Automated Dynamic Formation of Component Ensembles
Taking Advantage of Component Cooperation Locality

Filip Krijt, Zbynek Jiracek, Tomas Bures, Petr Hnetynka and Frantisek Plasil
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{krijt, jiracek, bures, hnetynka, plasil}@d3s.mff.cuni.cz

Keywords: Distributed coordination, architectural adaptation, ensemble-based component system, autonomic systems.

Abstract: Smart cyber-physical systems (sCPS) is a growing research field focused on scenarios where a set of autonomous software-hardware entities (components) is cooperating via network communication to achieve a type of swarm or cloud intelligence. Typically the components' cooperation is designed at a low level of abstraction and their behavior validated via simulations. As a remedy, a declarative language capable of specifying high-level component ensembles has been proposed in recent work. By capturing component functionality and the cooperation constraints, a specification serves both for generating platform-specific implementation and as a model@run.time to support self-adaption via dynamic formation of ensembles. However, for a particular specification, multiple possible architectural configurations exist with various impact on the system. Given their typically large number, we select the best one via an SMT solver. In this paper, we show that scalability of such approach can be supported by exploiting the effect of locality in component cooperation and by hoisting specific domain knowledge to the level of architecture.

1. INTRODUCTION

Recent proliferation of cheap yet powerful hardware components on the market and focus on initiatives such as IoT have led to an increased interest in Smart Cyber-physical Systems (sCPS). Typically distributed (and often decentralized), these systems consist of autonomous hardware/software entities (components) that are extended with a network connection to achieve a form of collective intelligence based on opportunistic cooperation, enabling them to better fulfil the desired goal(s). An efficient description of this cooperation proves to be a challenge due to the variability and sheer number of situations the system should respond to. This results in the need for novel software engineering concepts and practices.

In this paper, we focus mainly on architecting self-organizing sCPS, building on the paradigm of autonomic component ensembles (Wirsing et al. 2011) introduced within the ASCENS project (EU FP7 FET – http://ascens-ist.eu/). These dynamically formed component cooperation groups support many of the desired sCPS properties, e.g., dynamicity of system architecture and autonomous component operation. In particular, we utilize the intelligent ensemble (often referred to as ensemble for short further on) concept (Bures et al. 2015), which provides rich structural constraints and optimization constructs.

By taking advantage of the associated domain-specific language for ensemble specification, we apply a model-driven approach, employing the ensemble specification model both at compile-time for generating platform-specific implementation, and as a *model@run.time* (Morin et al. 2009) description of the required architecture. To enable general resolution of the architecture based on this specification model, as well as deal with the large number of possible architectural configurations, the problem is viewed as a SAT problem. An SMT solver is periodically used to select the best dynamic architecture configuration according to the specification model and the current context, i.e., the state of the system and its environment. The resulting configuration itself is also a *model@runtime*, representing the actual architecture of the system. However, finding the best configuration is naturally challenging in terms of scalability. The goal of this paper is therefore to show how the specification model can be extended with additional concepts that take advantage of application-specific domain knowledge to improve scalability.

The structure of the paper is as follows. Sect. 2 introduces a running example, and shows how to model it using the ensemble concepts. In Sect. 3 we discuss the scalability limitation of the approach and a conceptual way to address it. Sect. 4 describes the language concepts we have introduced to address scalability. Sect. 5 presents a discussion together with a short overview of related work. Finally, Sect. 6 concludes the paper.

2. APPLYING ENSEMBLES

2.1 Running Example

To illustrate the ensemble concepts we use a simplified example that can nevertheless be used to demonstrate many of the properties of ensembles. We model a railroad emergency response service in which trains move along a one-dimensional space representing parallel railroad tracks, with each train having a dedicated track. The trains' fuel tank is not of very high quality and occasionally breaks and leaks fuel. In such an event, a group consisting of several (e.g., 2 to 3) emergency repair vehicles and a refueling truck must be dispatched to the train to resume its operation. These vehicles travel on roads running parallel to the tracks. We assume dense placement of railroad crossings, and that the trains are equipped with reliable sensors and brakes, ensuring that there is no risk of collision with vehicles crossing the tracks.

