropriate "`+`" operator for handling anything `someInteger` could be converted to. It is necessary to search the list of possible implicit conversions of the left-hand operand, to see if one of them comes with an appropriate definition of "`+`".

This process works fine for most operators, but fails for "==". The problem here is that "==" is defined on the root class `Any` with the following signature:

```
final def ==(arg: Any): Boolean
```

The method is final, so it cannot be overridden. It calls the `equals` method internally, which is available for overriding, but then the return value is `Boolean`, which is not appropriate when we need the result as a `Logical`.

It is possible to overload a specialized version of "==" with a different return type. For the `Integer` trait, the following definitions are desirable:

```
def ==(other: Integer): Logical
def ==(other: Int): Logical
```

We need to have a special variant for `Int`, because implicit conversion won't help us here: "`someInteger.==(5)`" can use the default implementation for type `Int`.

The problem with this is that the integration is not seamless. If the `Int` value appears on the right-hand side, it will always be able to use the built-in "==". This is made worse by the fact that both of the following will compile:

```
val l1: Logical = someInteger == 5
val l2: Logical = 5 == someInteger
```

The type of "`5 == someInteger`" is `Boolean`, which has an implicit conversion to `Logical`, and so it is valid in every context that requires a `Logical` type. But the result of the comparison is always false, because we are comparing unequal types.

The `Integer` trait uses a triple-equals "===" operator for comparing equality in the desired way. This is a convention used by many other Scala-based DSLs and language customizations. Scala compiler will emit a warning whenever two unrelated types are compared with "==", which should notify the user that they made a mistake.

We considered raising an exception in an overridden `equals` method, but that would silence this warning, and would not work when `Int` is the left-hand operand anyway.

6.2.5 Type Bounds

One of the features required from the DSL is seamless working with different ensemble and component types. Consider the following code:

```
1 class Worker(val rank: String) extends Component
2 val workers: Iterable[Worker] = /* ... */
3 class Root extends Ensemble {
4     val supervisor = oneOf(workers)
5     constraint { supervisor.all(_.rank == "supervisor") }
6     allow(supervisor, "supervise", workers)
7 }
```

The "`supervisor.all()`" call must accept a function whose argument is of type `Worker` — otherwise, it would be impossible to access the `rank` attribute. That means that `supervisor` must carry the information that its member type is `Worker`, not just a generic `Component`. At the same time, the `allow` call must accept `supervisor` as its argument.

Roles are of type `MemberGroup`, which is generic over its member type. The signature looks like this:

```
class MemberGroup[+MemberType]
```

The "`+`" symbol indicates that the class is *covariant* in the `MemberType` argument. That means that for superclass `A` and its subclass `B`, `MemberType[B]` is considered a subclass of `MemberType[A]`.

Method `MemberGroup.all` accepts functions of type `MemberType => Boolean`. In our example, member type is `Worker`.

Method `accept` takes an argument of type `MemberGroup[Component]`, so it accepts a `MemberGroup[Worker]` object.

The type signature of `oneOf` looks like this:

```
def oneOf[C <: Component](items: Iterable[C]): MemberGroup[C]
```

The method takes an iterable of objects of concrete type `C`, which is a subtype of `Component`, as indicated by the "`<:`" symbol. It returns a `MemberGroup` parameterized by the appropriate concrete type. Scala's type inference makes sure that when passing an iterable of `Workers`, the returned `MemberGroup` will have `Worker` as its member type.

Ensemble groups work in a similar way, except that `EnsembleGroup` is a subclass of `MemberGroup` specialized so that its member type must be a subtype of `Ensemble`. This is the signature:

```
class EnsembleGroup[+EnsembleType <: Ensemble]
    extends MemberGroup[EnsembleType]
```

The "`+`" symbol indicates type covariance: an `EnsembleGroup` of a concrete type of ensemble is a subclass of `EnsembleGroup[Ensemble]`. The "`<:`" symbol indicates a type bound: the member type must be a subtype of `Ensemble`.

Similar to role functions, `rules` and `ensembles` must be defined with appropriate type signatures, so that they return the appropriate variant of `EnsembleGroup`.

6.2.6 Initialization

Every ensemble and group object requires an instance of the solver, through which variables and constraints are constructed. This is not available at ensemble definition time.

In terms of Scala, the whole policy is a series of instances of nested classes. In order to propagate a solver object, the user of the DSL would be required to add parameters to their ensemble class definitions and manually propagate them when instantiating sub-ensembles. This violates separation of concerns: the solver object is an implementation detail of the framework, not something the users should care or even know about.

Instead, the ensemble tree is constructed without access to the solver, and the solver object is passed to it in a separate initialization step. The `Initializable` trait defines a method `_init`, which allows classes and traits to hook into the initialization process. The `Policy` object calls `_init` on the root ensemble, and the `Ensemble` class is responsible for propagating the call to its ensemble groups and role objects. Ensemble groups are in turn responsible for propagating the call to sub-ensembles. This way it is ensured that every part of the policy tree is initialized.

The initialization runs in three phases:

1. `ConfigPropagation` propagates a `Config` object, which contains the newly created instance of the solver.

2. `VarsCreation` is the phase where solver variable objects are created. The class `MemberGroup` initializes its set and activation variables, and `Ensemble` initializes its selection variable.

3. `RulesCreation` can use variables created in the previous step to generate and post solver constraints.

While it might be possible to perform the initialization in a single pass, it would make it more difficult to implement properly. Developers would need to be extra careful about initialization order, e.g., make sure that an initialization step is not using variables from a child (or parent) object whose initialization isn't finished yet. Multi-phase initialization removes this source of fragility. Phase (1) can be done by every object individually. Phase (2) only requires that the solver is available, i.e., that phase (1) has finished everywhere, and phase (3) can rely on the fact that *all* variables across all objects have been generated in the previous phase.

Several component traits of `Ensemble` implement their own `_init` steps. This is possible because Scala has a well-defined trait linearization order, and calling `super._init()` within a trait will invoke the `_init` implementation in the next trait up.

6.2.7 By-Name Parameters

As stated in section 6.2.6, the solver object is not available at ensemble definition time — or, more precisely, at ensemble instantiation time. Most of constraint definition and role creation is done in class body, and this code actually runs when the corresponding instance is constructed. This poses two related but distinct problems.

First, consider a role that relies on external data:

```
1  class Scenario(val workers: mutable.ArrayBuffer[Worker]) {
2    class Root extends Ensemble {
3      val role = subsetOf(workers)
4      constraint { role.cardinality > 5 }
5    }
6    val policy = Policy.root(new Root)
7  }
8
9  object Scenario {
10   def main() {
11     val scenario = new Scenario(/* ... */)
12     scenario.policy.resolve()
13     scenario.workers.append(new Worker(/* ... */))
14     scenario.policy.resolve()
15   }
16 }
```

One would expect the second `resolve` call to use the newly added `Worker` instance. However, at that point, the ensemble seems to already exist, and the

old value of `workers` was used to construct `role`. Groups register their members at construction time (see section 6.1.7), so the original argument no longer affects the set of role members.

It would be useful if the framework could somehow "refresh" the policy definition using new data.

Second, the `constraint` statement works with `role.cardinality`. We expect the result to be a `Logical` object referencing an underlying `BoolVar`. But as stated above, at this point the solver instance is not available, so a `BoolVar` cannot be created.

Both of those issues are solved using *by-name parameters*.

In Scala, functions are first-class objects, so naturally it is possible to pass them as arguments explicitly. That is what happens in this statement:

```
val role = subsetOf(members, _ > 1)
```

The expression "`_ > 1`" is a shortcut for "`x => x > 1`". This is a function of type "`Integer => Logical`", i.e., takes an `Integer` argument and returns a `Logical`.

