A Language and Framework for Dynamic Component Ensembles in Smart Systems

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

***Abstract*—Smart system applications (SSA)—a heterogeneous landscape of applications of Internet of Things, Cyber-Physical Systems, and Smart Sensing Systems—are composed of autonomous yet inherently cooperating components. An important problem in this area is how to hoist the cooperation of software components forming dynamic groups —ensembles—at the architectural level of an SSA. This is hard since ensembles can overlap, be nested, and dynamically formed and dismantled based on several criteria. A related problem is how to combine component and ensemble specification with a well-established language supported on multiple platforms. To target these problems, we propose a specification and implementation language TCOEL (Trait-based Component Ensemble Language) based on Scala internal DSL, to describe both the architecture and formation of dynamic ensembles of components and their functional internals. To raise the level of expressivity, we introduce the concept of domain-specific extensions (traits) to the TCOEL core to reflect different paradigms' concerns—such as movement in a 2D map, state-space modeling of physical processes, and statistical reasoning about uncertainty. This allows for configuring TCOEL for the needs of a specific SSA use case, and, at the same time, facilitates reuse. To evaluate TCOEL, we show how it can be beneficially used in addressing the coordination of agents in a RoboCup Rescue Simulation application.**

***Index Terms*—architecture description language, autonomic components, component ensembles, smart cyber-physical systems**

I. Introduction

Smart systems manifest as heterogeneous, interconnected landscape of various applications of Internet of Things (IoT), Cyber-Physical Systems (CPS), and/or Smart Sensing Systems. A smart system application (SSA) is typically composed of hardware units running upon specific network(s) and of associated software components, achieving smartness by sensing and operation, both autonomous and collaborative. Responding to the dynamic nature of the environment in which the smart systems exist, such collaboration typically needs to be dynamically established to address situations localized both temporarily and spatially. From the software design and development perspective, such cooperation needs to take place at the syntactic level (e.g., API matching, language interoperability), at the semantic level (e.g., common vocabulary, contracts for assume-guarantee reasoning), and at the strategic level (such as sharing of goals performed by high-level tasks). In this work, we focus on cooperation of software components at the strategic level in the first place—assuming that they form dynamic groups, also known as component *ensembles* [1], [2]. Specifically, we presume that:

- There are strategic-level tasks to be performed and joint goals to be achieved in the given SSA.
- The SSA has strategic-level tasks that can be performed only by component ensembles.
- In the SSA, component ensembles are formed and dismantled dynamically, based upon the actual state of the SSA and its environment.
- Components are partially autonomous entities that can participate in ensembles (become their *members*).
- Components proactively sense the SSA's environment and provide their knowledge to other components to allow them to take smart and well-founded decisions.

For example, consider a smart emergency coordination system where fire fighters and medical first responders carry mobile hand-held devices running software components supporting the bearers' individual missions. Obviously, different ensembles can be formed of these components in order to let their bearers act autonomously and still 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).

There are many other smart systems where ensembles are inherently involved; for instance, it would be natural to apply them in the examples provided in [3], [4], such as on-street parking meters, employing swarms of sensors in a vehicle, and a number of applications in the area of road-side computing and intelligent transportation. Thus, specifying and implementing software component ensembles in an intuitive, reusable and semantically rich way is an important challenge in the area of smart systems.

Moreover, ensembles have been suggested to hoist the component cooperation and knowledge distribution concerns in a dynamically changing CPS at the level of software

[1] Tomas Bures, Petr Hnetynka, Frantisek Plasil, Filip Krijt, Jiri Vinarek, and Jan Kofron are with the Faculty of Mathematics and Physics, Charles University in Prague, Ke Karlovu 2027/3, 12000 Prague, Czech Republic (e-mail: {name.surname}@d3s.mff.cuni.cz).

[2] Ilias Gerostathopoulos is with the Department of Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam, De Boelelaan 1105, 1081 HV Amsterdam, Netherlands (e-mail: i.g.gerostathopoulos@vu.nl).

architecture[3] [1]. This has been done in the context of specialized component models and languages such as SCEL [6], DEECo [1], and Helena [7]. By a logical condition upon the ensemble and component actual states, an ensemble's description determines its member components and which particular roles in the ensemble they play. Further, it embodies the cooperation among the member components towards some strategic-level task(s). Ensembles should be formed according to a method that takes as input the overall goals of the system and decomposes them For details, including a method of ensemble design, we refer the reader to our work [8].

Problem statement. Despite the work done so far in ensemble-based systems (including our own work on DEECo), it is still hard to put these ideas into action in development of complex real-life systems where ensembles may overlap, be nested, dynamically formed and dismantled in a distributed environment built upon multiple network platforms. Obviously, in such settings, building the necessary programming abstractions and machinery for ensemble specification and formation from scratch is practically infeasible. What is missing in particular, is an easy, flexible, and elegant way of supporting the component and ensemble concepts in a well-established, "ubiquitous" programming language available at multiple platforms.

Main idea of contribution. We aim at addressing the problem mentioned above by proposing programming abstractions for ensemble and component specification. In particular, we strive to address and balance the specification simplicity and expressivity of the strategic-level tasks fulfilled by either an ensemble or by a single component. Taking a pragmatic approach, we propose a specification and rich architecture description language on top of Scala [9]. This potent multiple-purpose language, running on JVM, has been designed as extensible (scalable in Scala terms). Thanks to this property, it is possible not only use Scala as an implementation language but also as a specification, domain specific language by enhancing Scala's core features by domain specific constructs [10]. Moreover, owing to the use of JVM, Scala programs seamlessly interoperate with Java and other JVM-based languages; specifically, this allows an easy combination of specification of ensembles with existing libraries and SSA applications in Java.

Along these lines, we take advantage of the observation that there are a number of recurring concepts in articulating membership conditions of ensembles. For instance, some ensembles are formed based on number and type of members (e.g. "group together components representing 3 rescuers"), some on spatial constraints (e.g. "group together components representing firefighters 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 task *A* in time"). We have further observed that some of these concepts are independent of the particular SSA domain (e.g. number of members), whereas others depend on it (e.g. predictive functions, map-based routing functions). The latter concepts we propose to capture in reusable language extensions—*traits*, used as building blocks of specification languages in allied domains.

Goals. In summary, the main goal of the paper is to propose and implement a specification language for SSA termed Trait-based COmponent Ensemble Language (TCOEL), using the technique of Scala internal DSL [10]. TCOEL (1) supports partial autonomicity of components, (2) allows both nested ensembles (hierarchies) and overlapping ensembles, and (3) makes ensemble formation a part of component activities. Further, to increase the expressivity and reuse of ensemble descriptions, we introduce the concept of reusable traits embodying particular domain-dependent concepts and further device a way of mixing the traits with the core (domain-independent) features of TCOEL. This enables to tune up TCOEL for a specific application case.

An additional goal is to shortly report on Trait-based COmponent Ensemble Framework (TCOEF)—the runtime framework implementing TCOEL and providing some domain-dependent traits.

