Using Component Ensembles for Modeling Autonomic Component Collaboration in Smart Farming

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

Tomas Bures
Charles University
Prague, Czech Republic
bures@d3s.mff.cuni.cz

Ilias Gerostathopoulos
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
i.g.gerostathopoulos@vu.nl

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

ABSTRACT

Smart systems have become key solutions for many application areas including autonomous farming. The trend we can see now in the smart systems is that they shift from single isolated autonomic and self-adaptive components to larger ecosystems of heavily cooperating components. This increases the reliability and often the cost-effectiveness of the system by replacing one big costly device with a number of smaller and cheaper ones. In this paper, we demonstrate the effect of synergistic collaboration among autonomic components in the domain of smart farming—in particular, the use-case we employ in the demonstration stems from the AFarCloud EU project. We exploit the concept of autonomic component ensembles to describe situation-dependent collaboration groups (so called ensembles). The paper shows how the autonomic component ensembles can easily capture complex collaboration rules and how they can include both controllable autonomic components (i.e. drones) and non-controllable environment agents (flocks of birds in our case). As part of the demonstration, we provide an open-source implementation that covers both the specification of the autonomic components and ensembles of the use case, and the discrete event simulation and real-time visualization of the use case. We believe this is useful not only to demonstrate the effectiveness of architectures of collaborative autonomic components for dealing with real-life tasks, but also to build further experiments in the domain.

CCS CONCEPTS

• **Computer systems organization** → **Self-organizing autonomic computing**; • **Software and its engineering** → **Domain specific languages**; • **Applied computing** → *Agriculture*.

KEYWORDS

autonomic systems, self-adaptive architecture, smart farming, dynamic adaptation

1 INTRODUCTION

Smart systems can nowadays be found in almost any application domain. They have become the key technologies in achieving higher efficiency and lowering costs. One such an application, where smart systems offer great impact, is smart farming. Examples include agriculture vehicles like tractors and harvesting machines that, while at a field, continuously monitor their status (telemetry data, etc.) and upload it to a cloud services, where these data are analyzed and potential issues discovered. Another example is during the harvest, when these machines continuously measure the amount of crop and connect this with planning on how much fertilizer is to be applied on specific segments of the field for the next year. Yet another example, which we focus on in this paper, is the use of small and relatively inexpensive UAVs (drones) to monitor fields.

The important characteristic of these smart systems is the ability to perform relatively coarse-grained tasks autonomously – typically by means of self-adaptation. Another important and recent trend in these systems is that there is a shift from single isolated autonomic and self-adaptive components to larger ecosystems of components that heavily cooperate together to achieve a given common goal. This increases the reliability and often the cost-effectiveness of the whole system by substituting a big costly device by a number of smaller and cheaper ones.

The increased level of collaboration among components (e.g. drones monitoring fields) brings however issues in how to model and specify such collaboration at an architectural level. This is even more a complex task as the collaboration is invariably situation-dependent – e.g. in the smart-farming domain, the situation comprises things such as the position and energy level of all drones, vehicles, etc., data from their cameras and environment sensors, but also inference of the state of environment entities, which cannot be directly controlled but only observed, like fields, animals, etc.

In this paper, we describe an approach of autonomically composable and context-dependent rules for describing cooperation among

autonomic components a system is composed of. The approach is based on our ongoing work on autonomic component ensembles [16], which are situation-dependent collaboration groups. In particular, we use the Trait-based COmponent Ensemble Language (TCOEL) and the Trait-based COmponent Ensemble Framework (TCOEF), which allow for specifying ensembles in more complex hierarchical situations. Our approach allows for describing cooperation between both directly controllable and non-controllable components.

We demonstrate the approach on a specific smart farming scenario. The aim here is to provide a more comprehensive case that is accompanied with an an open-source implementation that covers both the specification of the autonomic components and ensembles, and also the discrete event simulation and real-time visualization. The overall value to the community lies in having a concrete showcase that can be used, explored and experimented with to gain in-depth understanding on the intricacies of systems of collaborating components and their specification using the concept of autonomic component ensembles.

Though simplified, the used example is also not a completely synthetic one but it is based on an actual cooperation with industrial and agricultural partners within the ongoing international project AFarCloud (ECSEL EU project).

The paper is structured as follows. In Section 2, the running example is described. Section 3 presents our approach to cooperation specification through autonomic component ensembles, while Section 4 provides details about experimentation with the implementation. In Section 5, related work is discussed and Section 6 concludes the paper.