By-name parameters are basically parameters that are passed as functions implicitly. The signature of a `constraint` verb is:

```
def constraint(clause: => Logical): Unit
```

`clause` is a by-name parameter, and its type is "expression of type `Logical`". The important point is that unlike normal parameters, the expression is not evaluated until actually used in the `constraint` method body. Furthermore, the following works:

```
val clauseFun: () => Logical = clause _
```

Using the above syntax, the "expression" parameter is converted to a normal function with no arguments. The `constraint` method stores all such functions in a collection and only runs them in the `RulesCreation` initialization phase, when the solver is available and all variables are already created.

The verbs `situation` and `utility` also take by-name parameters and call them at appropriate times.

Section 5.4.7 specifies that the root ensemble must be instantiated in the `Policy.root` method. The reason is that the argument of this method is actually a by-name parameter too. The policy object does not store the instantiated policy tree, it stores a *builder* function which constructs it. On every call to `init()` or `resolve()`, the policy tree is instantiated from scratch, and so it reflects the current values of all external variables.

6.2.8 Variadic arguments

Many functions that accept iterables also accept variadic arguments. For example, it is possible to call `oneOf` in two ways:

```
val a = oneOf(listOfMembers)
val b = oneOf(memberA, memberB, memberC)
```

Scala natively supports variadic arguments. For the following two signatures, the type of `items` is the same:

```
def oneOf(items: Seq[Component])
def oneOf(items: Component*)
```

One minor issue with the second signature is that it allows the list of arguments to be empty. This does not cause any problems in practice — after all, the user could as well pass an empty list of components — but still, it would be cleaner to disallow code like "`val x = oneOf()`".

To achieve this, all variadic functions actually have two arguments: one is mandatory, the other is variadic. This is the definition of `oneOf`:

```
1 def oneOf[C <: Component](itemFirst: C, itemRest: C*): Role[C] =
2     oneOf(itemFirst +: itemRest)
```

Type of `itemFirst` is `C`, and type of `itemRest` is `Seq[C]`. The code prepends the first item to the list and calls the other overload of `oneOf`. It is possible to call `oneOf` with one or more arguments; calling with no arguments is illegal.

7. Evaluation

To evaluate practicality and performance characteristics of our approach, we have designed a number of testing scenarios. Each tests a particular aspect or feature of the DSL.

7.1 Methodology

In this chapter, we take *scenario* to mean a particular situation or task with possible variables. A *configuration* is an instance of the scenario, where the variables are bound to particular values. A *test case* is a set of configurations to be evaluated. In a test case, each configuration is usually tested many times, and each attempt is called a *test run*.

E.g., a scenario can be described as: "pick several Workers and one Leader from a pool of Persons". A configuration of that scenario can be "pick one Worker and one Leader from a pool of 15 Persons". One test case involving this scenario can be a set of configurations "pick three Workers and one Leader from a pool of N Persons", with $N \leftarrow 10, 15, 20...100$. Another test case would be "pick N Workers and one Leader from a pool of 100 Persons", with $N \leftarrow 1, 2...20$

Each scenario is represented by a class containing the necessary objects and a definition of the associated ensemble(s). A companion `Spec` subclass is a concise description of scenario parameters and an interface for the test runner. It generates appropriate instances of the scenario class on which each test run is performed.

Because Scala runs in a Java Virtual Machine (JVM) with Just-in-Time compilation, it is necessary to "warm up" for the given scenario. Otherwise, earlier test runs would be slower than the later ones, where the JIT has had time to catch up and pre-compile hot paths. To prevent this, the smallest configuration is run repeatedly for 10 seconds of wall-clock time.

We run each configuration a specified number of times and record the time to find a solution, utility (if specified), and peak memory usage for each run. Raw data from the results can be found in the accompanying archive in `results` directory, organized by test case name. Graphs are generated with Python, using scipy [5], pandas [22], and matplotlib [6]. It is possible to regenerate them by executing the script `python/all.sh`.

All results are generated on an Intel® Core™ i5-6600K CPU, running on a single core at 3.90 GHz.

7.1.1 Time Limits

In most scenarios, we set a solver time limit to 30 seconds per test run. This number was chosen as an arbitrary cut-off point. Preliminary experiments showed that observed trends are robust even below 30 seconds, and increasing the time limit has diminishing returns in terms of solution quality (see also section 7.3.4). Setting a higher time limit would significantly prolong test times while providing very little additional useful data.

Moreover, 30 seconds seem like a reasonable real-world time limit. The framework is designed for highly dynamic scenarios and needs to be able to respond to changing conditions as they happen. Waiting more than a couple seconds for access control decisions is not acceptable in terms of user experience, and could possibly even endanger the security goals of the system. Taken in this context, even 30 seconds is possibly too long. Our results show that in real-world scenarios it might be possible to lower the time limit further.

7.1.2 Memory Usage

The JVM provides a memory-managed environment with asynchronous garbage collection, which makes it difficult to obtain reliable memory usage information. As a rough measure, our test runner uses garbage collector notifications from the Java Management Extensions API [21] to record peak memory usage during each test run. We trigger a GC cycle between test runs, and attempt to wait until the cycle is complete. This is inherently unreliable, however, and there is no way to ensure that all unreachable objects have been released.

As measured with this method, most scenarios use less than 200 MB of memory. These results cannot be used to draw any conclusions: any trends are lost in the variance, allocator granularity, and general overhead of the JVM runtime.

In section 7.2, we test relatively large configurations where memory consumption is significant. Although the data about memory usage is still very noisy, it provides some useful insights. We note that the measured data does not reflect true memory requirements; many scenarios can run with much less memory available, by doing garbage collection as needed during the computation.

We have configured the JVM to use a maximum of 10 GB of RAM for its heap. This ensures that our chosen test configurations never need to deal with memory pressure, which considerably reduces timing jitter.

7.2 Basic Benchmarks

To evaluate performance of the basic building blocks, we have designed a scenario that selects one of a list of 100 000 members. With the `oneOf` function, there is already a constraint specifying that the cardinality of the selection is 1. From there, we generate large numbers of additional meaningless constraints. Given that finding a valid solution is trivial, results of this test provide information about per-constraint processing time and memory consumption.

To evaluate simple constraints, we generate a number of `constraint` statements specifying that selection cardinality must be smaller than i for an increasing i. In the constraints test case, the total number of constraints N goes from 50 000 to 1 000 000.

To evaluate `Logical` expressions, we generate the following expression:

```
val condition: LogicalBoolVar = role.cardinality === 1
condition && condition && condition && /* ... repeated N times */
```