Structure of the text. Section II gives an overview of the main ideas in the design of TCOEL and describes the core concepts of the language. Section III presents the use case motivating our work on ensemble formation in SSA. In Section IV, the language TCOEL and framework TCOEF are described in more detail together with the TCOEL extension by domain-dependent traits. An evaluation on the application development effort when using TCOEL is provided in Section V, together with a discussion of limitations of the TCOEF framework. Finally, Section VI compares our approach to related work on component ensemble formation, and Section VII concludes with an overview of the contributions.

II. Core Concepts And Their Semantics

A. Core concepts

In our approach, a *component* is an autonomic (potentially mobile) entity, such as an agent mentioned later in Section III. It entails (i) a data structure (*knowledge*) reflecting its state and awareness of other components' partial state (in terms of belief [11]), last readings of its sensors and actuators, and (ii) periodic activity. Within this activity, the component operates upon its knowledge and interacts with other components and its environment by sensing and actuating. Thus, component's knowledge conceptually comprises *local knowledge*, which reflects the state of the component itself, and a snapshot of the partial knowledge of other components and environment (*mirror knowledge)*.

An *ensemble* groups a number of member components to support their cooperation. It dynamically changes as the members' knowledge and thus the ability to be a member is modified. This change is determined by satisfaction of *membership condition* indicated in the ensemble's

[3] Architecture hoisting is the ownership and management of a property by the architecture [5].

specification, which is a predicate defined over the components' knowledge and current ensemble status. For instance, a membership condition reads: "Not more than three components that are spatially close to a point of interest". The ensemble actively drives cooperation of components by (i) accepting one or more roles they offer within the ensemble (e.g., which component extinguishes the fire and which one protects the nearby buildings by cooling them down with water) and (ii) performing its strategic-level task(s) related computation (*task computation* for short). Technically, an ensemble has its *initiator*—a component that instantiates it, triggers its task computation, and performs communication with other member components to (1) collect required knowledge for the ensemble and to (2) distribute computational results of it back to its members so that they can update their knowledge, both local and mirror.

Ensemble type serves as a pattern for multiple ensemble instances. Each of the instances has its specific initiator (also referred to as "coordinator" in other works [12], [13]. For example, a specification of FireBrigade ensemble type serves as a pattern to creating FireBrigade ensemble instances, each responding one of the multiple fire incidents taking place at the same time. For the sake of brevity, by "ensemble" we mean "ensemble instance" if "type" is not explicitly mentioned.

To reflect the nature of processes and responsibilities of the real-world stakeholders in an SSA, hierarchical decomposition of ensembles takes place. The rule of thumb is that members of a child ensemble must be members of its parent ensemble too. As such, the highest-level ensemble – *root* (typically a singleton instance) – corresponds to an overall joint goal and serves to divide strategic-level tasks among its children. Nevertheless, multiple root ensembles may coexist in a single SSA – this is, in particular, beneficial in distributed settings. In a similar vein, ensembles can overlap in terms of sharing their members. This naturally reflects the fact that a component may have multiple roles by which can contribute to different strategic-level tasks (e.g. refilling water and observing surroundings for potential fire). In other words, a component can be a member of several ensembles at the same time.

B. Semantics

The semantics of ensembles and their instantiation can be seen as two separate problems; (a) how to determine and gather the state of components in a distributed system to correctly decide how to instantiate ensembles, and (b) how to instantiate ensembles provided that the state of components is known.

The gathering of knowledge needed to establish ensembles in distributed settings has been extensively tackled in our previous publications. As such, we do not focus on this in this paper and rather refer the reader to [1], [14], [15]. Likewise, we do not focus on the semantics of nondeterminism in component knowledge updates caused by their internal activities upon the local and mirror knowledge. Here we refer the reader to our work [16]–[18].

In the rest of this paper, we focus on how to specify and instantiate ensembles. Compared to the previous works on ensembles, we feature much richer semantics that covers hierarchical ensembles and also the ability to specify different variants of how to instantiate ensembles and decide the best.

Formally, we see the instantiation of ensembles as a constraint optimization problem. For the sake of dealing with this problem, we define components and ensembles as follows.

Component types and component instances

We distinguish *component types* and *component instances*. Each component instance c is instantiated from a particular *component type* C. A component type C is associated with a set of attributes K that form the *knowledge* (i.e. the state) of c. Each c is associated with a valuation of its knowledge – i.e. a function V_K that assigns each attribute $k \in K$ a particular value.

Definition 1 (Ensemble type):
An ensemble type E is a tuple (P, R, G, M, U, T), where:

- P is a set of ensemble parameters;
- R is a set of component roles in E;
- G is a set of sub-ensemble groups in E;
- M is a membership condition;
- U is a utility function;
- T is a task function.

Each component role $r \in R$ is associated with a function $r_{dom}(V_P)$ that for a given valuation of ensemble parameters V_P determines the component instances that may be selected for the role (i.e. the powerset $2^{r_{dom}(V_P)}$ is the domain for the role r). Each sub-ensemble group $g \in G$ is associated with function $gs_{dom}(V_P)$ that for a given valuation of ensemble parameters V_P yields a set of tuples (E_i, V_P^i). Each (E_i, V_P^i) – *ensemble instance template* – prescribes the ensemble type and parameters for instantiation of a potential sub-ensemble in the sub-ensemble group g. The membership condition (predicate) M determines the condition under which an ensemble instance based on E is valid. The predicate M is parameterized by V_P, the selection of component instances determined by $r_{dom}(V_P)$ for each role $r \in R$, and the set of sub-ensemble instances determined by $gs_{dom}(V_P)$ for each sub-ensemble group $g \in G$. Utility function U yields values upon which a total order $\leq$ is defined. Similarly to M, it is parameterized by V_P, selection of component instances to each role $r \in R$, and set of sub-ensemble instances for each sub-ensemble group $g \in G$. Task function T generates a set of tasks to be executed by instances of E (for given V_P) for V_P, selection of component instances to each role $r \in R$, and a set of sub-ensemble instances for each sub-ensemble group. Note that we intentionally do not provide a formal model for the tasks here because the nature of a task is not relevant for how ensembles are instantiated. The tasks are elaborated in Section IV in relation to the materialization of the semantics in the Scala language.

Definition 2 (Ensemble instance):
An *ensemble instance* e of ensemble type $E = (P, R, G, M, U, T)$ is a tuple (V_P^e, V_R^e, V_G^e), where:

- V_P^e is a function that assigns a value (of the appropriate type) to each parameter $p \in P$;

- V_R^e is a function that assigns a subset of component instances selected as the possible members of e of $r_{dom}(V_P^e)$ to each component role $r \in R$;
- V_G^e is a function that assigns a set of ensemble instances I_g to each sub-ensemble group $g \in G$.

Each ensemble instance $e_j \in I_g$ must comply (see below) with some (E_i, V_P^i) from the $gs_{dom}(V_P^e)$ associated with g. The projection from I_g to $gs_{dom}(V_P^e)$ does not have to be surjective (i.e. not all ensemble instance templates in $gs_{dom}(V_P^e)$ have to be actually instantiated).