2 RUNNING EXAMPLE

To illustrate our approach, we use the following example in the rest of the paper. The example is an actual scenario taken from our ECSEL JU project AFarCloud[1], which focuses on holistic and systematic integration of cyber-physical systems and cloud-based systems in farming to increase agriculture efficiency, productivity, animal health, and food quality and reduce farm labor costs. In the example, we focus on coordination of cyber-physical devices used for crop management.

The particular situation is depicted in Figure 1 (the figure is an actual screenshot from our simulator we developed to demonstrate the scenario). A farm area in the figure consists of several fields. The yellow ones still contain crop, while the brown ones have been already harvested and are empty. The darker green ones contain forage crops.

The whole farm area is continuously monitored by a fleet of autonomous drones. In addition to environment monitoring (moisture, etc.) and security monitoring (unauthorized access to fields, etc.), the drones also monitor for occurrence of flocks of birds, which can quickly make a serious damage to the crops. In a case a flock of birds is discovered over the crop fields, the drones are used to scare the birds and drive them away from the farm area or, at least, to already harvested fields or fields with crop "insensitive" to birds (e.g., forage crop), where they cannot do damage.
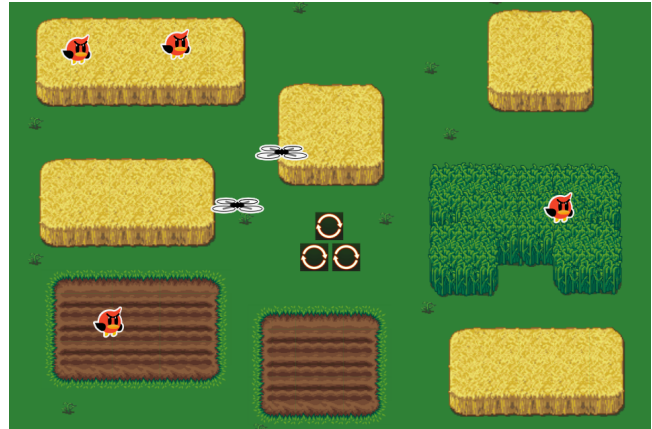
Figure 1: Running example.

For the system to be effective, sufficient amount of drones needs to be assigned for the particular task (monitoring, scaring of birds). Plus, the drones have only a limited amount of energy and need to recharge their batteries at the charger (the rounded arrows icons in the center). The charger can charge only a limited number (three actually) of the drones at the same time.

If a patrolling drone detects a flock of birds, based on a size of the flock, a sufficient number of other drones should be alarmed and redirected to the particular field. As the primary selection criterion, the distance of the drone to the flock is used. After the flock threat is removed, the drones returns to their previous activity.

To sum up, the system has to concurrently ensure several conditions: (i) a sufficient number of drones has to be assigned for a task (patrolling the area, scaring the birds), (ii) the best drones have to be assigned (the closest ones, with a sufficient amount of energy), (iii) drones with low level of energy have to be recharged.

3 DESCRIPTION OF AUTONOMOUS OPERATIONS

To describe and run autonomous systems like the running example, we employ our approach of autonomic component ensembles [16]. Via this approach, entities of a system (drones, chargers, but also persons, animals, etc.) are represented as *components* and cooperation among entities is modeled via *ensembles*, which are context-dependent groups of components. Ensembles are dynamic – typically bound to time and space.

In particular, components and ensembles are defined as follows (a complete formal semantics of components and ensembles is available in [4]). A component represents an autonomic entity, which has a state (called component's *knowledge*). Components periodically perform actions in which they can sense their context (environment), perform computation and actuation, and update their knowledge.