The goal of the system as a whole is to minimize the amount of time the trains are not operable. Assuming the time needed for an actual repair and refueling is negligible, the main performance goal is to get repairers and refueling trucks to the damaged trains fast. As all repairers and trucks are equivalent in terms of speed, this translates to selecting trucks that are near the inoperable train to repair it. However, care must be taken not to get too greedy, as simply assigning the closest vehicles could easily result in another train being left without repairs for a long time. For the sake of simplicity, we assume that there are enough vehicles to assist all trains simultaneously – thus every train will eventually be repaired. The described problem can be seen as forming emergency groups that are "good" according to some global metric, such as how far the involved emergency vehicles must travel in total. An example of such "good" assignment is in Figure 1.

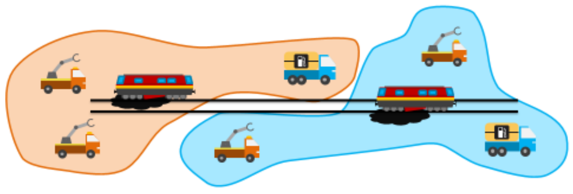

Figure 1: Example of two well-chosen emergency groups.

2.2 Ensembles and Components

Thanks to their ability to describe temporary collaboration groups that are dynamically formed and dissolved based on system's state and outer context, ensembles are very suitable for modelling problems like the one outlined above. Modelling with ensembles recognizes two main first-class concepts: *components* and *ensembles*.

Individual entities in the system are modelled as *components*, entities encapsulating state in the form of *component knowledge*, and behavior, represented by periodically executed or event-triggered *processes*. The components are designed to be independent, and thus not allowed to directly communicate with each other. Instead, the cooperation aspect is realized via *ensembles*, groups of components that are dynamically formed based on a declarative *membership* specification provided by the architect. Ensembles facilitate cooperation by means of a *knowledge exchange*, allowing the developer to specify data transfer. An important point is that the ensembles are not created directly, but are instead formed by the ensemble runtime framework based on the provided model/specification. Both the formation and the knowledge exchange occur periodically, allowing for dynamic system architecture. No overlap of components is permitted between the ensembles.

In the case of our scenario, the actors in the system, i.e., trains, repairers, and refueling trucks can be seen as components, with knowledge related mostly to their geographical position and processes handling sensing and actuating (movement), while ensembles can be used to model the emergency groups created to assist the damaged trains.

2.3 Modelling with Ensembles

In Figure 2, we show a particular description of our example system using the declarative Ensemble Definition Language (EDL) (Bures et al. 2015), allowing us to directly state our requirements on both shape and optimality, with the runtime framework taking care of forming the appropriate architecture. Because the understanding of the ensemble concepts is critical for the main contribution of this paper, we use this EDL description to show the semantics. As the EDL draft shown in (Bures et al. 2015) has since been backed by an implementation, and the language refined, there are subtle differences between the form presented here and the original syntax.

To be able to specify an ensemble type, we first define the types it depends on – in our case, the ensembles deal with trains, repairers and refueling trucks. On lines 1, 6 and 11, we use the *data contract* construct, which is essentially an interface over knowledge, to declare the required fields (such as position) for each type of entity in the system. Each component can satisfy multiple data contracts.

Next, we specify the ensemble type used for cooperation when repairing the trains, starting on line 17. To identify the ensemble, we declare *id* to be a component of the type Train – essentially saying that instances of this ensemble type cannot be created without being associated with a unique Train instance, which can be seen as a sort of coordinator of the instance. Apart from being useful for restricting the domain of all possible ensemble instances, the id serves an additional purpose of representing *ensemble knowledge*, a shared knowledge store for the ensemble – with the train component being the sole writer, thus avoiding any problems with synchronization.