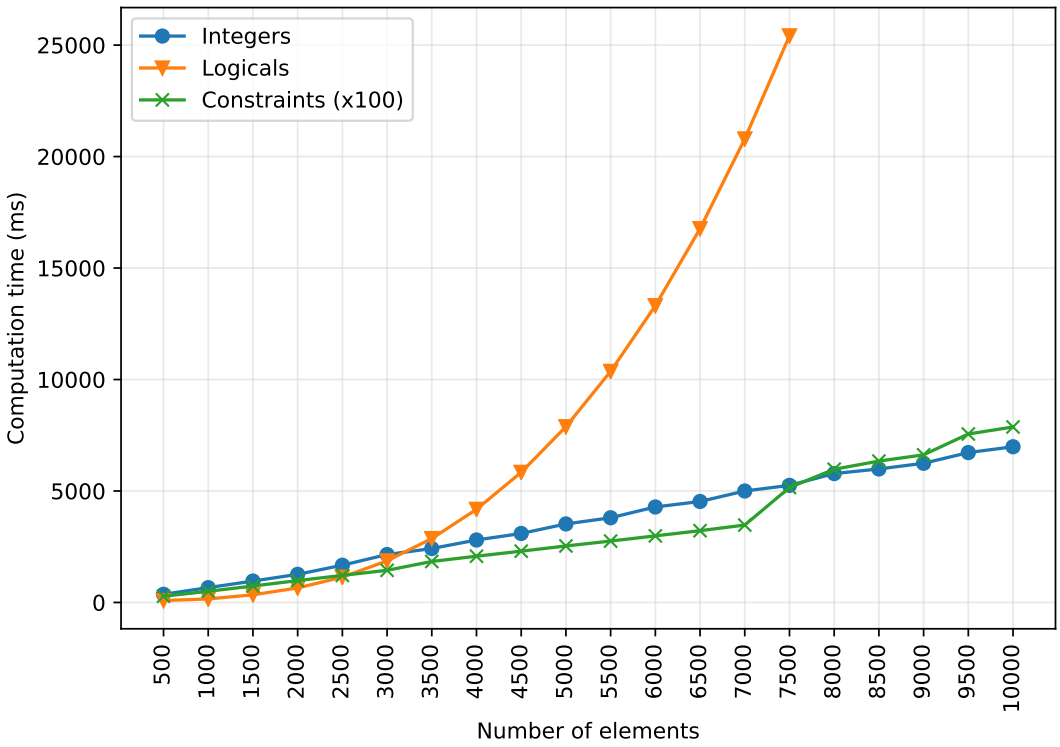

Figure 7.1: Computation times for basic building blocks

`condition` creates a single `BoolVar` in the solver, and the repeated use of the "`&&`" operator constructs an unbalanced tree of `LogOp` operations of a specified length. In this test case, N goes from 500 to 10 000.

To evaluate `Integer` expressions, we generate `IntVar`s by repeating idempotent arithmetic operations. As explained in section 6.1.6, every arithmetic operation creates a new `IntVar`. As the starting value, we use the selection cardinality, and repeatedly add 15 to it and then subtract 15 from it N times. Finally, we post a constraint that the selection cardinality must be equal to the new value, so that the solver is forced to propagate the results of all the individual calculations. In this test case, N also goes from 500 to 10 000.

Every configuration of every test case runs 100 times.

Figure 7.1 shows timing results. Each point represents median time to solve a configuration of a given size. Standard deviation across runs was under 150 ms in most cases, so we choose not to display it in the graph.

Computation time for constraints grows linearly with the total number of constraints, and 750 000 constraints can be processed in about 5 seconds. This would match the intuition that the solver needs to evaluate each constraint once to ensure that the tested solution is valid.

An interesting point is the discontinuity around the 750 000 mark. This is most likely caused by the corresponding discontinuity in memory usage, as discussed below.

Computation time for integers also grows linearly, which matches the intuition that the sequence of arithmetic operations must be propagated in linear time. The results also closely follow the time measured for simple constraints. Each

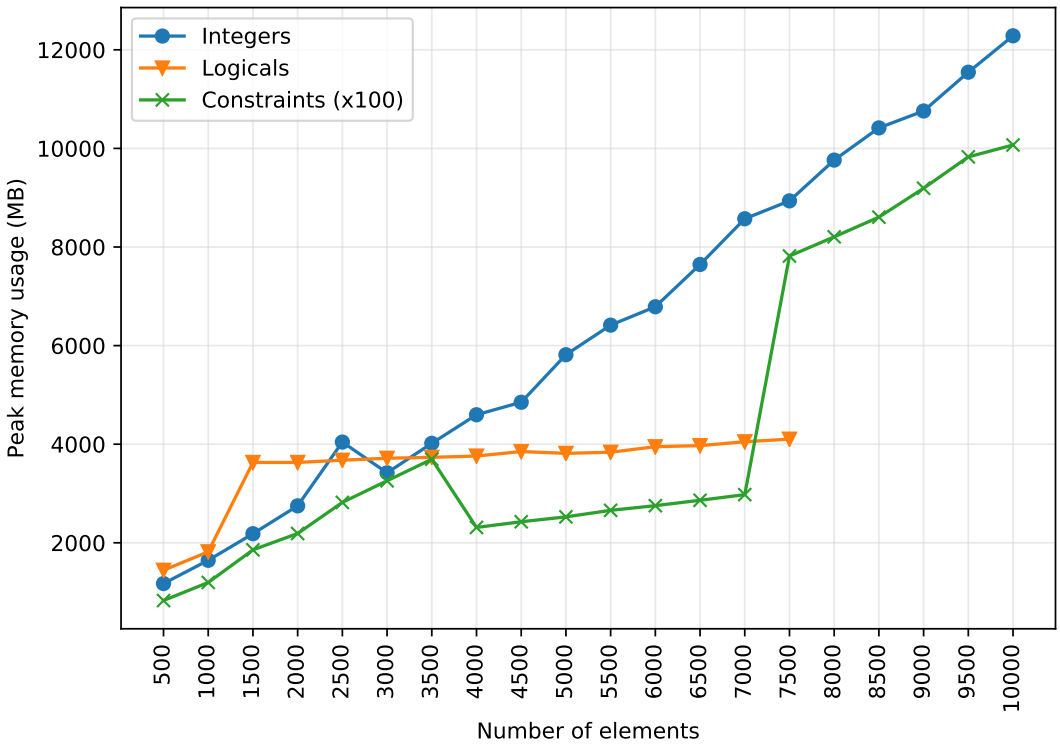

Figure 7.2: Memory consumption for basic building blocks

point represents two arithmetic operations per element, so we can conclude that each arithmetic operation takes roughly as much processing time as 50 simple constraints.

Computation time for logical operations grows quadratically and runs out of the allotted 30-second time limit at 8 000 operations. A `Logical` operation does not create new variables in the solver, and the full expression is posted as a single constraint. The quadratic behavior comes from an inefficiency in Choco's CNF normalization, which does not deal well with the unbalanced expression we are submitting. Still, 4 000 logical operations can be done under 5 seconds.

Figure 7.2 shows peak memory usages. Each point represents a maximum over the configuration runs. Memory usage data has an extremely high variance, sometimes on the order of hundreds of megabytes, which we chalk up to the unreliability of our measurements. Nevertheless, the maxima paint a clear picture.

Memory usage for constraints grows linearly, but shows a distinct discontinuity between 400 000 and 700 000 elements. As noted above, this corresponds to a discontinuity in processing time. We do not have an explanation for this behavior. However, the discontinuity starts before memory usage reaches 4 GB and returns to original projection at around 8 GB. We therefore hypothesize that this could be an artifact of JVM memory allocator behavior, or perhaps a custom allocator in Choco.

Memory usage for integers also grows linearly, in a much more predictable manner. We note that in this test case, the recorded peak usage resembles the actual usage more closely than in other tests. In previous experiments, when only 4 GB were allocated to the JVM, the integer test case ran out of memory

at around 7 500 elements, and computation time started to rise sharply around the 4 500 mark — presumably due to memory pressure and the need to run GC during the computation. The other test cases did not run into similar problems; GC runs were causing timing jitter, but other than that, the constraint test case continued linearly until 950 000 elements, and the logical operation test case was unaffected.

The logical operation test case reaches a plateau at the 1 500 mark, rising only very slowly in subsequent configurations. We expect that this corresponds to some pre-allocated internal array whose size is sufficient for all our configurations; given that the first jump is from 2 GB to 4 GB, it is possible that at some larger configuration the pre-allocated array would grow twice as large again. The `LogOp` tree and associated data is very small in comparison.

7.3 Evaluating the Running Example

The security policy from the running example contains both static and dynamic elements, and showcases all major features of the framework. It is therefore a good starting point for evaluation. We have designed a number of testing scenarios based on it.

To quickly summarize, the security policy manages a building with workrooms, which are statically assigned to projects, and lunchrooms, which are unassigned and limited by capacity. At lunch time, workers can request seats in lunchrooms and the policy should grant them access as soon as a seat becomes available, while upholding the overall security goal: workers from different projects must not use the same room.