An *ensemble* is a group of components that is formed in order to perform a required group-level activity (e.g. moving drones in a formation). Importantly, the members of the ensemble are not prescribed statically but they are determined dynamically at runtime via evaluation of the *membership condition*. The condition is a predicate over component types and knowledge. An ensemble

```scala
1  case class DroneComponent(
2    id: String, mode: DroneMode.DroneMode, position: Position, energy: Double,
3    chargingInChargerId: Option[ChargerId], observedFields: Map[String, ObservedFieldId]
4  ) extends Component {
5    name(id) /* ... */
6  }
7  case class FieldComponent(idx: Int, fieldObservations: List[FieldObservation]) extends
        Component {
8    name(s"Field ${idx}") /* ... */
9  }
10 case class ChargerComponent(idx: Int, isFree: Boolean) extends Component {
11   name(s"Charger ${idx}")
12   val chargerId = ChargerId(idx)
13   val position = chargerId.position
14   /*...*/
15 }
16 case class FlockComponent(position: Position) extends Component {
17   name(s"Flock ${position}") /* ... */
18 }
19
20 class DroneProtectionSystem extends Ensemble {
21   name(s"Root ensemble of the protection system")
22   val operationalDrones = allDrones.filter(drone => drone.mode != DroneMode.DEAD &&
           drone.mode != DroneMode.CHARGING && drone.energy >
           Drone.chargingThreshold)
23   val dronesInNeedOfCharging = allDrones.filter(drone => drone.mode !=
           DroneMode.DEAD && drone.mode != DroneMode.CHARGING && drone.energy
           < Drone.chargingThreshold)
24   val fieldsWithUnknownStatus = allFields.filter(_.isUnknown)
25   val fieldsUnderThreat = allFields.filter(_.isUnderThreat)
26   val freeChargers = allChargers.filter(_.isFree)
27
28   class ApproachFieldUnderThreat(field: FieldComponent) extends Ensemble {
29     name(s"ApproachFieldUnderThreat ensemble for field ${field.idx}")
30     val flocksInField = allFlocks.filter(x => field.area.contains(x.position))
31     val dronesInField = operationalDrones.filter(x => field.area.contains(x.position))
32     val droneCount = field.requiredDroneCountForProtection
33     val center = field.center
34     val drones = subsetOfComponents(operationalDrones, _ <= droneCount)
35
36     situation { dronesInField.size < droneCount }
37     utility {
38       drones.sum(x => if (field.area.contains(x.position)) 10 else dist2Utility(x.position,
             center))
39     }
40     tasks {
41       if (flocksInField.isEmpty) {
42         for (drone <- drones.selectedMembers) moveTask(drone, center)
43       } else {
44         val selectedDronesInFieldCount = drones.selectedMembers.count(x =>
               field.area.contains(x.position))
45         val flockPos = flocksInField.head.position
46         val step = Flock.disturbRadius * 2
47         var x = flockPos.x - (selectedDronesInFieldCount - 1) * Flock.disturbRadius
48
49         for (drone <- drones.selectedMembers) {
50           if (field.area.contains(drone.position)) {
51             moveTask(drone, Position(x, flockPos.y))
52             x += step
53           } else {
54             moveTask(drone, center)
55           }
56         }
57       }
58     }
59   }
60
61   class ScareFormation(field: FieldComponent) extends Ensemble {
62     name(s"ScareFormation ensemble for field ${field.idx}")
63     val dronesInField = operationalDrones.filter(x => field.area.contains(x.position))
64     val droneCount = field.requiredDroneCountForProtection
65     val segmentCenters = field.protectionCenters
66
67     situation { dronesInField.size >= droneCount }
68
69     class SegmentAssignment(segmentCenter: Position) extends Ensemble {
70       name(s"SegmentProtectionAssignment for field ${field.idx} @
             ${segmentCenter.x},${segmentCenter.y}")
71       val drone = oneOf(operationalDrones)
72
73       utility { drone.sum(x => dist2Utility(x.position, segmentCenter)) }
74       tasks { moveTask(drone, segmentCenter) }
75     }
76
77     val protectionSegmentAssignments = rules(segmentCenters.map(new
             SegmentAssignment(_)))
78     val drones = unionOf(protectionSegmentAssignments.map(_.drone))
79
80     utility {
81       protectionSegmentAssignments.sum(assignment => assignment.utility) / droneCount
82     }
83     constraint( protectionSegmentAssignments.map(_.drone).allDisjoint )
84   }
85
86   class PatrolUnknown(field: FieldComponent) extends Ensemble {
87     name(s"PatrolUnknown ensemble for field ${field.idx}")
88     val drone = oneOf(operationalDrones)
89     val fieldCenter = field.center
90
91     utility { drone.sum(x => dist2Utility(x.position, fieldCenter)) }
92     tasks { moveTask(drone, fieldCenter) }
93   }
94
95   class ChargerAssignment(charger: ChargerComponent) extends Ensemble {
96     name(s"ChargerAssignment for charger ${charger.idx}")
97     val drone = oneOf(dronesInNeedOfCharging)
98
99     utility { drone.sum(drone => dist2Utility(drone.position, charger.position) +
             (Drone.chargingThreshold - drone.energy).round.toInt)
100    }
101    tasks { chargeTask(drone, charger.chargerId) }
102  }
103  val patrolUnknown = ensembles(fieldsWithUnknownStatus.map(new PatrolUnknown(_)))
104  val chargerAssignments = ensembles(freeChargers.map(new ChargerAssignment(_)))
105  val approachFieldsUnderThreat =
           ensembles(fieldsUnderThreat.filter(ApproachFieldUnderThreat.map(new
           ApproachFieldUnderThreat(_)))
106  val scareFormations = ensembles(fieldsUnderThreat.map(new ScareFormation(_)))
107
108  utility {
109    approachFieldsUnderThreat.sum(assignment => assignment.utility) +
110    scareFormations.sum(assignment => assignment.utility) +
111    patrolUnknown.sum(assignment => assignment.utility) / 4 +
112    chargerAssignments.sum(assignment => assignment.utility)
113  }
114  constraint(
115    (patrolUnknown.map(_.drone) ++ approachFieldsUnderThreat.map(_.drones) ++
             scareFormations.map(_.drones)).allDisjoint &&
116    chargerAssignments.map(_.drone).allDisjoint )
117 }
```