The main section of the ensemble specification is the ensemble membership starting on line 19 and consisting of three sections. First, we define the shape of the ensemble by declaring the ensemble *roles* that the components can participate in – in this case, we say that all ensembles of this type contain 2 to 3 Repairer components (repairers role, line 21) and exactly one Truck component (refueler role, line 22). Each role declares its required data contract, and the set of roles a component can participate in is limited to those whose data contract it satisfies. In order for the ensemble instance to exist, the cardinalities and the required data contract of each role must be satisfied, essentially serving as a structural constraint on the shape of the ensemble.

Next, we place semantic constraints, represented by the *constraint* expression – an arbitrary logical expression based on boolean and integer knowledge fields (due to the limitations of the underlying solver) of ensemble members. In our scenario the only constraint is shown on line 24, saying that the train in question must have its needsHelp flag set to true, i.e., the ensemble instance is not formed unless the corresponding train breaks down and publishes its desire to get repaired.

The third part of the membership definition is the *fitness* function, specified with a numeric expression. The fitness function is not a constraint, but instead provides the optimizing aspect of the ensemble membership. If the ensemble formation framework must decide between forming two variants of a single instance, the one with higher fitness value will be chosen. More precisely, ensemble instances will be created in such a way as to maximize the sum of their fitness values – essentially performing global optimization. The fitness should therefore be carefully chosen by the system architect to capture the intuitive system utility (i.e., how good the system is at fulfilling its goals) or performance of the system as close as possible. In our example, the fitness function can be seen on line 25 and is calculated as a sum of inverse of distance from the train for all rescuers in the ensemble. Finally, knowledge exchange is specified, creating a data flow between the members and completing the specification. The EDL supports simple inline assignments among the ensemble members, or deferring to an external implementation in platform-specific language, e.g., Java.

```
1  data contract Train
2    position : int // position along the x axis
(railroad kilometers)
3    needsHelp : bool
4  end
5
6  data contract Repairer
7    position : int
8    target : int
9  end
10
11 data contract Truck
12   position : int
13   target : int
14   fuel : int
15 end
16
17 ensemble RepairTeam
18   id train : Train
19   membership
20     roles
21       repairers [2..3] : Repairer
22       refueler : Truck
23     constraints
24       constraint train.needsHelp
25     fitness sum repairers 1 / ((it.position -
train.position) * (it.position -
train.position)) + 1
26   knowledge exchange
27       refueler.target = train.id
28       repairers.target = train.id
```

Figure 2: EDL Specification of the example.

2.4 Ensembles MDE Workflow

The declarative EDL description is one half of the ensembles support, the other half being the runtime framework providing the ensemble formation capabilities. As of now, the intelligent ensemble concepts are supported by a framework implementation utilizing the Eclipse Modelling Framework (EMF), XText, the Java implementation of the DEECo component model (Bures et al. 2013), and the Microsoft Z3 SMT solver. Figure 3 captures the overall architecture and workflow of the framework. While being integrated with the DEECo Java implementation, the ensemble formation framework can support any environment capable of providing the general *knowledge container*, which provides access to knowledge in the form of data contracts, capturing the system context.

The process starts with the EDL file, described by an Ecore metamodel and compiled via the EMF pipeline. The document object model (DOM) representing a particular EDL file is a high-level model of the system specification and is used to generate the required classes, namely those representing the data contracts and ensemble instances, as well as a thin wrapper of the ensemble formation mechanism. The ensemble formation mechanism itself is the most complicated and important part of the framework, and is encapsulated in an *ensemble factory*. In the current implementation, the factory is realized by a mapping to the Z3 solver. This mapping combines the current knowledge data gained from the knowledge container and the ensemble specification and generates a problem description in the form of logical formulas for the Z3. An important point here is that the ensemble specification is provided in the form of the EDL DOM, so the model is used both for compile-time code generation and runtime specification representation – essentially being used in the spirit of the *models@run.time* approach (Morin et al. 2009), lending itself well to modelling adaptive systems. After the Z3 is run, the factory creates ensemble instances indicated by the results of the SMT solver. These instances – together another *model@run.time* representing the desired system architecture – are then used to drive ensemble formation and the knowledge exchange, influencing the data in the knowledge container and completing the loop.

