Department of Distributed and Dependable Systems
Technical Report no. D3S-TR-2017-01
December 22, 2017

# Trait-based Language for Smart Cyber-Physical Systems

Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Frantisek Plasil, Filip Krijt, Jiri Vinarek, Jan Kofron

**Abstract**: The problem this paper aims to target is how to hoist the cooperation of software components, acting as autonomous agents and forming coalitions, at the architectural level in smart cyber-physical systems (sCPS). This is a hard problem as coalitions can be overlapping, nested, and dynamically formed and dismantled based on several criteria. To target this issue, we propose and implement an architecture description language (TCOF-ADL) based on Scala internal DSL, that describes architecture and formation of dynamic coalitions of components. To raise the level of expressivity, we introduce the concept of domain-specific extensions (traits) of the core TCOF-ADL to reflect different concerns—such as movement in a 2D map, state-space modeling of physical processes, statistical reasoning about uncertainty. This allows configuring the ADL for the needs of a specific application case and facilitates reuse. To evaluate our approach, we show how it can be beneficially used in addressing the coordination of agents within the RoboCup Rescue Simulation League.

**This work was partially supported by the project no. LD15051 from COST CZ (LD) programme by the Ministry of Education, Youth and Sports of the Czech Republic.**

# Trait-based Language for Smart Cyber-Physical Systems

Tomas Bures[1], Ilias Gerostathopoulos[2], Petr Hnetynka[1], Frantisek Plasil[1], Filip Krijt[1], Jiri Vinarek[1], Jan Kofron[1]

[1] Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{surname}@d3s.mff.cuni.cz

[2] Fakultät für Informatik, Technische Universität München, Munich, Germany
gerostat@in.tum.de

**Abstract.** The problem this paper aims to target is how to hoist the cooperation of software components, acting as autonomous agents and forming coalitions, at the architectural level in smart cyber-physical systems (sCPS). This is a hard problem as coalitions can be overlapping, nested, and dynamically formed and dismantled based on several criteria. To target this issue, we propose and implement an architecture description language (TCOF-ADL) based on Scala internal DSL, that describes architecture and formation of dynamic coalitions of components. To raise the level of expressivity, we introduce the concept of domain-specific extensions (traits) of the core TCOF-ADL to reflect different concerns—such as movement in a 2D map, state-space modeling of physical processes, statistical reasoning about uncertainty. This allows configuring the ADL for the needs of a specific application case and facilitates reuse. To evaluate our approach, we show how it can be beneficially used in addressing the coordination of agents within the RoboCup Rescue Simulation League.

**Keywords:** smart cyber-physical systems; autonomic components; component coalitions; component ensembles; architecture description language

## 1 Introduction

Smart Cyber-Physical Systems (sCPS) consist of hardware and software components that need to operate with each other at the syntactic level (API matching, language interoperability), at the semantic level (common vocabulary, contracts for assume-guarantee reasoning), and at the strategic level (sharing of goals, cooperation among components). In this work, we are concerned with the modeling of cooperation of software components at the strategic level. We assume that:

- Components act as autonomous agents with their own belief, capabilities, and lifecycle.
- There are system-level tasks to be performed and joint goals to be achieved (thus not belonging to any individual component but to the composite system).

- Components form cooperation groups—coalitions—in order to perform the system-level tasks to achieve the required joint goals.

For example, consider an emergency coordination system where fire fighters and medical first responders carry mobile hand-held devices running software components supporting their individual missions. Each of those acts autonomously, evaluates the situation they are in and acts accordingly keeping in mind their own safety. At the same time, different coalitions can be formed between these components in order to let their bearers cooperate in the complex, multi-stakeholder tasks of rescue operations (e.g. moving as a group towards a fire scene and/or approaching those needing emergency medical care).

We believe that as sCPS will continue to be independently developed as autonomous entities with cooperation capabilities we will see more such coalitions in future systems. We thus view the modeling of coalitions in an intuitive, reusable and, at the same time, semantically rich way as an important challenge for sCPS. In this paper, we focus on how to model coalitions at design time and on how and under which conditions to form and dismantle coalitions at runtime.

Component *ensembles*, i.e. dynamic groups, have been suggested to hoist the cooperation concern of dynamically changing sCPS at the level of the architecture[1] [6]. This has been done in the context of specialized component models and languages such as SCEL [15], DEECo [6], Helena [9]. An ensemble determines (typically by a logical condition) which components are the members and in which particular roles of the ensemble they are. Further, it embodies the cooperation among the components towards some common goal. Conceptually, the ensemble operationalizes the goal.

Despite the work done so far in ensemble-based systems (including our own work in DEECo), we believe it is still hard for mainstream designers/developers to put these ideas to action in developing complex real-life systems where ensembles are overlapping, nested, and dynamically formed and dismantled. At the same time, building the necessary programming abstractions and machinery for ensemble specification and formation from scratch is both time-consuming and error-prone.

In this work, we try to provide a remedy by focusing on the specification (i.e. model and semantics) of ensembles realizing coalitions and autonomous cooperation within those coalitions. We specifically strive to address the simplicity and expressivity of the specification that aligns component-level and coalition-level goals. Taking a pragmatic approach, we have implemented a domain-specific language on top of Scala to ease the task of designers/developers in specifying and forming complex real-life ensembles.