Formally, for every $e_j = (V_P^j, V_R^j, V_G^j) \in I_g$ there exists an ensemble instance template $(E_i, V_P^i) \in gs_{dom}(V_P^e)$ such that e_j *complies* with the type (i.e. E_i is ensemble type of e_j and $V_P^j = V_P^i$).

An ensemble instance e is *valid* only if the following conditions are true:

- The membership condition is satisfied – i.e. $M(V_P^e, V_R^e, V_G^e)$ is true;
- All sub-ensemble instances are valid – i.e. $\forall g \in G, \forall e_s \in V_G^e(g)$: ($e_s$ is valid);
- An ensemble instance is not transitively its own child ensemble instance (i.e. there is no cycle in the ensemble instance hierarchy (tree));
- Any component instance that is a member of a child ensemble instance is also member of its parent ensemble instances higher in the ensemble hierarchy.

The *utility* of the ensemble instance e is $U_e = U(V_P^e, V_R^e, V_G^e)$. Note that by being parameterized by V_G^e, the utility function U can aggregate the utilities of sub-ensemble instances.

Definition 3 (Root ensemble instance):
An ensemble instance is called *root ensemble instance* if it is not a sub-ensemble of another ensemble instance. We denote A the set of all root ensemble instances.

Let AT_{dom} be the set of root ensemble instance templates (E_i, V_P^i). Each such template determines how to potentially instantiate exactly one root ensemble instance – i.e. for each root ensemble instance $e_j = (V_P^j, V_R^j, V_G^j) \in A$ there is exactly one ensemble instance template $(E_i, V_P^i) \in AT_{dom}$ such that e_j complies with this ensemble instance template.

For the sake of instantiation of ensembles, we also associate each tuple $(E_i, V_P^i) \in AT_{dom}$ with a component instance $c \in C$ responsible for instantiating the corresponding ensemble instance e_j. We call the component c the *initiator* of the ensemble instance e_j.

Definition 4 (Optimal instantiation of ensembles):
A *valid ensemble instance* $e = (V_P, V_R, V_G)$ of type E *is optimal* with respect to ensemble instance template (E, V_P) (that specifies how to instantiate the ensemble instance) if there is no other valid ensemble instance $e' = (V_P', V_R', V_G')$ that complies with the ensemble instance template (E, V_P) and $U_e < U_{e'}$.

Ensembles in a system are *optimally instantiated* (with respect to AT_{dom}) if for each ensemble instance template $(E_i, V_P^i) \in AT_{dom}$), one of the following conditions hold:

- There exists a corresponding $e_j = (V_P^j, V_R^j, V_G^j) \in A$ that complies with (E_i, V_P^i) and is optimal;
- There exists no valid $e_j = (V_P^j, V_R^j, V_G^j)$ that would comply with (E_i, V_P^i).

As mentioned above, we will use the terms "component" and "ensemble" to refer to a component instance and ensemble instance, respectively, and explicitly refer to their respective types only when needed.

C. Instantiation of ensembles

The SSA establishes ensembles to always achieve optimal instantiation of ensembles (as per Definition 4). The instantiation is partitioned per root ensemble instances. Since these are mutually independent, the instantiation can be seen as a set of (isolated) constrain optimization problems (COPs). For each root e the goal of the associated COP is identifying those component instances that should be the ensemble members throughout the ensemble hierarchy with e as the root. Technically, the COP is solved by the *initiator* component instance associated with e. In essence, the initiator gathers the knowledge needed to solve the COP, solves it and runs all the task defined by T functions of the ensemble types the ensemble instances comply with (bottom up throughout the ensemble hierarchy). Finally, the results of the tasks are propagated back to the member component instances of these ensembles. Since all the root ensemble instances are independent, all this happens in parallel across all initiators in the SSA.

III. Use case—RoboCup Rescue Simulation

In this Section, we describe the use case serving to exemplify TCOEL constructs and semantics and motivate our decisions in its design.

RoboCup Rescue Simulation[4] (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 rescue league, 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 city map which includes streets, intersections, and buildings, and of stationary and platoon agents. Buildings may collapse due to an 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 the platoon agents has specific strategic tasks to achieve and different capabilities. Fire Brigades are responsible for extinguishing

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

fires, Police Forces for removing blocking debris from the streets, and Ambulance Teams for rescuing humans by unburying them and carrying 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 being 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 teams at runtime of platoon agents in order to efficiently coordinate search and rescue operations. A team is formed as (potentially heterogeneous) group of agents, with each agent featuring a particular team-specific role in contributing to the strategic task(s) of the team. Agents are selected for a particular role based on several criteria (which can be also combined), in particular:

- Based on **agent type**. A team of Fire Brigades and Police Forces can enable the former to get information on a route to the fire that is cleared of blocking debris. This is reflected in the set of component roles of the ensemble definition (Section II.B).
- Based on **number of agents**. A team may require a certain number of Fire Brigades to cooperatively work on extinguishing the fire in a building. This is reflected in the membership condition of the ensemble definition (Section II.B).
- Based on a **soft optimization rule**. Among the Fire Brigades eligible for participating in a team, prefer those that are closer to the fire. This is reflected in the utility function of the ensemble definition (Section II.B).
- Based on **spatial proximity**. Platoon agents that are close-by can form a team in order to rendezvous and share updates regarding street blockages (by debris) via face-to-face communication. In TCOEL, this criterion is reflected in the `map` trait (Section IV.B).
- Based on **temporal proximity**. Only those buildings on fire are extinguished that can be reached before they are burnt out (the RCRS simulator assumes burnt out buildings need not be responded by Fire Brigades). In TCOEL, this criterion is reflected in the `map` trait (Section IV.B).
- Based on **estimated cooperation effort**. Upon detecting a fire, Fire Brigades can form a team composed of the (estimated) minimum number of Fire Brigades necessary to prevent the fire from spreading to nearby buildings. In TCOEL, this criterion is reflected in the `data prediction` trait (Section IV.B).
- 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 team which relies on regular rendezvous of agents to exchange data over close-range (i.e. face-to-face) communication is chosen over a team where all agents communicate via long-range radio. In TCOEL, this criterion is reflected in the `statistics` trait (Section IV.B).

IV. TCOEL AND TCOEF

In this Section, we articulate the criteria for ensemble formation employed for the use case in Section III via the TCOEL concepts. We also briefly describe a prototype implementation of TCOEL containing the engine forming the ensembles at runtime and trait library—*TCOEF framework* (middleware) available at http://github.com/d3scomp/tcoef.

We split the criteria for ensembles' formation (used in specification of ensemble types, see definitions in Section II.B) in two categories: (i) Based on the core concepts that are independent of a particular SSA domain, (ii) based on the concepts that depend on particular SSA domains; we group the latter into reusable *traits*. By trait, we denote a set of concepts that extend the core of TCOEL and can be used optionally; this is similar to object-oriented languages where a trait (sometimes called mixin), is typically a set of orthogonal methods that can be attached to a class to extend its behavior.