3. ADDRESSING SCALABILITY

3.1 Scalability Problems

It can be seen that the system description both addresses the example scenario and is complete in terms of supporting all possible system configurations (i.e., assignments of supporting vehicles to damaged trains). Modelling the system in this way has many advantages – it (i) lessens the mental overhead of the system architect, (ii) makes it easier to reason about the properties of the system being built, and (iii) provides a clean separation of concerns by moving the heavy-weight implementation of the cooperation and group formation logic from components to the framework.

However, this neat high-level declarative prescription also presents problems. The chief of these is the lack of scalability. Due to its level of abstraction, the runtime framework cannot take

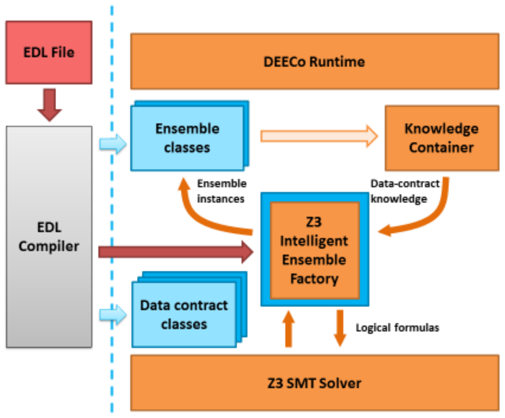

Figure 3: Framework architecture, with specification in red, generated code in blue and platform classes in orange.

advantage of any problem-specific heuristic (e.g., assigning the closest repairers), as this cannot be inferred from the description. Instead, the framework must consider all possible configurations and pick the best one based on the valuation of the fitness function. As the number of possible configurations is exponential with respect to the number of components in the system, this approach clearly cannot scale.

3.2 Enabling Performance Optimizations

Instead of trying to directly improve the performance of the framework and the underlying solver – which would not address the exponential nature of the problem itself – we can turn the problem around and find a way to inject the missing domain knowledge into the general assignment mechanism, allowing it to transform the problem to a variant of the problem that is easier to solve (i.e., digestible for the solver in terms of size and complexity) without compromising the utility of the system too much. In order to do that safely and decide what form the domain knowledge should take, we make some important assumptions about the class of sCPS systems we want to support.

Assumption 1: *Typically, many of the possible configurations satisfying ensemble's membership constraints are undesirable due to low fitness and consequently leading to subpar utility of the system.* In terms of our example, such configurations are easily seen, as they are going against our intuitive understanding of the fitness function – for example a configuration assigning the furthest repairers and truck to a train. While the percentage of these undesirable configurations highly depends on the problem and the threshold, it can be generally expected to grow with the scale of the system – for example, the larger the area our railroad service covers, the more repairers it has, and the more possibilities for assigning repairers that are beyond a reasonable operating radius there are. In fact, with scale, the undesirable configurations can also become more damaging to the system utility – it would take the furthest repairers much longer to reach the train.

Assumption 2: *A well-designed system is influenced by ensemble formation only in terms of system utility, not safety.* As sCPS are inherently distributed and only able to communicate with other parts of the system via unreliable channels, such as MANETs and other wireless networks, safety guarantees cannot be built on top of communication. Instead, the components must offer core safety guarantees by design, so that a component is still capable of secure operation even when cut off from other parts of the system. As the ensembles are essentially a communication abstraction, they cannot be used to guarantee safety. In our example, this can be seen in the design of the trains – regardless of any network problems, the train is equipped with trusted sensors and brakes to avoid any catastrophic scenario. If no ensembles are ever formed, a damaged train will hamper the system indefinitely, but will not endanger any lives.