There are two main types of sub-ensembles in the policy. `WorkroomAssignment` is registered for each project and allows workers of the project to enter all workrooms of that project. `LunchroomAssignment` is registered for each lunchroom and ensures that seats in that lunchroom are allocated as appropriate.

7.3.1 Static Assignments

The first scenario is in the morning, when workrooms are already open, but lunchrooms are closed. Per the situation predicate, `LunchroomAssignment`s are inactive, so only statically-assigning `WorkroomAssignment`s will be in play. Measuring their behavior should give us a performance baseline.

There are 100 workrooms that are assigned in a round-robin fashion to the configured number of projects. We have set up three test cases, for 5, 15, and 50 projects. In each case, we vary the number of workers from 500 to 10 000 in increments of 500. Each configuration is tested 500 times.

Figure 7.3 shows the results of measurements. Vertical lines from the points represent error bars of one standard deviation; in many cases, too small to be visible over the point marker.

Computation time grows linearly with the total number of access grants. For this reason, times for the 5-project test case are larger: the number of workrooms is fixed, so each project gets more rooms and each worker gets an access grant to each of the rooms.

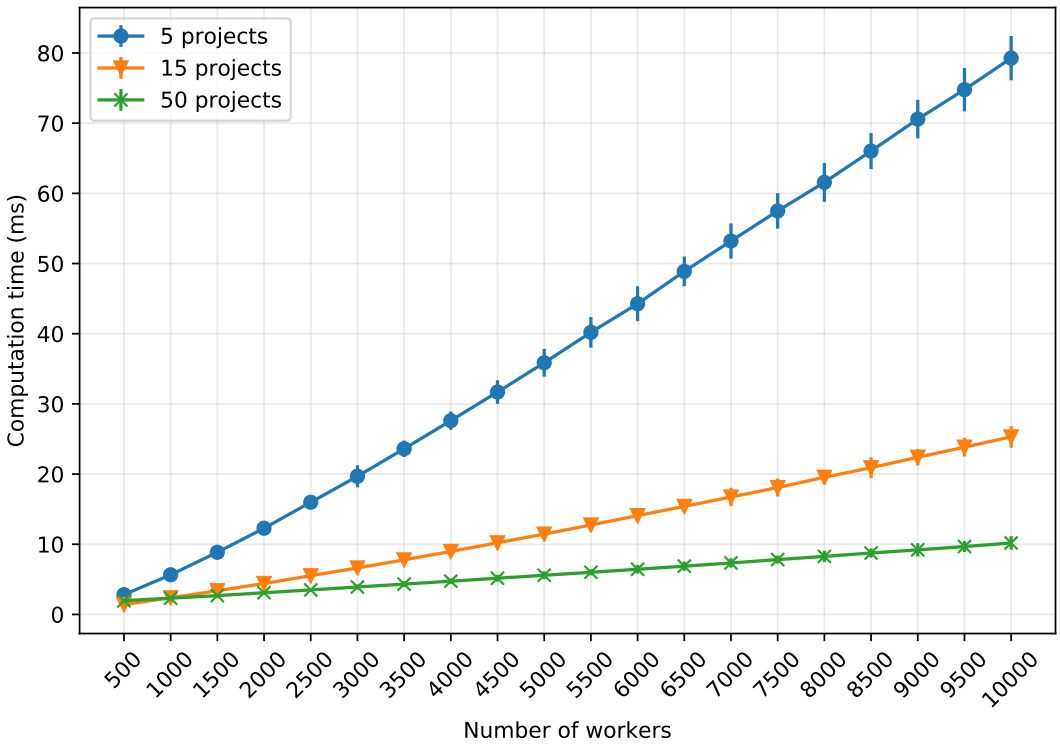

Figure 7.3: Static assignment to workrooms

The measurements are in the millisecond range, and it is possible to process 10 000 workers in 80 ms. Most of the time is actually spent pre-generating a lookup table for the `allows()` method; when this feature is disabled, computation times drop as much as 4x for the largest configuration.

7.3.2 Empty Lunchrooms

At lunch time, `LunchroomAssignment` ensembles are activated, and workers can request seats by setting their `hungry` attribute to true. In the second scenario, we assume that all available lunchrooms are empty, and the solver is attempting to seat an increasing number of hungry workers.

From the outset, it is clear that the problem has an exponentially large solution space. The solver is attempting to maximize a utility expression: $\sum |s_i|^2$, where s_i is the set of occupied seats in i-th lunchroom. Even with the knowledge that only the cardinality of seatings matters, the search still needs to implicitly consider all possible assignments of projects to lunchrooms, and all possible distributions of workers between the available rooms.

The test case in figure 7.4 has 4 lunchrooms with 10 seats each, and 3 available projects. Individual configurations then specify a total number of hungry workers, which are selected from the projects in a round-robin fashion.

The Y axis represents time to find an optimal solution. Each configuration is rendered as a box plot over the test runs, with whiskers representing 1.5 IQR, and outliers removed. The blue line is an exponential curve fitted to the medians.

Even in this very small test case, computation times are unacceptably slow. Seating 12 workers takes 5 seconds, and the solver crosses the 30 second time

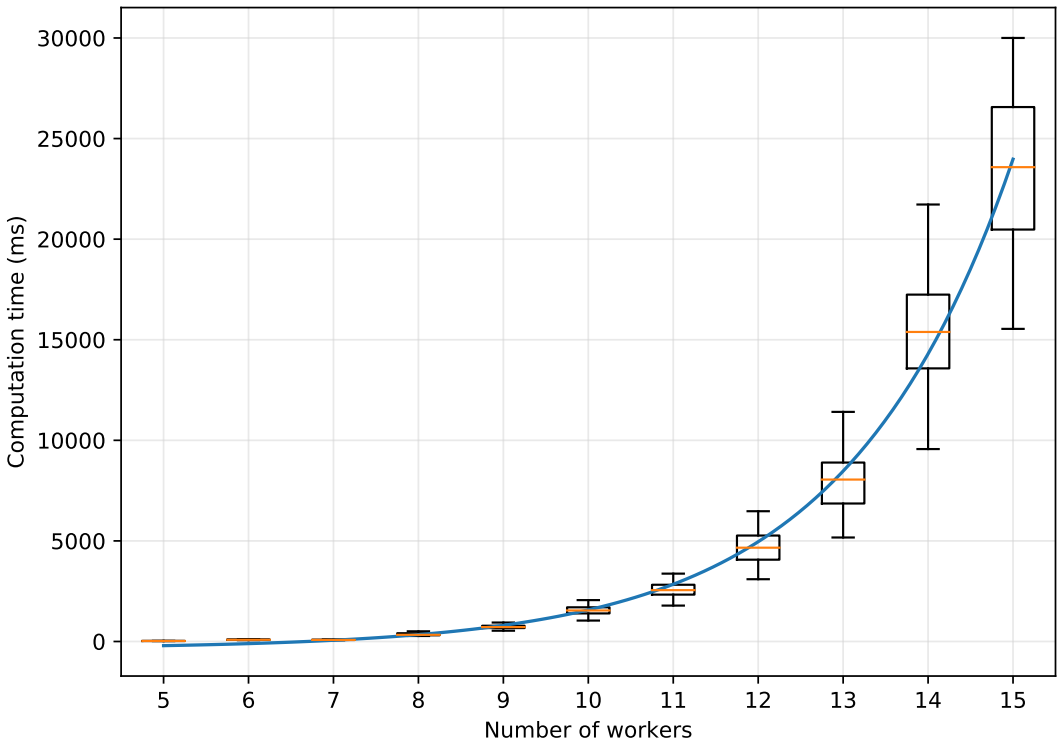

Figure 7.4: Dynamic assignment with more rooms than projects

limit at the 16 worker mark[1].

The situation with more available lunchrooms than projects represents the worst case for the solver: it is certain that at least one project will be able to use more than one room, so workers from that project can be distributed among the available rooms.