The concepts featured by a trait are specific to a particular SSA domain (e.g. connected mobility, emergency coordination, home automation, robotic swarm) 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 in applications that belong either to the connected mobility domain or to the domain of emergency coordination in a city. In contrast, criteria that do not depend on a particular SSA domain are universal in ensemble type specification, i.e. they typically make sense in any application domain. For example, the "type" of an agent is a possible criterion for ensemble formation in any application (irrespective of its domain).

Thus, an ensemble type in an SSA can be specified by using the domain-independent concepts and augmenting them with selected traits.

A. TCOEL, an example

In TCOEL we represent (specify) both component and ensemble types as Scala classes that extend the abstract classes `Component` and `Ensemble` respectively. Furthermore, we use the power of Scala to define new control structures (*constructs* for short) to express the operation steps of components and to declare membership condition and task computation of an ensemble. Technically, these constructs are realized as Scala functions with a "by-name" parameter [9].

We exemplify our approach on a part of the RCRS use case (Section III) illustrated in Fig. 1. Here the abstract classes `Component` and `Ensemble` are inherited by application-specific classes—lines 4, 41, 66, 81, 113. In particular, a `FireStation` component is expected to form an ensemble with `FireBrigade` components in order to extinguish the fire of a building and protect the surrounding ones. Similarly, there are components representing other RCRS agents (`AmbulanceTeam`, etc.). Due to space limits, Fig. 1 does not show how components are actually instantiated; for simplicity let us assume there exists a singleton of `FireStation` and *n* instances of `FireBrigade`. To achieve the required ensembles, the ensemble `FireCoordination` (line 66),

initiated by `FireStation`, introduces two sets of child ensembles—*`extinguishTeams`* and *`protectionTeams`* (lines 68-73). `FireCoordination` is a root ensemble (Section II.B). The members of these child ensembles cover disjoint locations on the city map. Members of an ensemble `ProtectionTeam` are selected from *`brigades`* (line 81, instances of `FireBrigade`) and are associated with a particular `fireLocation`; the selection is determined by the **membership** construct (lines 86-99) 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 building there burns down (otherwise, there is no reason to go there). Plus, their number has to be 2 or 3. In the **taskComputation** construct (lines 106-110) the selected *`brigades`* are assigned to specific fire locations. The ensemble type `ExtinguishTeam` is specified similarly.

Below we elaborate more on the TCOEL component and ensemble semantics.

Components. In support of self-adaptation, a component operates on a periodic basis by performing the classical MAPE-K loop [19]. This is done in the following four steps activated by TCOEF.

In *Monitoring*, the component senses data from its sensors and typically receives messages from the other components that are initiators of the ensembles it is a member of; as a follow-up it accordingly updates its knowledge (e.g., line 5), both local and mirror (the types of which are defined in Model). In TCOEL, this step is specified by the **sensing** construct (lines 8-13,44-46).

In *Analysis*, the component first determines the potential activities it can perform, given its state of knowledge. To achieve its conceptual autonomy and play its role in the current requirement of ensemble, the component's activities are controlled by its *behavior states* (BState, line 6). 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 behavior states at the same time, which corresponds to the ability to simultaneously fulfil several roles. The construct **constraints** (lines 15-19) serves to indicate (i) dependency of a behavior state on a particular value in component knowledge and (ii) conflicting states (e.g. moving and observing environment). To break ties in situation where different conflicting behavior states could be selected given the current valuation of knowledge, the **utility** construct provides a utility function, allotting weight to behavior states (lines 25-27).

In *Planning* initiation of ensembles takes place. This involves solving the constraint optimization problem (Section II.C) stemming from the ensemble type specification (membership condition, utility function) and from the current knowledge and utility value of the potential component members. This initiation is captured by the **ensembleResolution** construct (lines 48-52) employed in the component that becomes the initiator of the ensembles resulting from the resolution.

In *Execution* (the **actuation** construct—lines 20-23 and 54-63) the component performs the actuation and typically sends knowledge updates to the initiators of the ensembles it is a member of. If the component is an initiator of an ensemble (contains the construct **ensembleResolution**), then the final step in **actuation** is a signal for dismantling "its" ensemble(s) – this is completed once all their tasks are finished (specified in the **taskComputation** constructs).

As an aside, the periodic operation of components is not visible in Fig. 1, since component instances are activated by the RCRS simulator in its simulation steps.

Ensembles. The specification of an ensemble type is structured following the core concepts of ensemble formation as discussed in Section II.A. The selection of components, based on their type, is represented by the **role** construct (line 82). 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 condition (**membership** construct—lines 75-78 and 86-99) and the utility function (soft optimization rule defined by the **utility** construct—lines 101-104). For example, in the `ProtectionTeam` ensemble type, `membership` mandates that **utility** is computed as inversely proportional to the travel time needed for each selected member brigade to get to fireLocation. Strategic-level task computation takes the form of updating coordination-relevant knowledge of the ensemble's members (**taskComputation** construct—lines 106-110). The membership condition includes both the cardinality constraints on the number of components (agents) and the domain-dependent constraints pertaining to geographical proximity which exploit the concepts featured by the inherited traits (Section IV.B).

As to writing membership conditions in nested ensembles, there is a simple convention: Assuming the parent ensemble type P defines its membership condition MP, and a direct child ensemble type C defines its membership condition MC, then the actual membership condition of C is MP & MC. This reflects the rule that a member component of an instance of C has to be also a member component of an instance of P.

```
1.  class RescueScenario extends Model with RCRSConnectorTrait
2.    with Map2DTrait [MapNodeStatus] with StateSpaceTrait {
3.
4.  class FireBrigade(val entityID: EntityID) extends Component {
5.    var assignedFireLocation: …, waterLevel: …, location: …
6.    val Protecting, Refilling, Idle, Escaping = BState
7.
8.  sensing {
9.      sensed.messages.foreach {
10.       case (InitiatorToFireBrigade(receiverId, ..., fireLoc), _)
11.       // ...
12.   }
13. }
14.
15. constraints {
16.   (Escaping->(brigadeHealth<MINOR_INJURY_THRESHOLD)) &&
17.   (Refilling->(refillingAtRefuge || tankEmpty)) &&
18.    // ...
19. }
20. actuation {
21.   performAction()
22.   sendMessages()
23. }
24.
25. utility {
26.   if (Protecting) 1 else 0
27. }
28.
29. private def performAction(): Unit = state match {
30.   case Refilling if !refillingAtRefuge => moveTo(nearestRefuge)
31.   case Escaping if !regeneratingAtRefuge =>
32.                         moveTo(nearestRefuge)
```