Our approach is based on the observation that there are a number of recurring concepts in the specification of membership conditions for coalitions. For example, some coalitions are formed based on number and type of members (e.g. "group together 3 rescuers"), some on spatial constraints (e.g. "group together components that are physically close"), some on predictions of certain values/outcomes (e.g. "group together components based on the estimated number of components necessary to complete the task *A* in time"). We further observe that some of these concepts are independent of the particular sCPS application domain (e.g. number of members), whereas others depend

[1] Architecture hoisting is the ownership and management of a property by the architecture [8].

on the particular sCPS application domain (e.g. predictive functions, map-based routing functions). Such domain-dependent concepts can be captured in reusable extensions—*traits*—that can be used in addition to the core domain-independent specification abstractions.

**Goals:** The goal of the paper is to propose and implement an architecture description language (ADL) termed Trait-based Coalition Formation ADL (TCOF-ADL), in our case based on Scala internal DSL, that (1) supports separating responsibilities to components and coalitions, (2) allows easy and expressive definition of when to establish the coalition and how to select its members, and (3) includes the coalition formation to the process model of a component. Further, to increase the expressivity of coalition specification, we feature the concept of domain-dependent traits embodying particular domain-dependent concepts and introduce mixing of the traits with the core domain-independent features of the proposed ADL. This enables a designer to instantiate an expressive domain-specific ADL for describing autonomous components and coalitions for the particular application case in hand.

**Structure of the text:** Section 2 presents three use cases used for motivating our work on coalition formation in smart CPS. Section 3 gives an overview of the main ideas in the design of our ADL and describes both the core concepts of the language and their extension by domain-dependent traits. An evaluation on the reduce of development effort by using our ADL is provided in Section 4, together with a discussion of limitations of our approach. Finally, Section 5 compares our approach with other approaches for component coalition formation, and Section 6 concludes with an overview of the contributions.

## 2 Use cases

In this section, we describe three use-cases that motivate our work and our decisions in designing TCOF-ADL.

### 2.1 RoboCup Rescue Simulation

RoboCup Rescue Simulation[2] (RCRS) is a research and educational project targeted on evaluation of multi-agent solutions in disaster response scenarios. The research on the project is stimulated by the annual RoboCup competition, one of the most important competitions in robotics. RCRS provides a simulation platform that imitates a city after an earthquake. The simulation consists of a map of the city that includes streets, intersections, and buildings, and of stationary and platoon agents. Buildings may collapse due to the earthquake; they may also be on fire. Street fragments may be blocked by debris. Stationary agents include *Fire Stations*, *Police Offices* and *Ambulance Centers*. Platoon agents include *Fire Brigades*, *Police Forces*, and *Ambulance Teams*. Each type of platoon agent has specific tasks to achieve and different capabilities. Fire Brigades are responsible for extinguishing fires; Police Forces for removing blocking debris from the streets; Ambulance Teams for rescuing humans by unburying them and carrying

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

them to *Refuges (*special type of buildings). Importantly, platoon agents have a limited view of the world—based on their line of sight—and can communicate with each other and with the stationary agents either face-to-face (when they are close-by) or by transmitting messages via unreliable radio channels.

In this setting, one of the challenges RCRS raises is how to form and dismantle coalitions at runtime between platoon agents in order to efficiently coordinate search and rescue operations. Coalitions are formed as (potentially heterogeneous) groups of agents, with each agent featuring a particular coalition-specific role. Agents are selected for a particular role based on several criteria (which can be also combined), in particular:

- Based on **agent type**. A coalition between Fire Brigades and Police Forces can enable the former to get a route to the fire cleared of blocking debris.
- Based on **number of agents**. A coalition may require a certain number of Fire Brigades to cooperatively work on extinguishing the fire in a building.
- Based on a **soft optimization rule**. Among the Fire Brigades eligible for a coalition, prefer those that are closer to the fire.
- Based on **spatial proximity**. Platoon agents that are close-by can form a coalition in order to rendezvous and share updates regarding street blockages (by debris) via face-to-face communication.
- Based on **temporal proximity**. Only those buildings are extinguished which can be reached before they are burnt out (the RCRS simulator assumes burnt out buildings need not be responded by Fire Brigades).
- Based on **estimated cooperation effort**. Upon detecting a fire, Fire Brigades can form a coalition composed of the (estimated) minimum number of Fire Brigades necessary to prevent the fire from spreading to nearby buildings.
- Based on the **probability of effective cooperation**. Cooperation is decided based on the probability of successful communication via unreliable wireless channels. For example, if communication reliability falls below a certain level, a coalition which relies on regular rendezvous of agents to exchange data over close-range (i.e. face-to-face) communication is chosen over a coalition where all agents communicate via long-range radio.

## 2.2 Connected Mobility System

Recent initiatives in capital cities around the world (e.g. in Helsinki[3]) are pushing smart mobility models that tie together public transport modes with personalized ride-sharing services. A future Connected Mobility System (CMS) implements this vision by the cooperation of an array of agents. Principal agents are *Passengers* who use their mobile phone apps to request rides in the city. *Trams* and *Regional* and *Underground Trains* are also part of the system; they follow schedules of pre-defined routes with stops at designated locations in the city. *Cars* and *Buses*, owned by ride-sharing companies, are agents that follow dynamic on-demand routing. *Traffic Stations* are stationary agents

[3] http://www.cthreereport.com/helsinkis-plan-to-eliminate-cars-by-2025/

which gather data on the traffic intensity in different parts of the city. Communication between the agents is achieved via mobile or radio technology.