This factor is removed in the next scenario. Number of lunchrooms is fixed to 3, each with 5 seats, and number of projects varies from 5 to 9. Same as in previous scenario, the lunchrooms are empty and we are seating an increasing number of hungry workers.

The search is simplified by the fact that in terms of utility, distributing workers from the same project across rooms is always worse than putting them together and using workers from a different project for the other rooms. Although the number of solutions is still exponential, computation times are much lower.

Figure 7.5 shows the results. Variance in computation times is comparable to the previous scenario, so we are only showing median times to optimal solution. Two outlier configurations have been omitted: 6 projects with 35 workers, and 8 projects with 39 workers. See section 7.3.3 for discussion. After that, the longest time recorded is approx. 9 seconds, for 27 workers in 9 projects.

An interesting property is visible here. The peak times correspond to multiples of the number of projects, and after $4p + 1$ workers, computation times sharply drop to ~10 ms. This point corresponds to the moment when the round-robin algorithm first picks 5 hungry workers (same as lunchroom capacity) from the

[1]The solver was able to find the optimal solution under 30 seconds in several cases, but it timed out in most of the 100 test runs. For this reason, we have excluded this data entirely.

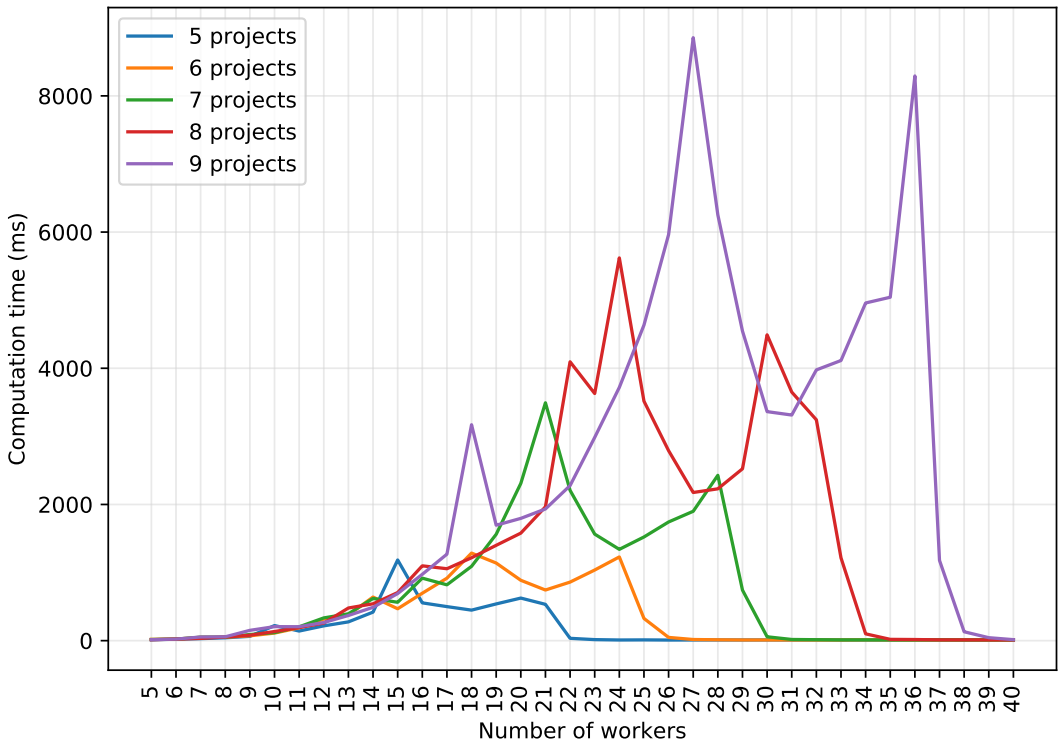

Figure 7.5: Dynamic assignment with more projects than rooms

same project. As soon as a lunchroom can be filled to capacity by a single project, the solver seems to be able to use this fact to optimize the search.

Similarly, multiples of number of projects are the points when the same number of workers is picked from every project. At that point, all projects are equally good candidates for seating, and so the number of solutions rises; after the peak, it is possible to quickly exclude the projects with fewer hungry workers.

Although the computation is still relatively slow, we are now within realistic problem sizes. In a real-world deployment, seating 30 workers in 5 seconds might be acceptable.

7.3.3 Probabilistic Search Anomalies

In the previous scenario, we observed that when a trivial solution of a certain type exists, the solver converges on that solution very quickly. In particular, when it is possible to fill a room to capacity, an optimal solution is found in milliseconds. This result was stable across runs and not affected by the number of "overflow" workers who are hungry but could not be seated.

However, in some cases, the optimization seems to fail. This often appears in the form of outliers — in a particular configuration, most attempts take 20 ms, but some can take several seconds. For some configurations, the short times are actually the outliers: most attempts take several seconds, and the occasional "good try" completes immediately.

We have not found a pattern in the bad configurations. The outliers in good configurations behave probabilistically and appear more frequently as configuration size grows.

To measure this behavior, we have set up the following scenario: the number of projects is the same as the number of lunchrooms, each lunchroom has a capacity of 5, and there are 5 hungry workers per project. The trivial solution is apparent: each lunchroom is filled to capacity with workers from one project, and the remaining search space is only as big as the number of project-lunchroom permutations.

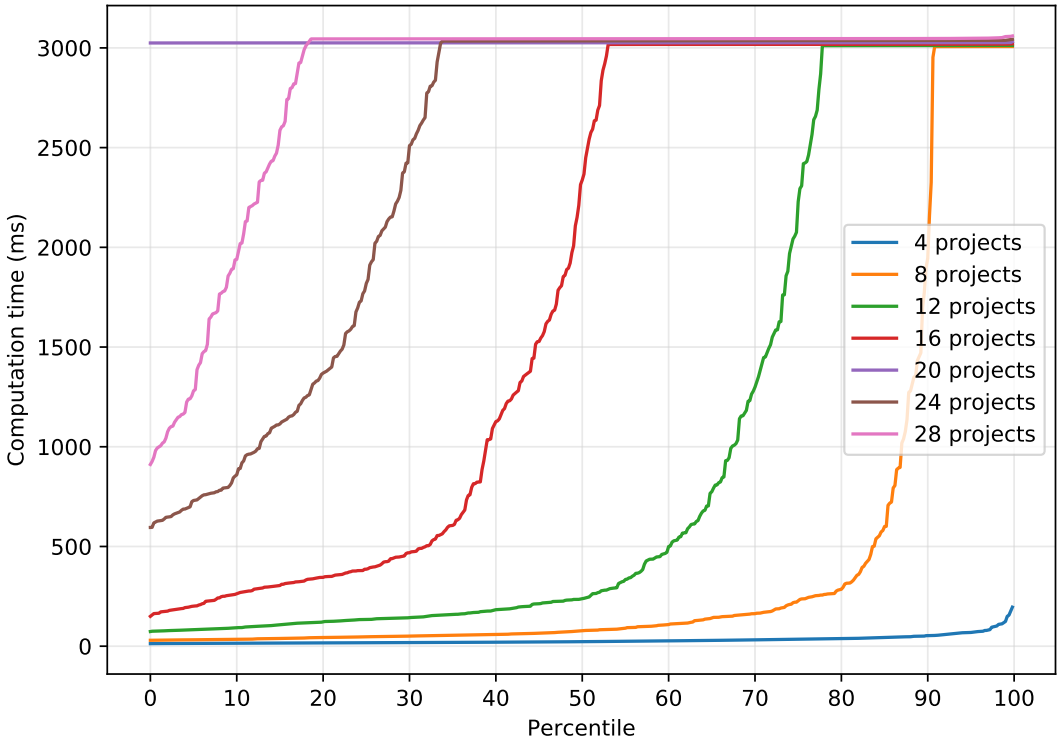

Figure 7.6: Frequency of search anomalies

We created configurations of 1 to 30 projects, and ran 500 tests of each, with a time limit of 3 seconds. The expected solving time of one run is below 100 ms, so this time limit was deemed sufficient. Results are shown as a percentile graph in figure 7.6. We arbitrarily chose to plot multiples of 4. Showing more configurations does not bring any additional insights and makes the graph less clear.

