Intelligent Ensembles – a Declarative Group Description Language and Java Framework

Filip Krijt[1], Zbynek Jiracek[1], Tomas Bures[1], Petr Hnetynka[1], Ilias Gerostathopoulos[2]

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

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

***Abstract*— Smart cyber-physical systems (sCPS) is a growing research field focused on scenarios such as smart cities or smart mobility, where autonomous components are deployed in a physical environment, and are expected to cooperate with one another, as well as with humans. As these systems typically operate in a highly uncertain and dynamically changing environment, being able to cooperate and adapt in groups to cope with various (possibly unanticipated) situations becomes a crucial and challenging task. In this artifact, we respond to this challenge by presenting the Intelligent Ensembles framework, consisting of a high-level declarative language for describing dynamic cooperation groups, and a Java runtime library for automatically forming groups that best satisfy the given specification. The framework provides dynamic architecture adaptation (i.e., forming groups of components and exchanging data between them) based on the state of components and situation in their environment. Further, the framework can be used as a first step of a group-wise adaptation (i.e., identifying components that are to negotiate and coordinate in an adaptation). The framework is built on top of the Z3 SMT solver and the Eclipse Modelling Framework.**

Keywords—smart cyber-physical systems; distributed cooperation; adaptive architecture; ensemble-based component system; group-wise adaptation; autonomic systems

I. INTRODUCTION

Recently, a lot of focus has been given to initiatives such as Internet of Things (IoT), smart cities, smart mobility, wearables and many more fields that have the same goal—making the environment that humans live in more intelligent, responsive, and helpful. The common denominator of these initiatives is the presence of physical devices (sensors, actuators, or complex devices such as smartphones) in the environment, and the fact that these devices are connected to one another over a network, which enables them to provide better or more complex services than they would be able to do individually.

The research field of smart Cyber-Physical Systems (sCPS) is focused on such types of systems—those comprised of individual components with both software and physical representation, capable of autonomous operation, but also of networked cooperation. There are many challenges in building such systems due to the number (and often conflicts) of the required quality attributes. In this artifact we focus on enabling cooperation via complex component group formation without breaking the autonomicity of the components, and the loose-coupling of the system as a whole. At the same time, we aim to support dynamicity in terms of addition and removal of components, and the ability to adapt to sensed changes in the environment or the system itself—a key property for sCPS, which often have to cope with highly uncertain and dynamic environment.

To this end, we build on the concept of autonomous component ensembles [1], as introduced within the ASCENS project (EU FP7 FET – http://ascens-ist.eu/). Ensembles are dynamic groups of components periodically formed at runtime based on the current state of the components (their knowledge). Components themselves are not allowed any kind of direct interaction—instead, interaction is the responsibility of the ensemble, and takes the form of data transfers (knowledge exchange in ensemble terminology) between components in an ensemble. Ensemble-based approaches have been shown to provide properties desirable in sCPS, such as resilience, open-endedness and architecture flexibility in face of high dynamism.

In our previous work, we have proposed the concept of Intelligent Ensembles, which allows for optimizing the process of member selection when forming ensembles, and for specifying arbitrary ensemble structure. The artifact described in this paper wraps this concept into a tool-based framework. The framework consists of (i) language editing tools for specifying complex groups of components in a custom DSL, and (ii) a runtime library capable of parsing the specification and forming groups adhering to this specification.

Note that while it is anticipated that groups formed by the framework will be realized in the system as ensembles, this is in no way mandatory—in fact, the framework itself should be viewed as a general tool for describing complex groups on a high level of abstraction, and then using an SMT solver for selecting the best possible assignment of components to these groups. Its applicability is thus not limited to ensemble-based systems, or even sCPS—the framework is suitable for any situation where complex, adaptive, prescriptive grouping of components is desirable. Therefore, where suitable, we use the general term group instead of ensemble in the rest of the text.

Since the groups are always formed at runtime based on the current state of a system, they are dynamic and able to

adapt to changes in the environment. As the group formation dictates also the connections between the components in the system, the architecture of the system as a whole is adaptive, dynamically resolved at runtime. Moreover, the formed groups themselves can also be used as a starting point for further adaptation among their members—with the framework able to identify what is to be adapted based on a declarative description. For example, one can imagine a high-level adaptation rule that prescribes that "all members of firefighters scouting groups have to switch to indoor positioning system from GPS when entering a building".