Similar to RCRS, there are a number of cases that necessitate cooperation between the agents in this complex system. Again, coalitions are formed based on several criteria, such as agent type and number (e.g. only a certain number of Passengers form a coalition to share a means of transport), spatial proximity, estimated cooperation effort (e.g. coalition among Cars or Buses in order to balance the transportation), agent preferences, etc.

### 2.3 Swarms of robots

Another example with strong emphasis on cooperation of autonomous agents are attempts in exploiting robotic *swarms*. When a failure or loss of a single robot is highly probable, as e.g. when moving and performing tasks in a challenging environment, swarms of robots are appealing. A swarm is composed of many typically cheap robots with limited functionalities compared to larger and more sophisticated robots. The advantage of the robotic swarm comes by achieving tasks collectively. For instance, robots in a swarm can collectively move a heavy obstacle or make a chain and cross a wide gap that a single robot would not be able to cross. Alternatively, they may even sacrifice a robot to achieve a particular goal (with quite some resemblance to the once-famous computer game Lemmings). This of course requires a good amount of cooperation among the robots in a swarm.

As in the cases in sections 2.1 and 2.2, coalitions have to be formed based on multiple criteria. For instance, several robots close to one another may group to move a boulder. Once they become part of the coalition, they are coordinated by the coalition to move at specified trajectories, thus effectively keeping a formation that acts together as a much larger and stronger robot. To form such a coalition, the spatial proximity is needed to select robots that can quickly come to the boulder. Similarly, estimated cooperation effort is needed to determine the required number of robots. Likewise, the probability of effective cooperation is useful to decide between alternative strategies to deal with a task. If there are multiple types of robots, the grouping and distribution of work to members of a coalition can be based on the actual type and capabilities of each robot (e.g. based on whether the robot is a flying drone or a two-wheeler).

## 3 ADL for Coalition Formation

In this Section, we generalize the criteria for coalition formation, described in the use cases of Section 2, into cooperation concepts and outline a Scala-based internal DSL that we have developed to specify coalitions and form them at runtime. A prototype implementation of Trait-based Coalition Formation ADL (TCOF-ADL) along with the engine forming the coalitions at runtime is available at http://github.com/d3scomp/tcof.

We split the criteria for coalitions in two categories: (i) core concepts that are independent of a particular sCPS application domain, (ii) concepts that depend on particular sCPS application domains; we group the latter into reusable *traits*. By the trait, we denote a set of specification concepts that extend the core of the specification language and can be used optionally (similarly to object-oriented languages, where a trait, or sometimes called a mixin, is typically a set of orthogonal methods that can be attached to a class to extend the class' behavior). The concepts featured by a trait are specific to a particular application domain (e.g. connected mobility, emergency coordination, home automation, robotic swarm, etc.) or to a particular aspect of the domain (e.g. navigation in 2D space). This makes the traits reusable across multiple use cases. For example, the "map" trait reflects the concept of spatial proximity and can be reused both in applications that belong to the connected mobility domain and in applications that belong to the domain of emergency coordination in a city. Any sCPS coalition can be specified by using the domain-independent concepts and augmenting them with selected traits.

In contrast, coalition criteria (concepts) that do not depend on particular application domains are universal in sCPS coalition specification, i.e. they typically make sense in any application domain. For example, the "type" of an agent is a possible criterion for coalition formation in any application (irrespective of its domain).

## 3.1 Core concepts

To capture the coalitions at the architectural level, we exploit *components* and *ensembles* [6]. A component is used to represent an agent, whereas an ensemble represents a coalition. Below we elaborate on them in detail.

A *component* represents an autonomic (and potentially mobile) entity. It consists of a belief (called *knowledge* in the paper) and periodic activity. Within its activity, the component operates over its knowledge and interacts with the environment (by sensing and actuating) and with other components (by sharing part of its knowledge). The knowledge conceptually comprises *local knowledge*, which reflects the state of the component itself, and a partial snapshot of the knowledge of other components, termed *mirror knowledge*.

While a component reflects the belief, activities and goals of an individual agent, the *ensemble* is used to reflect the shared belief, coordinated activities and joint goals of a group of agents (called *coalition* in this paper). An ensemble consists of a number of member components. It dynamically changes as the members' goals, knowledge and the ability to work in the ensembles change. To reflect this dynamic nature of the ensemble, the ensemble is determined by *membership condition*, which is a predicate defined over the components' knowledge. A membership condition example is: "Components that are spatially close to a point of interest". Components within an ensemble share parts of their knowledge to achieve the joint goal of the ensemble. In addition to conceptually representing the coalition, an ensemble actively drives and coordinates the cooperation of components by assigning roles to components within the ensemble (e.g. which component extinguishes the fire and which protects the nearby buildings by cooling them down with water) and by performing coalition-level computation. Technically, an ensemble can be established in centralized or decentralized manner. In the text below we describe the centralized case, in which an ensemble has an *initiator*, which is a component that establishes the ensemble, hosts its computation and performs communication with other member components to collect required knowledge for the ensemble and to distribute decisions of the ensemble back to its members.