For 4 projects, all tests finished under 500 ms. At 12 projects, 60 % of tests finished under 500 ms, and 25 % of tests timed out. 28 projects is large enough that no test finished under 500 ms, but the curve still has a similar shape; presumably, the likelihood of finding the good solution is too low.

The configuration with 20 projects is a "bad" one, as all of the 500 attempts have timed out.

7.3.4 Solver Time Limits

In all previous scenarios, we measured time to find an *optimal* solution. When the search timed out before declaring a solution optimal, it was counted as a failure. However, that does not mean that *no* solutions were found.

Choco's optimization process works by finding a satisfying solution for the problem and then posting an additional constraint that the next solution must have a strictly higher utility. Most of the time this means iterating over several successively better solutions. If an optimal solution is not required, it is possible to either switch the behavior of the solver, or to accept the best solution found when the time limit is reached.

To test the trade-off between speed and solution quality, a scenario was set up with 3 projects, 5 lunchrooms with 20 seats each, and 21 hungry workers, 7 per project. In this scenario, the optimal solution is seating all of the workers from a single project in one of the available lunchrooms, and keeping two rooms empty. The total utility of this solution is $7^2 \cdot 3 + 0 \cdot 2 = 147$.

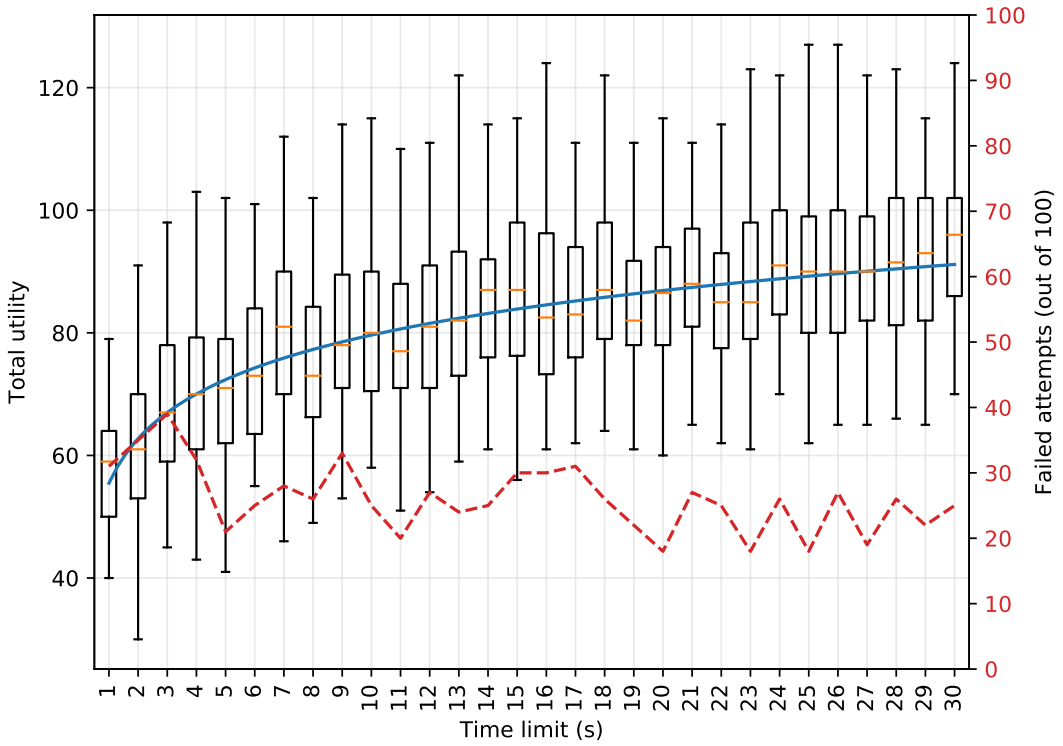

Figure 7.7: Utility of solutions found before time limit

We created configurations with a successively higher time limit from 1 to 30 seconds, and measured them over 100 runs. Figure 7.7 shows each configuration as a box plot over successful tests, with whiskers representing 1.5 IQR, and outliers removed. A relatively large amount of tests have failed to find a solution within the time limit. The red dashed line shows the amount for each configuration. Over 50 attempts were successful in every case, which gives us a sufficient number of samples.

The data has high variance, but the successful runs copy a logarithmic curve (shown in blue). That is the expected behavior. On an exponential problem, linear increase in computation time should provide logarithmic benefits.

None of the test runs get close to the optimal utility of 147. In preliminary experiments, even tests with a time limit of 5 minutes did not converge on the optimal solution. These results also retroactively show that our choice of 30 seconds as the default time limit was reasonable.

7.3.5 Practical Situations

Previous scenarios were testing the solver on arbitrary configurations in isolation. In a real-world deployment, however, the situation would look very different.

First thing to note is that in practice, it is almost impossible for dozens of workers to request lunch at exactly the same time. We have experimented with a "one-by-one" solving method, where instead of submitting all hungry workers to the solver at once, we incrementally submit one worker at a time. This converts the $O(k^N)$ problem of seating N workers to N separate $O(1)$ problems of seating one worker. In most cases, this approach will achieve the same utility, because workers from the same project are preferentially seated together. Computation time grows linearly with the number of workers.

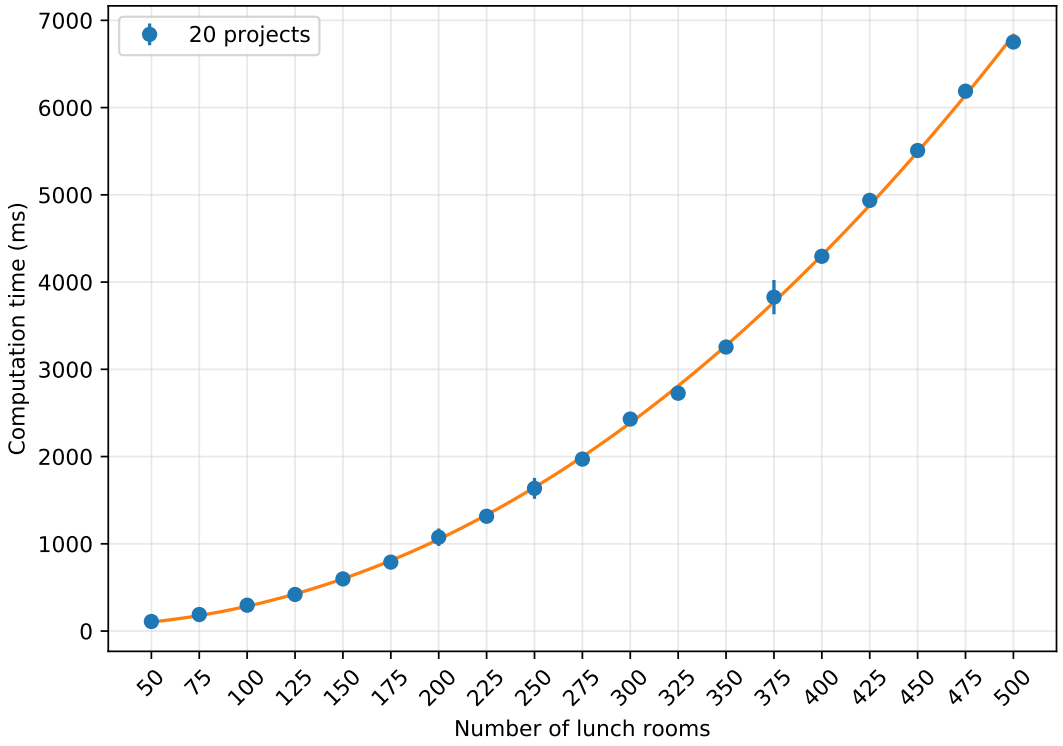

Figure 7.8: Time to seating one worker

Figure 7.8 shows computation times for seating one worker in an increasing number of lunchrooms. Each case was measured 100 times. Blue dots represent median times, vertical lines are error bars of one standard deviation.