Figure 2: Running example in DSL.

can be nested in another ensemble. The semantics of the nesting is that members of the ensemble must be also members of the parent ensemble (in which it is nested). This way, the top-level ensemble (which is not nested in another one) defines the overall goal of the system while nested sub-ensembles represent individual sub-goals.

In the rest of this section, we model the example shown in Section 2 in ensembles and provide details about the implementation. For simple and easy development and experimentation with ensembles, we have created a Scala-based internal domain-specific language (DSL) to specify components and ensembles called TCOEL [16]. Creating the DSL as an internal one allowed for its rapid development (there is no need to create a full toolchain for it), however the small downside is that its users need to have at least partial knowledge of the Scala language. On the other side, the required knowledge is rather minimal and basic familiarity with modern program language concepts (classes, etc.) is sufficient. Specifications in the DSL are at runtime processed by a CSP solver (the Choco solver [10] in particular), which resolves ensembles, i.e., determines which ensemble instances need to be created and which components need to be assigned to particular ensemble instances.

Figure 2 shows an excerpt of the specification in our DSL that models the farm example (its full version is available in the demonstration implementation).

In our DSL, both the types of components and ensembles are modeled via classes. The actual components and ensembles are then instances of these classes (every component and ensemble can be instantiated multiple times). In the example, there are four component types: DroneComponent, ChargerComponent, FieldComponent and FlockComponent (lines 1–18). The component knowledge is modeled via the class fields. FieldComponent and FlockComponent are non-controllable components, i.e., they cannot be controlled and their state is only observed. The FieldComponent is statically instantiated per field while the FlockComponent is instantiated dynamically per detected flock of birds.

Regarding the ensemble types, there are six of them – a top-level one, DroneProtectionSystem, and five nested ones. Figure 3 shows the ensembles hierarchy.
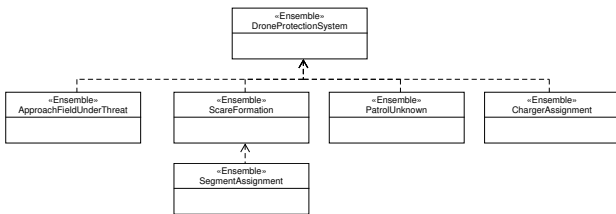
Figure 3: Ensembles hierarchy in the running example.

The top-level ensemble (starting at line 20) divides drones into two groups – ready-for-operation drones (line 22) and need-to-be-charged drones (line 23). The selection to these groups is based on the drone's current mode and energy level and is performed via functional-style operations. Similarly, the fields are divided to those under a threat (line 25) and fields with unknown status (line 24). As the fields are non-controllable components, they do not change their state actively; instead, the drones monitor them during patrolling for occurrences of flocks and the fields' state is updated accordingly. Finally, the unoccupied chargers are selected (line 26).