To better reflect the processes and responsibilities in the real-world, we feature hierarchical decomposition of ensembles. The rule is that members of a sub-ensemble must be members of the parent ensemble too. As such, the highest-level ensemble corresponds to the overall joint goal and serves to divide responsibilities. Note, however, that in contrast to classical component models, ensembles are generally allowed to overlap. This naturally reflects the fact that a component may have multiple responsibilities (roles) and joint goals at the same time (e.g. refilling water and observing surroundings for potential fire).

As the ensemble is bound to a situation (e.g. firefighter coalition bound to a particular fire incident) rather than statically to particular components, the same ensemble can emerge simultaneously at multiple places involving different members. As such, we distinguish between *ensemble definition* and *ensemble instances*. As the names suggest, ensemble definition describes the membership condition and the coordination (roughly corresponding to a class in OO languages). The ensemble instance on the other hand is bound to a certain situation which satisfies the membership condition; it is bound to particular members and hosted by a particular initiator. For the sake of brevity, we stick in the paper to the general term ensemble. To avoid confusion, we specifically distinguish between ensemble definition and instance when necessary.

## 3.2 TCOF-ADL

We exemplify our approach on a part of the RCRS use case (Section 2.1). In essence, each of the key concepts—component, ensemble—is captured as an abstract class. These are then inherited by application-specific classes as illustrated in the example in **Fig. 1** using TCOF-ADL (our Scala-based internal DSL to describe coalition). Here, the component instance of `FireStation` coordinates with instances of `FireBrigade` in order to extinguish the fire of a building and protect the surrounding buildings. Similarly, there are components for other RCRS agents (`AmbulanceTeam`, etc.). Due to space limits, **Fig. 1** does not show how the instances of these classes are actually established; for simplicity let us assume there exists a singleton `FireStation` and *n* instances of `FireBrigade`. To achieve the required coordination the ensemble `FireCoordination`, initiated by `FireStation`, decomposes to two sets of ensembles—*`extinguishTeams`* and *`protectionTeams`*. These sub-ensembles are disjoint in terms of their location on the city map. Members of the ensemble `ProtectionTeam` are selected from *`brigades`* (instances of the component `FireBrigade`) and are associated with a particular `fireLocation`; the selection is determined by the membership clause specifying that only the instances of `FireBrigade` which are either idle or already at the scene given by `fireLocation` are considered. Additionally, they have to be at a distance from which they can reach the `fireLocation` before the `fireLocation` (i.e., building) burns down (otherwise, there is no reason to go there). Plus, their number has to be 2 or 3. In the action clause the "protection role" is assigned to the members—selected *`brigades`*. The ensemble `ExtinguishTeam` is specified in a similar way.

We represent both components and ensembles as Scala classes that extend the abstract classes `Component` and `Ensemble` respectively. Further, we use the power of Scala to define new control structures to structure the operation of components (i.e. separation of sensing, actuation and constraints about states and utility as described later in the section) and to declare membership condition and coordination of an ensemble. Technically, these control structures are realized as Scala functions with a "by-name" parameter [16].

**Components.** In support of self-adaptation, a component operates on a periodic basis by performing the classical MAPE-K loop [11]. This is done in the following steps.

In *Monitoring*, the component senses data from the environment, receives knowledge from other components and updates its knowledge model (both its local and mirror knowledge) accordingly. In TCOF-ADL, this is contained within the `sensing` construct (Fig. 1, lines 7-12 and 42-44).

In *Analysis*, the component determines the potential activities it can do in the given situation. To achieve conceptual autonomy of a component and to align it with behavior dictated by a coalition, components activities are tied to so called *states* (line 5). Each state determines a particular component activity (e.g. going to refill water, seeking refuge in case a firefighter is hurt). A component can be in multiple states at the same time, which corresponds to the ability to simultaneously perform a number of orthogonal actions (e.g. moving and observing environment). At the same time, TCOF-ADL allows defining logical predicates over states (lines 14-18), which serve to express dependency of the state on some particular value of component knowledge and the mutual exclusion of states. To break ties in situation where different conflicting states could be selected, TCOF-ADL provides a utility function that assigns values to states. The sum of values of active states then determines the overall utility of a component at given time (lines 24-26).

This whole leads to a constraint solving problem of determining the active states that maximize the utility of the component. This is resolved in *Planning*.

*Planning* further involves initiation of ensembles. This involves resolving the constraint solving problem stemming from ensemble specification (detailed in the next subsection). This is captured by the `ensembleResolution` construct (lines 46-49). The ensemble resolution can be guarded by the component being in a particular state, which captures the fact that a component can initiate ensembles only when it itself is in certain situation.

In *Execution* (contained within `actuation` construct—lines 19-22 and 51-59) the component performs the actuation and sends knowledge updates to other components (to members of initiated ensembles and to potential future ensembles' initiators). To abstract the communication for any particular technology and still provide suitable communication abstractions for coalitions, we exploit the attribute-based communication paradigm [2]. In this type of communication, the addressing is performed by a predicate over the knowledge of the receiving component (as opposed to static recipient identity). This makes it easier to disseminate knowledge needed to establish an ensemble towards ensemble initiators.