The number of projects is irrelevant, as the worker being seated can only belong to one, and lunchrooms are only tied to projects through their occupants. Lunchroom capacity is also irrelevant, as long as it is 1 or more. At 200 lunchrooms, one worker can be seated in one second, which is probably good enough for any reasonable real-world deployment.

Solving time is perfectly quadratic (shown in orange) in the number of lunchrooms. The source of the quadratic behavior is most likely the `allDisjoint` constraint: a worker can be seated in any of the N rooms, and for each variant, N rooms must be checked to ensure disjointness.

7.3.6 Simulation

In practical situations, it is also very rare for all lunchrooms to be empty at the same time. When the first worker is seated in a lunchroom, the choice locks the worker's project to that room. All subsequent requesters from the same project will be preferentially seated there. This effectively splits the problem in two: workers from a chosen project will only be seated to that project's rooms, while these are excluded from the possibilities for workers of other projects.

In order to examine this behavior, and to test the solver in more real-world-like conditions, we have created a simulation of worker behavior at lunch time. The basis of the simulation is a building with 500 workers working on 10 projects, which is realistic for a modern office building. On each of the 5 floors there is a lunchroom with 40 seats.

At every step of the simulation, every worker that hasn't eaten yet has a 0,5 % chance of becoming hungry and requesting a seat. Once a worker receives a seating notification, they take up to 5 simulation steps to reach the lunchroom, and then 5 to 20 steps to eat, before leaving the lunchroom and getting back to work.

In a typical simulation run, all workers can have lunch in about 2 000 steps. We are not particularly interested in the tail end of the simulation, because it mostly involves waiting for workers to finish eating. Therefore, we stop the simulation after 1 500 steps, and run the first 1 500 steps 100 times, for a total of 150 000 steps.

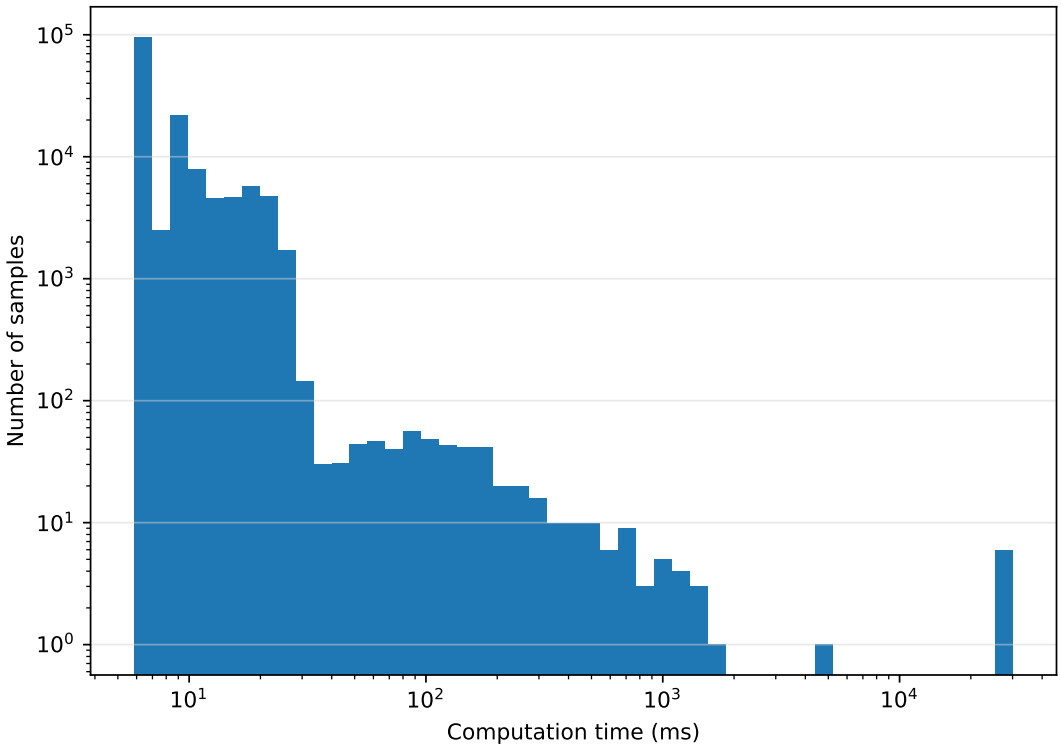

Figure 7.9: Histogram of computation times per simulation step

Figure 7.9 is a histogram of computation times per one simulation step, with both axes logarithmic. The median step time is 6.4 ms, and mean step time is

10.7 ms. Out of the 150 000 measurements, only 285 took more than 100 ms.

There is a small number of outliers that do not find the optimal solution within the 30 second time limit. However, an important feature of this kind of setup is that it can recover very fast from such failure. In the rare case where a difficult configuration arises, such as when two lunchrooms become available in the same simulation step, a sub-optimal seating solution will still place some workers in at least one of the rooms. Subsequent solver runs can continue off this result and place the remaining workers. We have never observed more than one failure in a row.

From these results, we conclude that the performance of the framework is suitable for real-world deployment.

8. Conclusion

In this work we have presented an ensemble-based approach to defining access control policies. Security situations are specified in terms of ensembles and their roles, and access control decisions are attached to the ensembles. This allows the policy to follow the evolution of a dynamic system as its shape and composition changes over time.

We introduced a framework, TCOOF-Trust, which consists of two parts: a policy specification language, implemented as an internal DSL in Scala, and a runtime environment for resolving the policies. The DSL can be used to specify ensembles, roles, constraints on their membership, and attached security decisions. The runtime environment processes the policy specification in the DSL, converts the problem to an input for its internal CSP solver, and uses the results to determine ensemble membership. Based on the solution and the rules attached to it, the framework can respond to access control queries.

We have designed and specified detailed semantics for the behavior of the framework, and extended the existing TCOOF-Trust prototype implementation to properly support them. A clean API for embedding the framework in external projects is provided, and detailed documentation was created for the CSP conversion process and for the implementation details of the framework.

An example security scenario with dynamic access control requirements was used to evaluate performance of the framework, both in synthetic and real-world-like configurations. While the underlying problem is exponential by nature, results show that the framework is capable of dealing with the complexity in a reasonable manner, and scales well in practical deployments.

Bibliography

[1] Rima Al Ali, Tomáš Bureš, Petr Hnětynka, Filip Krijt, František Plášil, and Jiří Vinárek. *Dynamic Security Specification Through Autonomic Component Ensembles*. In Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part III, pages 172–185, 2018. doi: 10.1007/978-3-030-03424-5_12. URL https://doi.org/10.1007/978-3-030-03424-5_12.

[2] T. Bures, F. Plasil, M. Kit, P. Tuma, and N. Hoch. *Software Abstractions for Component Interaction in the Internet of Things*. Computer, 49(12):50–59, 2016. ISSN 0018-9162. doi: 10.1109/MC.2016.377.

[3] Tomáš Bureš, Ilias Gerostathopoulos, Petr Hnětynka, František Plášil, Filip Krijt, and Jiří Vinárek. *Trait-based Language for Smart Cyber-Physical Systems*. Technical Report D3S-TR-2017-01, Department of Distributed and Dependable Systems, Charles University, 2017.