Disregarding Configurations. Based on the assumptions outlined above, a useful conclusion can be reached: *It is possible to completely disregard the undesirable configurations when deciding what ensembles to form.* While these configurations are valid per se, they represent a situation when the system is not performing well, and should never be needed under normal circumstances; meaning that situations when selecting such a configuration would be the right thing to do are also highly improbable. The only moment these configurations should be realized is when there are no other options for the system to utilize – and at such a time, the difference between forming badly performing ensembles and forming no ensembles at all becomes negligible. As ensemble formation only impacts utility and not safety, dropping these configurations influences the system utility minimally, and safety not at all.

4. PROPOSED SOLUTION

4.1 Importance of Locality

Before we can drop undesirable configurations to make the ensemble formation more scalable, we must define what form this additional domain knowledge should take, and introduce a suitable concept to the EDL. In essence, we need a way to allow the architect to exactly specify when a configuration is to be deemed undesirable, and eliminate these before handing the problem description over to the solver.

In Sect. 3.2, we have intuitively rated the configuration based on domain-specific properties of individual components, e.g., saying that picking repairers that are too far from the damaged train does not make sense. More generally, this can be interpreted as a distance metric expressed in terms of the local knowledge of an individual component and domain data of the ensemble instance in question, and there is a correlation between the distance metric and the impact on instance fitness (and by extension overall system utility) if this component would join. Assuming the existence of such a metric is fairly realistic – many sCPS applications tend to be very large and deployed on physical entities. It is therefore necessary to partition the system into manageable parts, often by taking advantage of the physical locality of the components both in design and execution. This is especially seen when dealing with geographical position, but can also take form of network distance or a similar property.

Once we have a metric with suitable properties, we can assume that we can separate suitable and unsuitable components with a preference function based on their computed distance valuation. To enable the configuration filtering, we must allow the developer to specify this preference in the EDL.

4.2 EDL Filtering Concepts

Figure 4 shows the modified EDL ensemble type code, which we use to introduce the two concepts that enable the identification of the undesirable configurations. The code is similar to the original, except for the *where* and *limit* clauses added to the role specification in the ensemble type.

The purpose of the *where* clause (line 33) is to limit the component selection for a particular role only to the components satisfying the corresponding condition – unless the where clause is used, all components of the declared role type (i.e., satisfying its data contract) are considered suitable. The where clause can therefore be seen as a threshold-based filter. Syntactically, it is a logical expression with a single restriction – it can only refer to the ensemble knowledge, represented by the defined id name, and the knowledge of the component being considered, represented by the **it** keyword. This is in sharp contrast with the general constraint clause, which can refer to any role and knowledge, thus requiring the ensemble structure to be fully decided before its evaluation. This restriction allows us to evaluate the condition for each candidate component separately and potentially discard it *before* the problem is presented to the solver, making the problem smaller.

While the *where* clause and the hard true/false threshold it defines are suitable for many problems, sometimes it is more natural to define the filtering based on some suitability order, e.g., one defined by the distance metric. We have therefore also proposed the *limit* clause, which limits the selection to a specified number of best components, ordered by a given expression. It should be noted that unlike where, limit is not implemented in the current version of the runtime, but can be easily supported. The usage of the limit clause can be seen on the line 34. The end effect of the *limit* clause is similar to the

```
29 ensemble RepairTeam
30   id train : Train
31   membership
32     roles
33       repairers [2..3] : Repairer where
Abs(it.position - train.position) < 100
34       refueler : Truck limit 10 orderby
Abs(it.position - train.position)
35     constraints
36       constraint train.needsHelp
37     fitness sum repairers 1 / ((it.position -
train.position) * (it.position -
train.position)) + 1
38   knowledge exchange
39       refueler.target = train.id
40       repairers.target = train.id
```

Figure 4: Specification enhanced with filtering concepts.

where clause, but instead of dropping the undesirable configurations entirely, the runtime will consider them only when no better options are available. Its effect is also more predictable, as it will always select at most the specified number of components, whereas the *where* clause can in some situations select no components (e.g., every component is further than the threshold), or all of them (e.g., all components are clustered nearby).