**Ensembles**. The definition of an ensemble is structured following the core concepts of coalition formation as discussed in Section 2. The selection of agent based on their type is represented by the `role` construct (line 74). It determines the potential components that can take responsibility in the ensemble in the given role. The actual selection of components is then based on the membership constraints (encapsulated by `membership` construct—lines 64-70 and 79-91) and the soft optimization rule (defined by the `utility` construct—lines 93-96). For example, in the `ProtectionTeam` ensemble the `membership` mandates that the `utility` is computed as inversely proportional to the travel time needed for each selected member fire brigade to get to the fire location. Coordination takes the form of updating certain coordination-relevant knowledge of the ensemble members (encapsulated by `coordination` construct—lines 98-102). The membership constraints include both the cardinality constraints (on the number of agents) and the domain-dependent constraints (such as pertaining to geographical proximity) which exploit the concepts featured by the traits (Section 3.3).

```
1.  class RescueScenario extends Model with RCRSConnectorTrait
2.     with Map2DTrait[MapNodeStatus] with StateSpaceTrait {
3.
4.   class FireBrigade(val entityID: EntityID) extends Component {
5.      val Protecting, Refilling, Idle, Escaping = State
6.
7.      sensing {
8.        sensed.messages.foreach {
9.           case (InitiatorToFireBrigade(receiverId, ..., fireLoc), _)
10.         // ...
11.      }
12.  }
13.
14.  constraints {
15.     (Escaping -> (brigadeHealth < MINOR_INJURY_THRESHOLD)) &&
16.     (Refilling -> (refillingAtRefuge || tankEmpty)) &&
17.      // ...
18.  }
19.  actuation {
20.     sendMessages()
21.     performAction()
22.  }
23.
24.  utility {
25.     states.sum(s => if (s == Protecting) 1 else 0)
26.  }
27.
28.  private def performAction(): Unit = state match {
29.     case Refilling if !refillingAtRefuge => moveTo(nearestRefuge)
30.     case Escaping if !regeneratingAtRefuge => moveTo(nearestRefuge)
31.     case Protecting =>
32.       if (inExtinguishingDistanceFromFire) extinguish()
33.       else moveTo(assignedBuildingOnFire)
34.     case _ => rest()
35.   }
36.  // ...
37. }
38.
39. class FireStation(val entityID: EntityID) extends Component {
40.   val fireCoordination = root(new FireCoordination(this))
41.
42.   sensing {
43.     processReceivedMessages()
44.   }
45.
```

component

```
  ensembleResolution {
    fireCoordination.initiate()    //establishes a number of ProtectionTeam
                                   //and of ExtinguishTeam instances
  }

  actuation {
    for (protectionTeam <- fireCoordination.protectionTeams.selected)
      for (brigade <- protectionTeam.brigades.selected) {
        val message = InitiatorToFireBrigade(brigade.entityID,
            brigade.brigadeState, brigade.assignedFireLocation)
        agent.sendSpeak(time, Constants.TO_AGENTS, Message.encode(message))
        // ... likewise for ExtinguishTeam
    }
  }

class FireCoordination(coordinator: FireStation) extends Ensemble {
  val extinguishTeams =
        ensembles(buildingsOnFire.map(new ExtinguishTeam(coordinator, _)))
  val protectionTeams =
        ensembles(buildingsOnFire.map(new ProtectionTeam(coordinator, _)))

  membership {
    extinguishTeams.map(_.brigades)
                  ++ protectionTeams.map(_.brigades)).allDisjoint
  }
}

class ProtectionTeam(fireLocation: EntityID) extends Ensemble {
  val brigades = role("brigades",components.select[FireBrigade])
  val routesToFireLocation = map.shortestPath.to(fireLocation)
  val firePredictor = statespace(burnModel(fireLocation), time,
                                        fireLocation.status.burnoutLevel)

  membership {
    brigades.all(brigade =>
     (brigade.state == Idle) ||
     (brigade.state == Protecting) &&
      sameLocations(brigade.assignedFireLocation)
    ) &&
    brigades.all(brigade =>
      routesToFireLocation.timeFrom(mapPosition(brigade)) match {
         case None => false
         case Some(travelTime) => firePredictor.valueAt(travelTime) < 0.9
      }) &&
    brigades.cardinality >= 2 && brigades.cardinality <= 3
  }

  utility {
    brigades.sum(brigade => travelTimeToUtility(
                  routesToFireLocation.timeFrom(mapPosition(brigade))))
  }

  coordination {
    for (brigade <- brigades.selectedMembers) {
        brigade.assignedFireLocation = Some(fireLocation)
      }
    }
  }

  class ExtinguishTeam(fireLocation: EntityID) extends Ensemble { /* ... */ }
 }
```

component

ensemble

ensemble

**Fig. 1.** Example of using TCOF-ADL in forming coalitions

## 3.3 Expressivity through domain-dependent traits

The core concepts described above allow for specifying entities (components), the features of these entities (components' knowledge) and coalitions of entities (ensembles). However, conditions for coalition creation (i.e., the membership condition) lack an ability to express real-world conditions such as that one component is spatially close to another one or that a building does not burn down before a firefighter unit reaches the building, etc. The type of such necessary conditions strongly depends on the particular domain of an application. To include all the possible types of conditions to the core of the specification language is not only impractical, as the language would be quite complex and hard to learn, but even impossible, as all the possible application domains cannot be foreseen. Plus, a single application typically would not need all the condition types. Indeed, all the examples in Section 2 specify conditions over spatial distances and estimated travel times. However, while the RCRS use case (Section 2.1) prescribes conditions over estimates of fire spreading/burning speed, the connected mobility system example (Section 2.2) prescribes conditions over estimates of traffic congestions and vehicle speeds. Thus, in our approach, all these domain-dependent condition types are designed as reusable traits; designers/developers can pick and use or create only the traits necessary for their application.