component

```
33.     case Protecting =>
34.       if (inExtinguishingDistanceFromFire) extinguish()
35.       else moveTo(assignedBuildingOnFire)
36.     case _ => rest()
37.   }
38.   // ...
39. }
40.
41. class FireStation(val entityID: EntityID) extends Component {
42.   val fireCoordination = root(new FireCoordination(this))
43.
44.   sensing {
45.     processReceivedMessages()
46.   }
47.
48.   ensembleResolution {
49.     fireCoordination.initiate()
50.       //establishes a number of ProtectionTeam
51.       //and of ExtinguishTeam instances
52.   }
53.
54.   actuation {
55.     for (protectionTeam <-
56.                   fireCoordination.protectionTeams.selected)
57.       for (brigade <- protectionTeam.brigades.selected) {
58.         val message = InitiatorToFireBrigade(brigade.entityID,
59.           brigade.brigadeState, brigade.assignedFireLocation)
60.           agent.sendSpeak(time, Constants.TO_AGENTS,
61.                         Message.encode(message))
62.       // ... likewise for ExtinguishTeam
63.     }
64.   }
65.
66. class FireCoordination(fireStation: FireStation) extends Ensemble {
67. // fireStation ∈ P in Definition 1
68.   val extinguishTeams = // g ∈ G in Definition 1
69.       ensembles(buildingsOnFire.map(fireLocation => new
70.                   ExtinguishTeam(fireLocation)))
71.   val protectionTeams = // g ∈ G in Definition 1
72.       ensembles(buildingsOnFire.map(fireLocation => new
73.                   ProtectionTeam(fireLocation)))
74.
75.   membership { // M in Definition 1
76.     extinguishTeams.map(_.brigades)
77.           ++ protectionTeams.map(_.brigades)).allDisjoint
78.   }
79. }
80.
81. class ProtectionTeam(fireLocation: EntityID) extends Ensemble {
82.   val brigades = role("brigades",components.select[FireBrigade])
        // r ∈ R in Definition 1
83.   val routesToFireLocation = map.shortestPath.to(fireLocation)
84.   val firePredictor = statespace(burnModel(fireLocation), time,
85.                                 fireLocation.status.burnoutStage)
86.   membership {   // M in Definition 1
87.     brigades.all(brigade =>
88.      (brigade.state == Idle) ||
89.      (brigade.state == Protecting) &&
90.       sameLocations(brigade.assignedFireLocation)
91.     ) &&
92.     brigades.all(brigade =>
93.      routesToFireLocation.timeFrom(mapPosition(brigade)) match {
94.         case None => false
95.         case Some(travelTime) =>
96.                   firePredictor.valueAt(travelTime) < 0.9
97.       }) &&
98.     brigades.cardinality >= 2 && brigades.cardinality <= 3
99.   }
100.
101.    utility {   // U in Definition 1
102.      brigades.sum(brigade => travelTimeToUtility(
103.          routesToFireLocation.timeFrom(mapPosition(brigade))))
104.    }
105.
106.    taskComputation {  // T in Definition 1
107.      for (brigade <- brigades.selectedMembers) {
108.        brigade.assignedFireLocation = Some(fireLocation)
109.      }
110.    }
111. }
112.
113. class ExtinguishTeam(fireLocation:EntityID) extends Ensemble
114.       { /* ... */ }
115. }
```

Fig. 1. Fragment of ROBOCUP Rescue scenario in TCOEL

B. *Expressivity through domain-dependent traits*

In a membership condition it is not easy to express real-world constraints 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 articulation of such conditions strongly depends on the particular SSA domain. To build support for all the possible types of conditions to the core of the specification language is not only impractical, since the language would be quite complex and hard to learn, but even impossible, as all the possible SSA domains cannot be foreseen. Plus, a single application typically would not need all the condition types. Indeed, all the examples in Section III specify conditions over spatial distances and estimated travel times. However, while the RCRS use case prescribes conditions over estimates of fire spreading/burning speed, a connected mobility system would, e.g., prescribe conditions over estimates of traffic congestions and vehicle speeds, etc. Therefore, in our approach, all these domain-dependent condition types are to be designed as reusable traits to be picked up and used in compliance with the needs of a specific application whenever possible.

In the rest of the Section, we overview three traits that are already available in TCOEF. As with the core concepts, we illustrate two of them on the RCRS example in Fig. 1. On the lines 1-2 there are the *map trait (*Map2DTrait) and *data prediciton trait* (StateSpaceTrait); furthermore, there is also a specific RCRSConnectorTrait connecting the language run-time (TCOEF) with the Rescue simulator (it creates particular agents and processes messages from/to the simulator—not explained here). Technically, TCOEL traits are developed as Scala traits.

Map Trait. This trait serves to capture spatial-temporal relations between the components to be included in an ensemble. The typical use is, e.g., to select the entities that are close to each other or close to a particular location in terms of travel time. An example is on lines 83, 93, 102-103. Line 82 computes the shortest routes to fireLocation (via Dijkstra's algorithm). Lines 93 and 102-103 query the computed travelTime needed for a `FireBrigade` to reach fireLocation.

Data prediction Trait. This trait serves to form ensembles based on the prediction of a data value in SSA. Such predictions can rely either on state-space models that characterize data evolution based on physical processes [20] or on machine learning models that capture patterns and trends in historical data. Examples of application of this trait include (i) the team of "agents within travel time less than the estimated time until building B is burnt out" and (ii) the team of "agents within travel time less than the estimated time-to-survive of victim V".

In TCOEL, the former is captured by lines 84-85 and 96. Lines 84-85 initialize a predictor of how quickly a particular building (at `fireLocation`) burns out based on its burning model represented as an ordinary differential equation (ODE); such a

model is assumed to be associated with each building. The initial conditions for ODE are the current time and current burnoutStage of the building.

The predictor uses a solver (i.e. a numerical integrator) to solve ODE for a specified point of time (line 96). By combining Map2DTrait and StateSpaceTrait, lines 93-96, it is ensured that "All agents selected for the team have to be able to reach the building (i.e. travelTime is not None) while the burnoutStage of the building and the travelTime the agent needs to reach it has to be below 0.9 (the building is not burnt out yet)".

Statistics Trait. This trait offers the possibility to construct an ensemble 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 [21]. Due to space constraints, we do not demonstrate this on the example in Fig. 1 since this would necessitate including other parts of the scenario; instead, we give an illustration below.

Consider an agent team that heavily relies on radio communication, so that it can be formed only if "The expected probability of packet delivery over the radio is 90 percent or more, evaluated over the last hour with a confidence of 95%" (technically, such parameters can be set in the RCRS simulator). This would be captured in TCOEL 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 the probability of being true less or equal to 0.9 with significance level $\alpha = 0.05$.

C. TCOEF

TCOEF is a runtime framework (middleware in the form of a Scala library) which executes the individual steps of component's MAPE-K loop and takes care of resolving ensembles. Furthermore, it provides a basic extensible library of reusable traits (Section IV B).