The details of the concepts' implementation are rather technical and out of scope of this paper. In short, before the ensemble factory transforms the ensemble specification into formulas, it creates a set of possible components for each role in an ensemble. Initially, this set consists of components with the matching data contract. The filtering concepts are evaluated after this type matching phase and further restrict the set before passing it on to the Z3 solver – the solver itself is therefore unaware of the filtering and deals directly with a reduced problem.

Unlike other intelligent ensemble concepts, the *where* and *limit* clauses cannot be seen as enhancing the expressivity of the ensembles; instead, they are a way to inject the missing pieces of domain knowledge into ensemble specification and enable the solver to perform domain-specific optimizations without losing its generality. Being application-specific, the exact values used for the filtering must be decided by the architect based on his experience, simulation runs, or logging of the actual application.

5. DISCUSSION AND RELATED WORK

Discussion: The degree to which the filtering concepts are effective highly depends on the used filtering expression. Ideally, the expression should be chosen in such a way that the components that would result in the highest fitness valuation of an ensemble will always be preferred. This requires the existence of a distance metric as described in Sect. 4.1. Alternatively, we may utilize a weaker form of a distance metric that only approximates the fitness function. In this case, we may cut off some configurations that would result in high fitness, with the possible loss of utility depending on the density of such "good" configurations among those that are discarded. If no suitable metric can be found, the usefulness of the filtering is severely limited.

Additionally, if several possible ensemble instances had similar sets of filtered suitable components, the solver would be much more constrained than intended, possibly resulting in a markedly reduced system fitness. In the worst case, some instances that would otherwise form with reasonable fitness would not exist due to their most suitable components being needed elsewhere. This would also happen without filtering, but in that case, the instance may have other candidate components to use instead. This effect will be less pronounced in systems with strong locality and clear partitioning.

Finally, it is worth noting that there are situations when not all possible ensembles can be formed (e.g., if there are not enough repair vehicles for all trains, or due to filtering), and that a specific ensemble instance (e.g., for a very distant train) is repeatedly ignored in favor of other ensembles with higher fitness. This lack of progress may be undesirable, and while it can be addressed by using a variable representing priority of the instance as part of fitness functions and filtering, a part of future work can be to consider a more systematic way to deal with it.

The two assumptions from Sect. 3.2 limit where the filtering concepts can be used, but not unreasonably. Most sCPS are naturally manifested in the physical world (e.g., smart cities, smart mobility, wearables) and thus exhibit both the low density of reasonable configurations (due to geographical position) and the need for communication-independent safety (especially if they interact with humans). Of course, if the system requires cooperation of all components or is based on reliable communication, the assumptions will not hold. However, such system is typically of centralized nature and will not need the benefits of ensembles.

Related work: Several frameworks based on the concept of ensembles have already been realized and the paradigm has been successfully applied to sCPS case studies, such as (Hoch et al. 2015). One of these ensemble-based frameworks is Helena (Hennicker & Klarl 2014). It offers the same core concepts, i.e., components with roles and ensembles, however it focuses primarily on communication between components and does not explicitly capture the architecture of an application. Also, it does not provide means for optimizing during ensemble formation. The same is also true for JRESP (http://jresp.sourceforge.net), another ensemble-based framework. Ab^{a}CuS (Alrahman et al. 2016b) – a Java-based implementation of AbC (Alrahman et al. 2016a) – is not directly an ensemble-based framework but it shares several key concepts (components with roles and opportunistic communication among them described via a set of rules and conditions). As the frameworks above, it also does not allow explicitly capturing architecture and expressing optimizations for establishing communication links.