In the rest of the section, we overview several traits that are already available in TCOF-ADL. As with the core concepts, we illustrate them on the RCRS example in **Fig. 1**. The selected traits (line 1) are the *map trait* and *data prediction trait* plus a trait connecting the language run-time with the Rescue simulator (i.e., creates particular agents and processes messages from/to the simulator—not explained here). Technically, our traits are developed as Scala traits.

**Map Trait.** This trait serves to capture spatiotemporal relations between agents to be included in a coalition. The typical use is to select the agents that are close to each other or close to a particular point in terms of travel time. An example is on lines 75, 86, 94-95. Line 75 computes the shortest routes to a fire location (via Dijkstra's algorithm). Lines 86 and 94-95 query the computed travel time needed for a Fire Brigade to reach the fire location.

**Data prediction Trait**. This trait serves to capture coalitions formed based on the prediction of a data value in the system. Such predictions can rely either on state-space models that capture data evolution based on physical processes [1] or on machine learning models that capture patterns and trends in historical data. Examples of application of this trait include (i) the coalition of "Agents within travel time less than the estimated time until building B is burnt out" and (ii) the coalition of "Agents within travel time less than the estimated time-to-survive of victim V".

In our TCOF-ADL, the former is captured by lines 76 and 88. Line 76 initializes a predictor of how quickly a particular building (`fireLocation` in the code) burns based on the burning model of a building represented as an ordinary differential equation (ODE), which is assumed to be associated with each building, and initial conditions—i.e. current time and the current burnout level of the building.

The predictor uses a solver (i.e. a numerical integrator) to solve the ODE for a specified point of time (line 88). By combining the *Map2DTrait* and the *StateSpaceTrait* (data prediction), lines 86-88, it is ensured that “All agents selected for the coalition have to be able to reach the building (i.e. travel time is not None) and the burnout level of the building at the time the agent reaches it has to be below 0.9 (i.e. the building is not burnt out yet).

**Statistics Trait**. This trait offers the possibility to construct a coalition based on statistical evidence about the behavior of certain stochastic processes in the system. Here, we build on our previous work in mode-switching based on statistical tests [7]. Due to space constraints, we do not demonstrate this on the example in **Fig. 1** (as this would necessitate including other parts of the scenario); instead, we give an illustration below.

Consider the coalition that heavily relies on radio communication, so that it can be formed only if “Expected packet delivery probability over the radio is 90 percent or more, evaluated over the past one hour with a confidence of 95%”. This would be captured in our DSL as msgDelivery(time - 3600, time).probability > 0.9 withConfidence 0.95, where *msgDelivery* is a Boolean timeseries recording whether an expected packet was received or not. The whole expression denotes a one-sided statistical test whether one can reject the null hypothesis that the samples over the last hour have probability of true less or equal to 0.9 with significance level $\alpha = 0.05$.

Note that the above traits can be reused in some sCPS application domains, but not in others. For example, the data prediction trait for fire and burn out levels can be reused in applications belonging to the emergency coordination and home automation domains, but not to the connected mobility one.

## 4 Evaluation and Discussion

### 4.1 Code size and level of reuse

To compare the development effort when using TCOF-ADL against not using it, we have developed two versions of the RCRS example described in sections 3.2 and 3.3—one exploiting the TCOF-ADL framework and one without the TCOF-ADL (both are available at http://github.com/d3scomp/tcof). The TCOF-ADL-based implementation is formed by a main class (with nested classes) of 256 lines of code (LOC) (without blank lines and comments) and 6 additional classes holding some auxiliary functions. The main class with the auxiliary classes amounts to 409 LOC in total. It further uses three reusable traits—a connector to the RoboCup simulator, *map trait* and *data prediction trait.* Each of these traits is implemented as a set of classes with the overall sizes of 216, 273 and 70 LOC, respectively. Thus, in total, approximately half of the code is the business logic of the example and half is the reusable traits.

The implementation without TCOF-ADL is a single Scala class (with nested classes) with domain-dependent concepts (corresponding to the traits above) embedded in its code. Since it cannot take advantage of the solver used in TCOF-ADL, it implements simple search heuristics to figure out the cooperation groups (corresponding to ensembles in TCOF-ADL). Overall, it amounts to 1064 LOC—i.e. roughly two times the TCOF-ADL-based solution. Note that this is less than the TCOF-ADL-based solution plus the reusable traits. This is because the abstractions introduced in the traits to make them use case independent add to the total size of the reusable traits. These are of course missing in the implementation without TCOF-ADL. Though these measurements are difficult to generalize to other examples, they suggest that the reusability of traits can bring advantage already when the trait can be reused across two use-cases. Also, TCOF-ADL removes (at least in the less complicated cases where an ensemble can be expressed via a logical predicate and utility) the need to provide code that figures out the composition of cooperation groups. The process of resolving the cooperation groups typically leads to developing some heuristics that is likely combined with backtracking. Such heuristics with backtracking is intrinsically difficult to develop and debug. As such it corresponds to significant development effort, even though, due to recursion, the eventual amount of LOC is relatively small (39 LOC in our case).