The sub-ensembles ApproachFieldUnderThreat and ScareFormation work in unison to protect a field under threat. First the ApproachFieldUnderThreat is instantiated and used to bring the necessary number of drones to the field. While the rest of the drones are arriving, the drones already present in the field are used to pursue the flock.

This is expressed by the situation construct, which expresses the situation in which the ensemble should be formed. The selection of drones is determined by lines 31 and 37 which select operational drones that are close to the field. Specifically, the utility construct specified the optimization function for the ensemble.

The tasks construct then provides instructions for the components-members of the ensemble instance (but only to the controlled components). In this case, as the ensemble is formed when there are not yet enough drones to scare away the flock, the task of its members is to move towards the birds to at least disturb them. If more then one drone is already present over the field, the drones disperse to approach the flock in a wider formation.

The ScareFormation ensemble (starting at line 61) replaces the previous ensemble when a necessary number of drones is present over the field. (The situation – line 67 – is an opposite condition to the one in ApproachFieldUnderThreat ensemble.) The ScareFormation ensemble disperses the drones over the field so that they fully cover

it and the flock has no other place to move over on the field. To do so, the ensemble identifies the target coordinates over the field and assigns drones to them in such a way that the sum of distances of the drones to the target positions is minimized.

To simplify the definition, the ensemble contains the sub-ensembles SegmentAssignment (line 69), each of which pairs one drone with one target position. The rules function (line 77) in the ScareFormation ensemble registers the SegmentAssignment sub-ensemble instances to be created and used in the ensemble resolution process. The constraint construct (line 83) defines additional conditions that have to hold – in this case the fact that no drone can be assigned to two field segments.

The PatrolUnknown ensemble (line 86) is used to guide a drone to a field that has unknown status. Technically, it groups a drone with a field with unknown status, while Its utility makes sure that the closest drone is selected.

The ChargerAssignment ensemble (line 95) is instantiated per charger place and assigns a drone that has a low energy level to the charger. The utility selects the drone that is the closest to the charger and has the energy level closer to the threshold.

The ensemble construct on lines 103–106 registers the sub-ensembles of the top-level ensemble and prescribes that the PatrolUnknown ensemble is instantiated per field with unknown threat status, the ChargerAssignment instantiated per free charger, and ApproachFieldUnderThreat and ScareFormation ensembles per a field under threat.

The utility of the top-level ensemble (line 108) ensures that the scaring the flocks and charging have higher priority than assigning a drone for a field with unknown threat level (the value divided by 4 at line 111).

Finally, the top-level ensemble constraint ensures that drones assigned for chargers are different ones (i.e., a single drone is not assigned to two chargers) and similarly drones assigned for scaring are different ones (i.e., the ScareFormation ensemble instance groups different drones).

4 IMPLEMENTATION AND LESSONS LEARNED

We have implemented the scenario as a demo which comprises (a) the ensemble-based specification (as shown and described in Section 3); (b) the solver for ensembles that implements all the ensemble constructs and employs a constraint-solver to determine which ensembles should be instantiated in a given situation and what components should play which roles in them; (c) implementation and simulation of components (drones, flocks, fields, chargers); and (d) animated web-based visualization. The code to this whole package is available as open-source at https://github.com/smartarch/afcens.

The visual part is implemented as a Javascript web-based client and a NodeJS server, the simulation and ensemble-resolution part is implemented in Scala using the Akka agent framework[2]. The choice of Scala allows us to construct the internal DSL that is used to specify the ensembles. The choice of Akka as the runtime framework allows us to run the simulation as multithreaded and also to run multiple simulations concurrently.

The simulation can be run in two modes – one in which ensembles are not used and the drones are configured to take decisions

---

[2]https://akka.io/

independently of one another (we call this *baseline implementation*, and another, which relies on group-behavior specified via ensembles (we call this *ensemble-based implementation*).

The behavior of a drone in the baseline mode is such that if it sees flock within its observation radius that is above a field, it follows it. If it sees no flock (or all flocks it sees are above some non-field area), it starts patrolling – that it is goes from one field to another in a pre-defined order. If its energy drops below a given threshold, it moves to the charging station. If all charging stations are occupied, it hovers above them until any gets available, then it charges itself. Once fully charged, it returns to the behavior already described.

The ensemble-based mode works as described in Section 3. The constants used as observation radius, threshold for charging, etc., are the same for both the modes.