Internally, TCOEF translates the component instances and ensemble types to a constraint optimization problem (COP) that describes optimal instantiation of ensembles as defined in Definition 4. The COP which encodes instantiation of ensembles based on their respective ensemble instance templates (E_i, V_P^i) as generated by functions gs_{dom} (Definition 1). This instantiation starts from root ensembles instances (Definition 3 and line 42 in Fig. 1) and goes recursively over the sub-ensemble groups (i.e. G in Definition 1). The membership of a component in an ensemble instance and sub-ensemble instances are encoded as Boolean variables. Membership conditions are encoded as hard constraints and utility functions as soft constraints. The rules that determine if an ensemble is valid (Definition 2) and determine ensemble hierarchies are also encoded as hard constraints.

A solution found by the constraint solver corresponds to valid instantiation of root ensemble instances. The optimal solution found by the solver then implies the optimal instantiation of ensemble hierarchy (or hierarchies).

V. Evaluation

As stated in Sect. I, with TCOEL and TCOEF we aim at providing (1) a language that allows for expressive yet reusable specification of complex components and ensembles and (2) a runtime framework for resolving such complex ensembles. We therefore evaluated both the level of reuse that TCOEL enables and the scalability of TCOEF in resolving complex hierarchical ensembles.

A. Level of reuse enabled by TCOEL

To compare the development effort when using TCOEL against employing a classical approach, we have developed two versions of the RCRS use case—one exploiting the TCOEF framework (as described in Sect. IV.A and IV.B) and another one employing a single-purpose backtracking algorithm (both are available at http://github.com/d3scomp/tcoef). The TCOEL-based implementation is formed by a main class and several additional classes defining auxiliary functions. The main class with the additional classes amounts to roughly 400 LOC in total. It further uses three reusable traits—the 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 560 LOC. Thus, in total, approximately a half of the code is the business logic of the use case and another half the reusable traits.

The implementation without TCOEL is a single Scala class (with nested classes) with domain-dependent concepts embedded in its code (those corresponding to the traits above). Since it cannot take advantage of the solver used in TCOEL, it implements a simple search heuristic to figure out the agent teams (corresponding to ensembles in TCOEL). Overall, it amounts to 1065 LOC—i.e. roughly two times the size of business logic in the TCOEL-based solution. Note that the size of the former is just slightly less than the overall size of the latter—a small penalty for taking advantage of the reusable traits. Though these measurements are difficult to generalize, they suggest that the reusability of traits can bring an advantage already when a trait can be reused across two use cases. Also, TCOEL removes the need to provide code that figures out the composition of agent teams. The process of resolving the agent teams typically leads to developing some heuristic likely combined with backtracking. Such technique leads intrinsically to difficult development and debugging. As such it requires significant elaboration effort, even though, due to recursion, the size of actual code can be relatively small.

B. Scalability of ensemble resolution process in TCOEF

To evaluate the scalability of TCOEF, we benchmarked the ensemble resolution process employed in the RCRS use case (Sect. III). This process works in the following way: There is a custom RCRS agent realized as a base class for every component instance in the scenario (firefighters, fire station). The class provides implementation for sensing, actuating and message sending, and more importantly, allows a component to act as an ensemble initiator. To this end, the class invokes the

ensemble resolution process (namely the Choco solver[5]) and finally executes the tasks of the instantiated ensembles.

To evaluate the scalability of ensemble resolution, we ran the RCRS use case on a small map (58 roads and 37 buildings) with 1 `FireStation` and variable number of fires and `FireBrigades` for a predefined duration (100 simulation steps). At each step, the COP solver was invoked. We measured the duration of each step; the median duration for different number of fires and fire brigades is depicted in Figure 2. We repeated the experiment with a larger map ("Kobe" featuring 1602 roads and 757 buildings); median durations are depicted in Figure 3. In both experiments we observed that, due to the exponential complexity of ensemble resolution as COP, ensemble resolution with 5 fires and 5 `FireBrigades` or more takes more than 20 secs to complete, whereas "easier" cases were mostly solved within 1 sec. We further comment on the scalability limitations of our approach in the following section.

VI. Discussion of Limitations

As mentioned in Section IV.C, for resolving ensembles TCOEF currently internally uses a centralized COP solver. We could generally speed up the ensemble resolution by incorporating a more optimized solver, however this does not provide an answer to the intrinsic complexity of the problem. In this respect, there are two promising methods to address the problem: (a) employ multiple root ensemble instances since their resolution processes are independent and can thus be parallelized, or (b) employ a non-exhaustive search of the state space by stopping the solver after some fixed time. This should be combined with preconditioning the problem model in such a way 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 in which they contribute to the system utility and by discarding those having a smaller contribution to the utility than a given threshold. In the RCRS use case, it means sorting `FireBrigades` by the distance to the fire and removing those being too far. Nevertheless, this may result in finding a non-optimal solution, or even in not finding a solution at all (though it generally exists); our initial experiments suggest that such relatively simple means can significantly help in addressing the inherent complexity problem without compromising the average quality of the system too much [22].

Regarding the effectiveness of the proposed TCOEL language, we have provided an initial assessment of the level of reuse that can be achieved via using TCOEL compared to using plain Scala. We acknowledge that a more complete evaluation of the effectiveness and comprehensibility of TCOEL requires one or more user studies (such as[23]). Nevertheless, the result of such user studies will depend on the familiarity of the users with Scala.

VII. Related Work

As mentioned in Section I, component ensembles have been

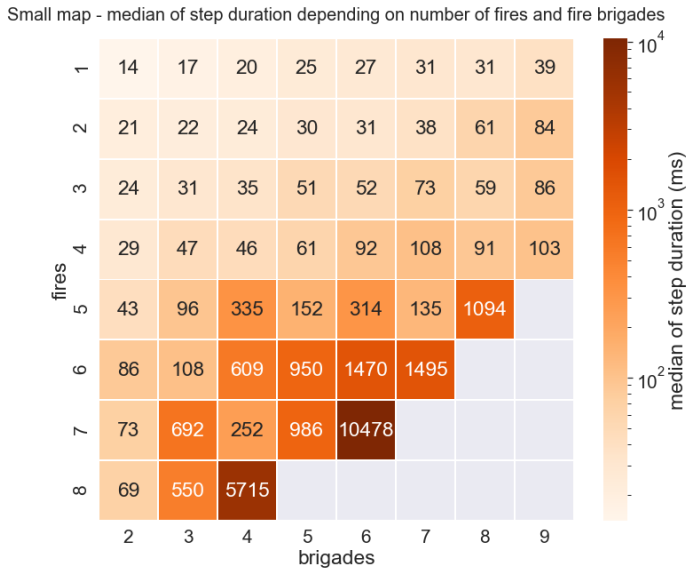

Figure 2: Ensemble resolution duration for different number of fires and fire brigades on a small map in the RCRS. Durations more than 20 secs are depicted in grey.