## 4.2 Limitations – Coping with exponential complexity

The current biggest limitation of our approach is scalability w.r.t. to the exponential complexity of ensemble resolution. We have tested our implementation with 1 `FireStation` and 5 `FireBrigades` on the Kobe map (that is provided with the RoboCup simulator), which has 755 buildings and 1602 roads. For up to 3 fires in the map, the implementation we provide resolves ensembles within 1 second. However, with more fires, the run time grows exponentially (20 s for 6 fires, 120 s for 7 fires).

Internally, our implementation (which we provide as an open source library to resolve ensembles) uses a constraint solver for building ensembles (namely the Choco solver[4]). We could generally speed up the solving by incorporating a more optimized solver, however this does not solve the intrinsic complexity of the problem. In this respect, a promising solution seems to be a non-exhaustive search of the state space (by stopping the solver after some fixed time) and connect it with preconditioning the problem model such that reasonably good solutions are likely to be found first and likely suboptimal solutions are discarded upfront. This can be done by sorting the components in the order by which they are contributing to the utility and discarding components that have smaller contribution to the utility than a given threshold. In our case, it means sorting the `FireBrigades` by the distance to the fire and removing `FireBrigades` that are too far. Though this may result in finding a non-optimal solution or in not finding a solution at all (though it generally exists), our initial experiments suggest that even such simple means can significantly help in solving the problem without compromising too much the average quality of the system [12].

[4] http://www.choco-solver.org/

### 4.3 What about an external DSL or a graphical DSL?

In parallel with TCOF-ADL, we have designed a similar language but this time as an *external* DSL, i.e., an independent language (in contrast to TCOF-ADL which is an internal DSL since it is embedded in Scala). This DSL, dubbed the Ensemble Definition Language (EDL) [12], has been implemented with the help of the Eclipse DSL development stack—namely by employing Xtext and XTend technologies, as well as Ecore-based modelling tools. Due to space constraints, examples of the `ProtectionTeam` ensemble (with a similar functionality as the `ProtectionTeam` in **Fig. 1**) are not included here but can be found at the companion web-page dedicated to the technical aspects (http://github.com/d3scomp/tcof/blob/master/TECHNICAL.md). Compared with the internal-DSL way, the external-DSL way has different pros and cons. Due to having a separate compilation step and working with the model of the ensemble description instead of just data, EDL is directly capable of reflective code generation—it is therefore potentially more powerful. Additionally, because the external language design is not bound by the syntax of the host language, some concepts can be captured more naturally (e.g. cardinality can be written simply as "[2..3]"). In favor of the internal DSL speaks the tight integration with a general-purpose language and the ability to reuse all libraries that already exist for it. This makes it easy to introduce various domain-dependent traits that provide expressive constructs for reasoning about navigation, potentiality, etc.

Another option for developing a DSL would be a graphical one, e.g., as extension to UML (a UML profile). This has the advantage of providing intuitive understanding of the relationships between elements. In our case, however, since we attempt to capture the dynamic evolution of a system, structural diagrams (e.g. extended UML class diagrams) that captures a static snapshot are rather unsuitable. It has also been shown that while a graphical DSL is useful in providing big-picture view, text-based DSLs are more fitting in capturing the details of a complex application (like the connection of situations to logical constraints as in our case [18]).

## 5 Related Work

As mentioned in Section 1, component *ensembles* have been proposed to realize coalitions in sCPS. Till now, several frameworks based on the concept of ensembles have already been developed and applied. Helena [9], JRESP (http://jresp.sourceforge.net) and DEECo [6] are examples of them. They all offer components with roles and ensembles, however they do not provide self-formation of ensembles based on high-level specification constructs. The same holds for Ab$^{a}$CuS [3], which, though not an ensemble-based framework, employs opportunistic and attribute-based communication among components.

The coalition formation is intensively studied in a community around RCRS. Many approaches [17], based on different criteria, have been already proposed and implemented within the project, however their implementations are low-level (i.e., directly in Java, in which the whole RCRS simulator is developed) and thus hardly reusable. To solve the issue, the Agent Development Framework (ADF, https://github.com/ RCRS-ADF/RCRS-ADF) [19] has been recently proposed and started to be employed in order to allow easy reuse of code among multiple teams participating in the league. ADF defines a set of Java interfaces/base classes for individual aspects of the implementation, e.g., tactics for platoons and centers, path planning, communication, and the developers implement them and compose a complete system from them. This way, it provides basic structure for agents. It however lacks any ability of defining coalitions as first class architecture concepts. In general, ADF uses a similar technique as our approach, i.e., a core plus reusable traits, yet quite low-level (plus, at this time, without almost any documentation). Also, the traits (i.e., interfaces/base classes) are reusable only within the rescue league.