[4] Rocco De Nicola, Gian Ferrari, Michele Loreti, and Rosario Pugliese. *A Language-Based Approach to Autonomic Computing*. 01 2013. doi: 10.1007/978-3-642-35887-6_2.

[5] SciPy developers. *SciPy*. URL https://www.scipy.org/. Accessed 2019-07-15.

[6] Matplotlib development team. *Matplotlib*. URL https://matplotlib.org/. Accessed 2019-07-15.

[7] Espertech. *Esper*. URL http://www.espertech.com/esper/. Accessed 2019-06-20.

[8] David Ferraiolo, Janet Cugini, and D Richard Kuhn. *Role-based access control (RBAC): Features and motivations*. In Proceedings of 11th annual computer security application conference, pages 241–48, 1995.

[9] Rolf Hennicker and Annabelle Klarl. *Foundations for Ensemble Modeling - The HELENA Approach*. In Specification, Algebra, and Software, volume 8373 of *Lecture Notes in Computer Science*, pages 359–381. Springer, 2014. URL http://pmi.pst.ifi.lmu.de/text_files/0000/0012/sas14_final.pdf.

[10] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication, 800:162, 2014. URL https://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.SP.800-162.pdf. Accessed 2019-06-20.

[11] Anas Abou El Kalam, R EI Baida, Philippe Balbiani, Salem Benferhat, Frédéric Cuppens, Yves Deswarte, Alexandre Miege, Claire Saurel, and Gilles Trouessin. *Organization based access control*. In Proceedings POLICY 2003.

IEEE 4th International Workshop on Policies for Distributed Systems and Networks, pages 120–131. IEEE, 2003.

[12] Jeffrey O. Kephart and David M. Chess. *The Vision of Autonomic Computing*. Computer, 36(1):41–50, January 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055. URL http://dx.doi.org/10.1109/MC.2003.1160055.

[13] Annabelle Klarl. *HELENA - Handling massively distributed systems with ELaborate ENsemble Architectures*. PhD thesis, lmu, 2016.

[14] Filip Krijt, Zbynek Jiráček, Tomáš Bureš, Petr Hnětynka, and František Plášil. *Automated Dynamic Formation of Component Ensembles - Taking Advantage of Component Cooperation Locality*. In Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017., pages 561–568, 2017. doi: 10.5220/0006273705610568. URL https://doi.org/10.5220/0006273705610568.

[15] Devdatta Kulkarni and Anand Tripathi. *Context-aware role-based access control in pervasive computing systems*. In Proceedings of the 13th ACM symposium on Access control models and technologies, pages 113–122. ACM, 2008.

[16] Arun Kumar, Neeran Karnik, and Girish Chafle. *Context sensitivity in role-based access control*. ACM SIGOPS Operating Systems Review, 36(3):53–66, 2002.

[17] R. Laborde, A. Oglaza, F. Barrère, and A. Benzekri. *dynSMAUG: A dynamic security management framework driven by situations*. In Proceedings of CSNet 2017, Rio de Janeiro, Brazil, pages 1–8. IEEE, October 2017. doi: 10.1109/CSNET.2017.8241987.

[18] Rocco De Nicola. *A formal approach to autonomic systems programming: the SCEL language*. In ICTCS, 2014.

[19] OASIS. *eXtensible Access Control Markup Language (XACML) Version 3.0*. Technical report, 2013. URL http://docs.oasisopen.org/xacml/3.0/xacml-3.0-core-spec-en.pdf. Accessed 2019-06-20.

[20] OASIS. *Abbreviated Language for Authorization Version 1.0*. Technical report, 2015. URL https://www.oasis-open.org/committees/download.php/55228/alfa-for-xacml-v1.0-wd01.doc. Accessed 2019-06-20.

[21] Oracle. *Java Management Extensions (JMX)*, 2018. URL https://docs.oracle.com/javase/6/docs/technotes/guides/jmx/index.html. Accessed 2019-07-10.

[22] pandas. *Python Data Analysis Library*. URL https://pandas.pydata.org/. Accessed 2019-07-15.

[23] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017. URL http://www.choco-solver.org. Accessed 2019-06-20.

[24] M. Mohsin Saleemi, Natalia Díaz Rodríguez, Johan Lilius, and Iván Porres. *A Framework for Context-Aware Applications for Smart Spaces*. In Proceedings of NEW2AN 2011 and ruSMART 2011, St. Petersburg, Russia, volume 6869 of *LNCS*, pages 14–25. Springer, August 2011. doi: 10.1007/978-3-642-22875-9_2.

[25] Mazeiar Salehie, Liliana Pasquale, Inah Omoronyia, Raian Ali, and Bashar Nuseibeh. *Requirements-driven adaptive security: Protecting variable assets at runtime*. In 2012 20th IEEE International Requirements Engineering Conference (RE), pages 111–120. IEEE, 2012.

[26] Mark Strembeck and Gustaf Neumann. *An integrated approach to engineer and enforce context constraints in RBAC environments*. ACM Transactions on Information and System Security (TISSEC), 7(3):392–427, 2004.

[27] M. H. Valipour, B. Amirzafari, K. N. Maleki, and N. Daneshpour. *A brief survey of software architecture concepts and service oriented architecture*. In 2009 2nd IEEE International Conference on Computer Science and Information Technology, pages 34–38, Aug 2009. doi: 10.1109/ICCSIT.2009.5235004.

[28] École Polytechnique Fédérale Lausanne. *The Scala Programming Language*. URL https://www.scala-lang.org/. Accessed 2019-06-23.

A. Data Archive Contents

The data archive accompanying this work is a copy of the `tcoof-trust` GitHub repository. Its up-to-date version can always be found at the following URL: https://github.com/matejcik/tcoof-trust

The code uses the Scala Build Tool (SBT) for building, running, and generating documentation. Please refer to `README.md` for details.

To run the example code, use `sbt run`.

To build the API documentation in HTML format, use `sbt doc`. The resulting documentation will be stored in `target/scala-2.12/api`.

To run the unit test suite, use `sbt test`.

The following is a list of paths in the archive and descriptions of their contents.

`README.md`
: Markdown-formatted `README` for the Git repository. Please refer to this file for requirements to run the TCOOF-Trust framework and basic usage examples.

`build.sbt`
`project/`
: SBT build script and build properties. These files are required for building the project with `sbt`, and for importing into IntelliJ IDEA, the preferred Scala IDE.

`python/all.sh`
: Shell script that installs the Python environment and regenerates graphs used in this work.

`python/Pipfile`
`python/Pipfile.lock`
: `pipenv` configuration files. Contain list of Python package dependencies required for generating graphs.

`python/lunch.py`
`python/other.py`
`python/resultlib.py`
`python/variables.py`
: Individual Python scripts that generate the graphs. `variables.py` generates graphs from section 7.2, `other.py` generates the simulation histogram and the timeout graph, `lunch.py` generates all the rest. `resultlib` is a small library of common functions.

`results/final/badsolver-growingprojects.log`
: Source data for figure 7.6.

`results/final/booleans.log`
`results/final/constraints.log`
`results/final/integers.log`
: Source data for figures 7.1 and 7.2

`results/final/moreprojects.log`
: Source data for figure 7.5.

`results/final/morerooms-optimizing.log`
: Source data for figure 7.4.

`results/final/oneworker-params.log`
: Source data for figure 7.8.

`results/final/simulated.log`
: Source data for figure 7.9.

`results/final/timelimits.log`
: Source data for figure 7.7.

`results/final/workercount-simple.log`
: Source data for figure 7.3.

`src/main/scala/cz/`
: Scala source files for the TCOOF-Trust framework, specifically the package `cz.cuni.mff.d3s.trust`.

`src/main/scala/scenario/`
: Scala source files for the test measurement scenarios.

`src/test/scala/`
: Scala source files for the unit test suite.