We have conducted several experimental runs with the demonstrator in which we varied the number of flocks (3 to 5), the number of drones (3 to 4), and the constants governing the behavior of the flocks and strategies used to control and coordinate the drones. As a metrics to compare different runs of the simulation, we use the total amount of time the different flocks spend undisturbed on a field.

We have not tried not try to draw systematic and generalizable results of these experiments. Since we have ourselves provided the implementation of the system without ensembles that we use for comparison, even though we have tried to make it sufficiently smart, it would pose a serious threat to the construct validity of any experimental result. Thus, instead, we present in the rest of the Section a few lessons learned that give more intuitive understanding of these systems. We also invite everyone to experiment with the demonstrator and get the feel for these collaborative systems her/himself.

4.1   Lessons Learned

1) Probably the biggest lessons learned for us was that the difference between the baseline case and the version with ensembles was not overly dramatic – only about one third (i.e., in the baseline implementation, the flock could spent by about 30% more time on the field than in the ensemble-based implementation). Bigger difference however was in the variance across different simulations. The ensemble-based implementation seems to provide more stable results.

On the other hand, the baseline implementation, which does not coordinate among components, tends to easily fall into certain highly inefficient situations. Even in this relatively simple scenario, it sometimes happens that drones get aligned and all pursue the same flock. As they are aligned, they do not cover large enough area of the field to scare the flock away completely, they only push it from one corner of the field to another. Meanwhile, the other flocks just undisturbed consume the other fields. This situation is broken only once the drones start running out of battery, which brings them in the middle to recharge, spatially separates them (because the chargers are located apart) and effectively resets their behavior.

This kind of singular behavior emerges because they all pursue the same goal and they do not distribute tasks among themselves

(which is done by ensembles). The overall performance of the drones thus highly depends on how long the systems can work before falling into the singular behavior, which is also the reason for the higher variance. On the contrary, by coordinating the drones, the ensembles effectively counteract the singular behaviors in the system.

We see this as one of the biggest benefits in explicitly specifying the coordination among components – the specification allows us to bound and avoid certain emergent behavior (i.e., the one that contradicts the logical specification of ensembles).

2) Another observation we made in creating the demo is that it is surprisingly hard to design the coordination. It is all natural that in the ensemble-based implementation, the ensemble takes some of the component's logic, which leaves the component relatively lightweight and the emphasis lies on the ensembles. This was indeed our case too. In the ensemble-based implementation the drone's behavior is limited mostly to low-level control/simulation of the movement, while the ensemble sets the destinations to move to.

Nevertheless, thinking about the coordinated behavior felt qualitatively more difficult than designing a single agent. A part of it was also that we tried to stick with the principle that the state of the ensemble (and also the condition whether to create it or not) should be fully based on the state of the components. We designed the ensembles such that they do not have any internal state themselves and, in particular, do not depend on history of previous states of the ensemble. This creates a challenge upfront because we (as software engineers and developers) are trained to think in state-machines rather than in invariants. However, once designed, this type of a system has simpler specification and is more robust. This is because all rules governing the establishment of an ensemble relate to the current situation, rather than on the pair of <previous state, current situation> which yield many more options that have to be handled and are prone to over/underspecification.

3) An interesting phenomenon that we also observed is that the system that is specified in the declarative way using ensembles exhibits self-optimizing/self-stabilizing behavior. In particular, we observed this in the case of patrolling. If not interrupted by situation when some flock is in the field or battery would be running out, the drones relatively equally spread over the map and stop at positions which allow them to see the fields. The drone would typically find its spot among two adjacent fields, so that one drone can monitor both the fields. If two fields that are further apart are to be covered by a single drone, the drone would just identify the shortest line between the fields and would move on this line from one end to another, just far enough to see the whole field. This yields very efficient movement – minimal enough to gather all necessary observations. The placement of the drones is not globally optimal, however, it is still good enough. As we did not include any randomness factor that would push the system out of such local minimum, the system would stay there until recharging is needed or some field is attacked by the flock. After such a disturbance ceases, the drones return to similar near-optimal formation.

It is necessary to note that this exact kind of behavior is not directly specified in the ensembles. The patrolling ensemble only states that (i) a drone should fly to the closest field which needs observation (observations are remembered for 500 seconds of the simulated time), and (ii) two drones should not patrol the same field.

The formation of the drones and the fact that they find an optimal vantage point where they stop is behavior that emerges from the specification. Similar to what we described in #1, the ensembles regulate the emergent behavior. In this case, the emergent behavior is formally compliant with the specification of the patrolling ensemble.