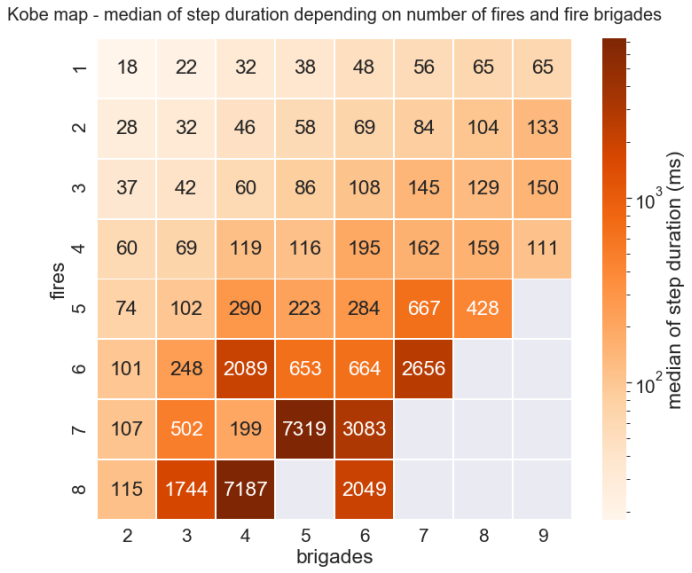

Figure 3: Ensemble resolution duration for different number of fires and fire brigades on the Kobe map in the RCRS. Durations more than 20 secs are depicted in grey.

proposed to be realized for SSA. The original idea is based on SCEL [6], [13], which proposes the specification of ensembles implicitly (via attributes of component knowledge). Until now, several frameworks based on the concept of ensembles have already been developed and applied. Helena [7], JRESP (http://jresp.sourceforge.net) and DEECo [1] are examples of them. Specifically in the context of DEECo, we have we have proposed a method for designing component ensembles [8], [24] and discussed the typical software engineering assumptions violated in software intensive CPS, which is

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

mitigated by the ensemble [25]. Furthermore, in support of ensembles, we have studied several aspects of underlying networks, including their dynamically modified topologies [17], [26]. The benefits of ensembles in DEECo-based applications were also illustrated in [1].

The aforementioned frameworks differ in the semantic of ensemble formation: In Helena, a component instance explicitly indicates which of the ensembles it belongs to. In contrast, a JRESP ensemble is an abstraction capturing the groups of cooperating components, dynamically determined by means of their attribute-based communication. Hence, ensembles are realized implicitly in JRESP. (In a similar vein, Ab^{a}CuS [27], though not an explicitly ensemble-based framework, employs opportunistic and attribute-based communication among components). Finally, DEECo ensembles are specified explicitly. Their formation employs a "greedy" strategy—a component is a member of each of the ensemble instance, the membership condition of which it satisfies, and, at the same time this condition is solely based on the component's current knowledge and cannot express, e.g., any reflection property (like cardinality). This is in contrast to the much more expressive power of TCOEL in membership conditions thanks to the concepts like trait, cardinality, and utility function.

Team formation is intensively studied in the RCRS community. Many approaches [28], based on different criteria, have been already proposed and implemented within the project, however their realizations are low-level and technically single-purpose, not reusable elsewhere. To address the issue, the Agent Development Framework (ADF, https://github.com/RCRS-ADF/RCRS-ADF) [29] was recently proposed and has started being employed in support of an easy reuse of code among multiple teams participating in RCRS. ADF defines a set of Java interfaces/base classes for individual aspects of RCRS applications, e.g., tactics for platoons and centers, path planning, and communication. This way it provides a basic structure for creating agents. In general, ADF also uses traits, yet quite low-level (plus, at the time of writing, lacking almost any documentation) and reusable only within the RCRS applications. Nevertheless, ADF lacks the ability to define teams as first class architectural concepts.

Another framework targeting reuse in the RCRS applications is RMASBench [30]—a testbed for multi-agent coordination algorithms. The main focus of RMASBench is benchmarking of distributed constraints optimization problem (DCOP) algorithms used within this framework to resolve team formation of agents. For this purpose, the API of RMASBench adds an extra channel that allows an extensive exchange of messages between agents. The DCOP algorithms in RMASBench are modelled as reusable first-class entities and when a problem is to be solved by reusing one of these algorithms, just a scoring function needs to be implemented.

Multi-paradigm domain-specific and modeling languages have recently emerged [31], similar to TCOEL in the aspect of specification. Multi-paradigm modeling targets a combination of multiple abstractions, formalisms and (meta-) modeling levels into a single approach [32]. An example of such a language is QAML [33], a quantitative analysis modeling language constructed by following the multi-paradigm modeling approach, i.e., for individual paradigm concerns it reuses existing modeling languages (e.g., AADL for architecture, MathML for math expressions), composing them together into a single tool. Compared to TCOEL, however, QAML is not designed as extensible nor, notably, supports a notion similar to ensemble. Another example can be found in [34], where the authors describe the process and challenges they faced during designing a multi-paradigm-based software architecture description language—but again as in the case of QAML, their language is not designed to be extensible and there is no notion similar to ensemble.

VIII. Conclusion

In this paper, we have focused on the problem of specifying and forming complex, real-life dynamic groups of cooperating components in Smart System Applications (SSA), which manifest as heterogeneous landscape of various applications of Cyber-Physical Systems, Internet of Things, and Smart Sensing Systems. We have provided an extensible specification and also implementation language TCOEL that allows specifying component ensembles in an intuitive, reusable and, at the same time, semantically rich way. Built as an internal DSL in Scala, it thus serves as an implementation language as well. Moreover, since Scala complies to JVM, it easily interoperates with Java and other JVM- based languages.

A conservative way to design and implement an SSA use case would be to introduce a dedicated DSL from scratch; such an approach obviously lacks reusability when multiple use cases featuring partly common concepts are to be considered. On the contrary, the strength of our TCOEL is that it provides a compositional way to design a DSL for a particular SSA featuring ensembles. It does it by taking advantage of the overlap that exists in SSA domains in terms of paradigms they exploit in modeling the reality. These paradigms are reflected as reusable traits in the library being a part of TCOEF—the software framework which implements TCOEL. The viability of TCOEL and the traits was demonstrated on a use case inspired by RCRS—a multi-agent solution to disaster response scenarios where dynamic teams of agents/components are an inherent necessity. This use case and, primarily, TCOEL together with TCOEF are publicly available as an open source library at http://github.com/d3scomp/tcoef.

Acknowledgement

The research leading to these results has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783221. Also, this work was partially supported by the Czech Science Foundation project 20-24814J.

References

[1] T. Bures, F. Plasil, M. Kit, P. Tuma, and N. Hoch, "Software Abstractions for Component Interaction in the Internet of Things," *Computer*, vol. 49, no. 12, pp. 50–59, Dec. 2016, doi: 10.1109/MC.2016.377.

[2] M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds., *Software engineering for collective autonomic systems: the ASCENS approach*. Cham: Springer, 2015.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," in *Proceedings of MCC'12, Helsinki, Finland*, 2012, pp. 13–16, doi: 10.1145/2342509.2342513.