Another framework targeting reuse in the RCRS simulator is RMASBench [13] – a testbed for multi-agent coordination algorithms. Main focus of RMASBench is benchmarking of distributed constraints optimization problem (DCOP) algorithms. DCOP algorithms are used within RMASBench to solve coalition formation of the agents. API in RMASBench adds an extra channel that allows extensive exchange of messages between agents. DCOP algorithms in RMASBench are modelled as a reusable first-class entities and when a problem can be solved using an already developed algorithm, programmer needs to only implement a scoring function.

*Multi-paradigm* domain-specific languages and modeling languages have recently emerged [10] and they are quite similar to our approach. Multi-paradigm modeling targets a combination of multiple abstraction, formalism and (meta-)modeling levels into a single approach [14]. An example of such language can be QAML [5], which is a quantitative analysis modeling language constructed following the multi-paradigm modeling approach, i.e., for individual paradigm concerns reuses existing modeling languages (e.g., AADL for architecture, MathML for math expressions) and composes them together. However, compared to our approach, the language is not designed to be extensible and has no notion of coalitions. Another example can be found in [4], where the authors describe the process and challenges faced during designing a multi-paradigm-based software architecture description language—but again as the approach above, the language is not designed to be extensible and there is no notion of coalitions.

## 6 Conclusion

In this paper, we have focused on the problem of specifying and forming complex real-life coalitions of components in smart cyber-physical systems (sCPS). We have provided an extensible architecture description language that allows for specifying coalitions as component ensembles in an intuitive, reusable and, at the same time, semantically rich way. While a conservative way to design and implement a particular sCSP use case would be to design a dedicated DSL (basically from scratch), this obviously lacks reusability when multiple use cases featuring partly common concepts are to be considered. Thus, a strength of our approach is that it provides a compositional way to design a DSL for a sCSP use case featuring coalitions, by taking advantage of the overlap that exists in some sCPS application domains in terms of paradigms they exploit in modeling reality. Our implementation of the proposed architecture description language is publicly available as an open source library at http://github.com/d3scomp/tcof.

## References

1. Al Ali, R. et al.: Architecture Adaptation Based on Belief Inaccuracy Estimation. In: Proceedings of WICSA 2014, Sydney, Australia. pp. 87–90 IEEE CS (2014).
2. Alrahman, Y.A. et al.: On the Power of Attribute-Based Communication. In: Proceedings of FORTE 2016, Heraklion, Crete, Greece. pp. 1–18 Springer (2016).
3. Alrahman, Y.A. et al.: Programming of CAS Systems by Relying on Attribute-Based Communication. In: Proceedings of ISOLA 2016, Corfu, Greece. pp. 539–553 Springer (2016).
4. Balasubramanian, D. et al.: Taming Multi-Paradigm Integration in a Software Architecture Description Language. In: Proceedings of MPM 2014, Valencia, Spain. pp. 67–76 (2014).
5. Blouin, D. et al.: QAML: a multi-paradigm DSML for quantitative analysis of embedded system architecture models. In: Proceedings of MPM '12, Innsbruck, Austria. pp. 37–42 ACM (2012).
6. Bures, T. et al.: Software Abstractions for Component Interaction in the Internet of Things. Computer. 49, 12, 50–59 (2016).
7. Bures, T. et al.: Statistical Approach to Architecture Modes in Smart Cyber Physical Systems. In: Proceedings of WICSA 2016, Venice, Italy. pp. 168–177 IEEE (2016).
8. Fairbanks, G.: Architectural Hoisting. IEEE Software. 31, 4, (2014).
9. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling – The Helena Approach. In: Iida, S. et al. (eds.) Specification, Algebra, and Software. pp. 359–381 Springer (2014).
10. Horst, A., Rumpe, B.: Towards Compositional Domain Specific Languages. In: Proceedings of MPM 2013, Miami, USA. pp. 1–5 (2013).
11. Kephart, J., Chess, D.: The Vision of Autonomic Computing. Computer. 36, 1, 41–50 (2003).
12. Krijt, F. et al.: Automated Dynamic Formation of Component Ensembles. In: Proceedings of Modelsward 2017, Porto, Portugal. SCITEPRESS (2017).
13. Maffioletti, F. et al.: RMASBench: A Benchmarking System for Multi-agent Coordination in Urban Search and Rescue. In: Proceedings of AAMAS 2013, St. Paul, MN, USA. pp. 1383–1384 (2013).
14. Mosterman, P.J., Vangheluwe, H.: Guest Editorial: Special Issue on Computer Automated Multi-paradigm Modeling. ACM Transactions on Modeling and Computer Simulation. 12, 4, 249–255 (2002).
15. Nicola, R.D. et al.: A Formal Approach to Autonomic Systems Programming: The SCEL Language. ACM Trans. Auton. Adapt. Syst. 9, 2, 7:1–7:29 (2014).
16. Odersky, M. et al.: Programming in Scala: A Comprehensive Step-by-Step Guide, Third Edition. Artima Press (2016).
17. Parker, J. et al.: Exploiting Spatial Locality and Heterogeneity of Agents for Search and Rescue Teamwork. J. Field Robotics. 33, 7, 877–900 (2016).
18. Poch, T. et al.: Threaded behavior protocols. Formal Aspects of Computing. 25, 4, 543–572 (2013).
19. Takayanagi, K. et al.: Implementation of NAITO-ADF and its Team Design NAITO-Rescue 2015. In: Proc. of RoboCup Intl. Symp. 2015, Hefei, China. (2015).