5 RELATED WORK

Software engineering for collaborative software systems has been a topic of active research fueled by European projects such as AS-CENS[3]. The research line that focuses on autonomic component ensembles has two characteristics. First, it assumes that such collaborations are *dynamic* and depend on the runtime state of the system and its environment. In other words, the uncertainty related to which members to group together in ensembles, when, and for how long, has to be resolved at runtime, which makes such systems inherently self-adaptive. Indeed, forming and absolving ensembles at runtime can be seen as the planning function of a high-level Monitor-Analyse-Planning-Execute over Knowledge (MAPE-K) loop [11], whereas each autonomic component can be seen as a lower-level MAPE-K loop. Second, it focuses on architecture abstractions and corresponding architecture and specification languages that can be used for handling the challenge of designing and implementing such systems.

In particular, in our previous work we have introduced the DEECo component model [2, 3] for the specification and deployment of ensemble-based systems. Similar to the approach of this paper, DEECo employs the notion of components with state and periodically triggered methods. Ensembles in DEECo also have a membership condition that is specified based on the components types and knowledge. Although ensembles can overlap in DEECo, allowing a component to belong to more than one ensembles, there is no ensemble hierarchy. We have seen though that in many real-life collaborative systems, it is beneficial to be able to specify collaboration hierarchies, which is possible in our approach. Helena [8] is another framework for building ensemble-based systems. In Helena, an ensemble is a set of roles, where a role is specified by its attributes and operations, and a set of role connectors. The main difference between Helena and our approach or DEECo is that in Helena, a component explicitly indicates which ensembles it belongs to. jRESP [15] is another ensemble-based framework. It implements the Software Component Ensemble Language (SCEL) language [14]. In jRESP, components, called nodes, dynamically form ensembles by relying on the attribute-based communication paradigm. While not strictly ensemble-based, AbaCuS [1] can be also considered here as it employs opportunistic and attribute-based communication among components and it is based on the AbC calculus. Another implementation of this calculus is ABEL [5], which is also a DSL for specifying attribute-based communication among components. In contrast to our approach for DSL creation, ABEL is rather only a set of API calls for the Erlang programming language.

The core features of ensemble-based systems are modeled in [9] where a formal semantics of communication within ensembles is proposed and it is based on dynamic logic and bi-simulation.

Apart from smart farming, autonomic component ensembles have been employed in other application domains where collaboration is important. In particular, they have been used in mobility scenarios, e.g., in modeling vehicles collaborating to achieve smart parking functionality [12]. Another class of applications is emergency collaboration scenarios, e.g., collaboration between firefighters [6] or agents in the Robocup Rescue Simulation[4] [7].

Finally, in the robotics domain, collaborations between robots that need to move (similar to drones flying together) or push an object in synchrony are naturally modeled via autonomic component ensembles. In our previous work, we have provided a testbed for experimenting with ensembles in the robotic domain [13]. In particular, the testbed combines ROS (Robotic Operating System), the corresponding Stage simulator, OMNET++ (a network simulator supporting ad-hoc networks), and jDEECo (our Java framework for developing and running DEECo-based systems). It allows the simulation of a number of robots (Turtlebots in particular) that need to coordinate in cleaning an indoor space consisting of corridors and office rooms. This robotic testbed focuses on issues stemming from sensing and environment uncertainty, featured mainly via imprecise localization and deadlock situations in the navigation of the robots. Ensemble-based coordination is meant to deal with these issues, e.g., via having an ensemble of two robots that exchange their destinations in order to avoid a deadlock. On the contrary, our smart farming use case focuses on increasing the *efficiency* of coordinated drone operations via building complex ensembles. Also, in the work presented here, we extended our DSL for ensembles with several constructs that allow for easier specification and faster ensemble formation.

6 CONCLUSION

In this paper, we have presented an approach of autonomically composable and context-dependent constructs for describing cooperation among autonomic components together with a demonstration implementation of the approach in the domain of smart farming. For defining the cooperation constructs, an internal Scala-based DSL has been developed and also a system for evaluation of the dynamic groups at runtime. The implementation is available as an open-source and ready for further experiments. The proposed approach has been already applied in an ongoing international project and discussed with industrial partners within the project.