[4] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog Computing: A Platform for Internet of Things and Analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, Springer, Cham, 2014, pp. 169–186.

[5] G. Fairbanks, "Architectural Hoisting," *IEEE Softw.*, vol. 31, no. 4, Jul. 2014, doi: http://doi.ieeecomputersociety.org/10.1109/MS.2014.82.

[6] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A Formal Approach to Autonomic Systems Programming: The SCEL Language," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 7:1–7:29, Jul. 2014, doi: 10.1145/2619998.

[7] R. Hennicker and A. Klarl, "Foundations for Ensemble Modeling – The Helena Approach," in *Specification, Algebra, and Software*, S. Iida, J. Meseguer, and K. Ogata, Eds. Springer, 2014, pp. 359–381.

[8] T. Bureš, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "The Invariant Refinement Method," in *Software Engineering for Collective Autonomic Systems*, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer International Publishing, 2015, pp. 405–428.

[9] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, Third Edition*. Artima Press, 2016.

[10] C. Artho, K. Havelund, R. Kumar, and Y. Yamagata, "Domain-specific languages with scala," in *International Conference on Formal Engineering Methods*, 2015, pp. 1–16.

[11] A. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *Proc. of the First International Conference on Multi-Agent Systems*, 1995, pp. 312–319.

[12] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECo – an Ensemble-Based Component System," in *Proc. of CBSE'13*, 2013, pp. 81–90.

[13] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese, "A Language-Based Approach to Autonomic Computing," in *Formal Methods for Components and Objects*, vol. 7542, B. Beckert, F. Damiani, FrankS. de Boer, and MarcelloM. Bonsangue, Eds. Springer, 2013, pp. 25–48.

[14] C. Kroiß and T. Bureš, "Logic-based modeling of information transfer in cyber–physical multi-agent systems," *Future Gener. Comput. Syst.*, vol. 56, pp. 124–139, Mar. 2016, doi: 10.1016/j.future.2015.09.013.

[15] R. A. Ali *et al.*, "DEECo Computational Model – I," Technical report no. No. D3S-TR-2013-01, , Dep. of Distributed and Dependable Systems, Charles University in Prague, 2013.

[16] J. Barnat, N. Beneš, T. Bureš, I. Černá, J. Keznikl, and F. Plášil, "Towards Verification of Ensemble-Based Component Systems," in *Formal Aspects of Component Software*, J. L. Fiadeiro, Z. Liu, and J. Xue, Eds. Springer International Publishing, 2014, pp. 41–60.

[17] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "Gossiping Components for Cyber-Physical Systems," in *Software Architecture*, P. Avgeriou and U. Zdun, Eds. Springer International Publishing, 2014, pp. 250–266.

[18] T. Bures, P. Hnetynka, F. Krijt, V. Matena, and F. Plasil, "Smart Coordination of Autonomic Component Ensembles in the Context of Ad-Hoc Communication," in *Proceedings of ISOLA 2016, Corfu, Greece*, 2016, vol. 9952, pp. 642–656, doi: 10.1007/978-3-319-47166-2_45.

[19] J. Kephart and D. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[20] R. Al Ali, T. Bures, I. Gerostathopoulos, J. Keznikl, and F. Plasil, "Architecture Adaptation Based on Belief Inaccuracy Estimation," in *Proceedings of WICSA 2014, Sydney, Australia*, 2014, pp. 87–90, doi: 10.1109/WICSA.2014.20.

[21] T. Bures, P. Hnetynka, J. Kofron, R. Al Ali, and D. Skoda, "Statistical Approach to Architecture Modes in Smart Cyber Physical Systems," in *Proceedings of WICSA 2016, Venice, Italy*, 2016, pp. 168–177, doi: 10.1109/WICSA.2016.33.

[22] F. Krijt, Z. Jiracek, T. Bures, P. Hnetynka, and F. Plasil, "Automated Dynamic Formation of Component Ensembles," in *Proceedings of Modelsward 2017, Porto, Portugal*, 2017, pp. 561–568, doi: 10.5220/0006273705610568.

[23] G. Dubochet, "Computer Code as a Medium for Human Communication: Are Programming Languages Improving?," in *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group*, 2009.

[24] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch, "Design of ensemble-based component systems by invariant refinement," in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, 2013, pp. 91–100.

[25] I. Gerostathopoulos, J. Keznikl, T. Bures, M. Kit, and F. Plasil, "Software Engineering for Software-Intensive Cyber-Physical Systems," in *Proc. of CPSData workshop in INFORMATIK'14. To Appear.*, 2014.

[26] M. Kit, F. Plasil, V. Matena, T. Bures, and O. Kovac, "Employing Domain Knowledge for Optimizing Component Communication," in *Proceedings of CBSE 2015, Montreal, Canada*, 2015, pp. 59–64, doi: 10.1145/2737166.2737172.

[27] Y. A. Alrahman, R. D. Nicola, and M. Loreti, "On the Power of Attribute-Based Communication," in *Proceedings of FORTE 2016, Heraklion, Crete, Greece*, 2016, vol. 9688, pp. 1–18, doi: 10.1007/978-3-319-39570-8_1.

[28] J. Parker, E. Nunes, J. Godoy, and M. Gini, "Exploiting Spatial Locality and Heterogeneity of Agents for Search and Rescue Teamwork," *J. Field Robot.*, vol. 33, no. 7, pp. 877–900, Oct. 2016, doi: 10.1002/rob.21601.

[29] K. Takayanagi *et al.*, "Implementation of NAITO-ADF and its Team Design NAITO-Rescue 2015," in *Proc. of RoboCup Intl. Symp. 2015, Hefei, China*, 2015.

[30] F. Maffioletti, R. Reffato, A. Farinelli, A. Kleiner, S. Ramchurn, and B. Shi, "RMASBench: A Benchmarking System for Multi-agent Coordination in Urban Search and Rescue," in *Proceedings of AAMAS 2013, St. Paul, MN, USA*, 2013, pp. 1383–1384.

[31] A. Horst and B. Rumpe, "Towards Compositional Domain Specific Languages.," in *Proceedings of MPM 2013, Miami, USA*, 2013, pp. 1–5.

[32] P. J. Mosterman and H. Vangheluwe, "Guest Editorial: Special Issue on Computer Automated Multi-paradigm Modeling," *ACM Trans. Model. Comput. Simul.*, vol. 12, no. 4, pp. 249–255, Oct. 2002, doi: 10.1145/643120.643121.

[33] D. Blouin, E. Senn, K. Roussel, and O. Zendra, "QAML: a multi-paradigm DSML for quantitative analysis of embedded system architecture models," in *Proceedings of MPM '12, Innsbruck, Austria*, 2012, pp. 37–42, doi: 10.1145/2508443.2508450.

[34] D. Balasubramanian, T. Levendovszky, A. Dubey, and G. Karsai, "Taming Multi-Paradigm Integration in a Software Architecture Description Language," in *Proceedings of MPM 2014, Valencia, Spain*, 2014, pp. 67–76.