Conceptually very close to ensemble-based systems are multi-agent systems (MAS) with communication among agents via coalition formation. There are many approaches for coalition formation and optimizations, e.g., (Michalak et al. 2010; Rahwan et al. 2012; Sandholm et al. 1999), however they assume fully connected networks and thus are not suitable for sCPS (which typically have to operate in a loosely coupled environment).

6. CONCLUSION

In this paper we have presented the ensemble concepts and their model-driven support in the intelligent ensembles framework, and highlighted the scalability problem. To address it, we have introduced new concepts to the EDL language that allow the system architect to provide optimization hints to the framework. We have also described our assumptions and the reasoning behind the concepts, particularly the importance of utilizing locality.

Even though the approach shows promise, it still requires a further evaluation. In particular, it is necessary to verify that the assumptions from Sect. 3.2 are indeed as common to sCPS scenarios as expected, and to measure how the use of the filtering concepts impacts the properties of the system.

In terms of future work, it may be worth investigating whether the domain knowledge can be injected into the solver in a more elegant or more powerful form than the one presented here – or possibly even inferred from the DSL via language analysis. A systematic way to enforce progress, mentioned in Sect. 5, remains to be explored as well.

On the whole however, we believe that this approach is promising and allows for easier design and implementation of common sCPS scenarios.

AKNOWLEDGEMENTS

This work was partially supported by Charles University Grant Agency project No. 390615, and partially supported by Charles University institutional funding SVV-2016-260331.

REFERENCES

Alrahman, Y.A., Nicola, R.D. & Loreti, M., 2016a. On the Power of Attribute-Based Communication. In *Proceedings of FORTE 2016, Heraklion, Crete, Greece*. LNCS. Springer, pp. 1–18.

Alrahman, Y.A., Nicola, R.D. & Loreti, M., 2016b. Programming of CAS Systems by Relying on Attribute-Based Communication. In *Proceedings of ISOLA 2016, Corfu, Greece*. LNCS. Springer, pp. 539–553.

Bures, T. et al., 2013. DEECo: An ensemble-based component system. In *Proceedings of CBSE 2013, Vancouver, Canada*. ACM, pp. 81–90.

Bures, T. et al., 2015. Towards Intelligent Ensembles. In *Proceedings of ECSAW 2015, Dubrovnik/Cavcat, Croatia*. ACM, pp. 1–4.

Hennicker, R. & Klarl, A., 2014. Foundations for Ensemble Modeling – The Helena Approach. In S. Iida, J. Meseguer, & K. Ogata, eds. *Specification, Algebra, and Software*. LNCS. Springer, pp. 359–381.

Hoch, N. et al., 2015. The E-mobility Case Study. In M. Wirsing et al., eds. *Software Engineering for Collective Autonomic Systems*. LNCS. Springer, pp. 513–533.

Michalak, T. et al., 2010. A Distributed Algorithm for Anytime Coalition Structure Generation. In *Proceedings of AAMAS 2010, Toronto, Canada*. pp. 1007–1014.

Morin, B. et al., 2009. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10), pp.44–51.

Rahwan, T. et al., 2012. Anytime coalition structure generation in multi-agent systems with positive or negative externalities. *Artificial Intelligence*, 186, pp.95–122.

Sandholm, T. et al., 1999. Coalition structure generation with worst case guarantees. *Artificial Intelligence*, 111(1–2), pp.209–238.

Wirsing, M. et al., 2011. ASCENS: Engineering Autonomic Service-Component Ensembles. In B. Beckert et al., eds. *Proceedings of FMCO 2011 (Revised Selected Papers), Turin, Italy*. LNCS. Springer, pp. 1–24.