As a near-future work, we plan to further enhance the capabilities of our approach in modeling and resolving ensembles and extend accordingly our DSL. Also, we plan to further use the approach within the AFarCloud project and other projects and deploy it in an actual environment.

---

[3]http://www.ascens-ist.eu/

[4]http://roborescue.sourceforge.net

REFERENCES

[1] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. 2016. Programming of CAS Systems by Relying on Attribute-Based Communication. In *Proceedings of ISOLA 2016, Corfu, Greece (LNCS)*, Vol. 9952. Springer, 539–553. https://doi.org/10.1007/978-3-319-47166-2_38

[2] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. 2013. DEECo: An ensemble-based component system. In *Proceedings of CBSE 2013, Vancouver, Canada*. ACM, 81–90. https://doi.org/10.1145/2465449.2465462

[3] Tomas Bures, Frantisek Plasil, Michal Kit, Petr Tuma, and Nicklas Hoch. 2016. Software Abstractions for Component Interaction in the Internet of Things. *Computer* 49, 12 (2016), 50–59.

[4] Tomáš Bureš and Petr Hnětynka. 2019. *Formal Semantics of Component Ensembles*. Technical Report D3S-TR-2019-01. Charles University, Faculty of Mathematics and Physics, Department of Distributed and Dependable Systems. https://d3s.mff.cuni.cz/sites/default/files/publications/bures_formal_2019.pdf

[5] Rocco De Nicola, Tan Duong, and Michele Loreti. 2019. ABEL - A Domain Specific Framework for Programming with Attribute-Based Communication. In *Proceedings of COORDINATION 2019, Lyngby, Denmark (LNCS)*, Vol. 11533. Springer, 111–128. https://doi.org/10.1007/978-3-030-22397-7_7

[6] Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, Adam Hujecek, Frantisek Plasil, and Dominik Skoda. 2017. Strengthening Adaptation in Cyber-Physical Systems via Meta-Adaptation Strategies. *ACM Transactions on Cyber-Physical Systems* 1, 3 (July 2017), Article No. 13.

[7] Ilias Gerostathopoulos, Dominik Škoda, František Plášil, Tomáš Bureš, and Alessia Knauss. 2019. Tuning Self-Adaptation in Cyber-Physical Systems through Architectural Homeostasis. *Journal of Systems and Software* 148 (2019), 37–55. https://doi.org/10.1016/j.jss.2018.10.051

[8] Rolf Hennicker and Annabelle Klarl. 2014. Foundations for Ensemble Modeling – The Helena Approach. In *Specification, Algebra, and Software*. Number 8373 in LNCS. Springer, 359–381. https://doi.org/10.1007/978-3-642-54624-2_1

[9] Rolf Hennicker and Martin Wirsing. 2018. Dynamic Logic for Ensembles. In *Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems*. LNCS, Vol. 11246. Springer, 32–47. https://doi.org/10.1007/978-3-030-03424-5_3

[10] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. 2008. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*. Paris, France, 1–10. https://hal.archives-ouvertes.fr/hal-00483090

[11] Jeffrey Kephart and David Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.

[12] Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. 2015. An Architecture Framework for Experimentations with Self-Adaptive Cyber-physical Systems. In *Proceedings of SEAMS 2015, Florence, Italy*. 93–96. https://doi.org/10.1109/SEAMS.2015.28

[13] Vladimir Matena, Tomas Bures, Ilias Gerostathopoulos, and Petr Hnetynka. 2016. Model Problem and Testbed for Experiments with Adaptation in Smart Cyber-physical Systems. In *Proceedings of SEAMS 2016, Austin, USA*. ACM, 82–88. https://doi.org/10.1145/2897053.2897065

[14] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. 2013. A Language-Based Approach to Autonomic Computing. In *Formal Methods for Components and Objects*. Number 7542 in LNCS. Springer, 25–48. https://doi.org/10.1007/978-3-642-35887-6_2

[15] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. 2014. A Formal Approach to Autonomic Systems Programming: The SCEL Language. *ACM Transactions on Autonomous and Adaptive Systems* 9, 2 (July 2014), 7:1–7:29. https://doi.org/10.1145/2619998

[16] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Frantisek Plasil, Filip Krijt, Jiri Vinarek, and Jan Kofron. 2020. A Language and Framework for Dynamic Component Ensembles in Smart Systems. *International Journal on Software Tools for Technology Transfer (STTT)), in press* (2020).