The structure of the paper is as follows. In Section II, we show several scenarios where the artifact would be helpful to developers and summarize the challenges common to the scenarios. Section III presents an overview of the framework including the domain-specific language we have devised for describing groups of components, as well as the technical architecture. A summary of the artifact structure is presented in Section IV, followed by Section V detailing the framework from the perspective of the user. A discussion of the limitations and future work is provided in Section VI, together with an overview of related work. The paper is concluded with a short summary of contributions of this artifact.

II. MOTIVATION

To better illustrate the kind of groups we are interested in, we provide several examples with focus on a smart domain that has been rapidly gaining traction for the past few years—smart mobility. We assume a system consisting both of mobile entities, such as cars, as well as static entities, such as gas stations, traffic lights and parking lots. To enable smart coordination, each entity is equipped with a mobile network device enabling communication both with a central server, and directly amongst the components. Many examples of hard-to-form groups can be found in this domain:

- *Emergency response*—Suppose a car accident happens. Depending on the severity of the event, various parties should be alerted and vehicles dispatched—an ambulance, one or more police cars, towing service vehicle, etc. Normally, reporting an accident requires a report call, so the event must be witnessed or the driver must be conscious. With a smart car however, the car itself (unless utterly demolished) can detect and assess the situation (i.e., state of the passengers, risk of fire, mobility of the car, damage costs, etc.) via sensors, set the corresponding severity level, and then form a group with the most suitable vehicles that are required to respond based on their position and the severity of the event. Consider also that assigning the closest response vehicles might not be the best strategy for the system as a whole, as these vehicles might be more needed elsewhere. To optimize for this case, groups should be decided globally.
- *Smart parking*—In this scenario we assume an integrated system of cars and smart parking lots, each equipped with an embedded device and a network connection. Cars desire to park near their destinations, and should thus consider grouping with parking lots close to them. At the same time, the number of assigned cars to one parking lot must not exceed its capacity. Additionally, more constraining requirements might be placed here as well, such as prioritizing cars that have been looking for a parking place longer, or cars with higher priority level, or only considering a certain number of cars that are closest to a lot.
- *Autonomous road trains*—A lot of cargo is transported via trucks, often travelling at different speeds, overtaking one another and forming a transportation bottleneck. With smart mobility, it is possible to form groups of trucks traveling in the same direction, with the first truck in a group setting the speed and the others following it. Such road train group could also include and coordinate with passing cars, letting emergency vehicles pass, etc.
- Many additional scenarios in smart mobility can be addressed by dynamic prescription-based groups—traffic light preference can be done by grouping an emergency vehicles with all traffic lights along its path. Similarly, cars waiting at red light could form a group and accelerate at the same time when given the green light.

A. Engineering and Coordination Challenges

When dealing with loosely-coupled systems comprised of autonomous components, such as those outlined in the previous section, forming non-trivial, structured groups is a difficult and important task, since the assignment of components to groups effectively dictates the system's structural architecture. Without a framework capable of understanding high-level group prescriptions and using them to efficiently form groups at runtime, developers of such systems are limited to creating tailored solutions to the group formation process. Such solutions would have to either explore all possible configurations, or use some kind of application-specific heuristic or approximation. Such a process is of course possible, but also tedious and error-prone, especially when dealing with multiple complex group types. Also, it demotes the reuse of the group formation logic.

In all the previous scenarios, a developer tasked with implementing such dynamic constrained groups would face several engineering challenges, namely:

- *Adaptive architecture*—Since the groups are typically dynamic and cannot be a part of the static structural architecture, the implementation must be adaptive in terms of components (cars, etc.) in the system. This entails creating an adaptation mechanism enabling pre-existing groups to change, or reform the groups when certain situations arise.
- *Separation of concerns*—As the group formation mechanism is essentially a part of dynamic architecture, good software engineering dictates that its implementation be separated from the actual business logic, on the grounds of having different responsibilities.

- *Resolving structural and semantic constraints*—In many cases of group formation, the group has members of various types, possibly with a limit on number of members, or even constraints on their state. The implementation would either have to deal with these cases uniformly, or use similar code for each group type, leading to duplicities.
- *Optimization*—Often, it is not enough to form any groups satisfying certain constraints, but instead form the best groups possible. For example, assigning police vehicles that are geographically the closest in case of emergency, or assigning parking slots to cars that have spent the longest time trying to find a place to park.

The Intelligent Ensembles framework was designed from the ground up to help developers tackle these challenges. It offers a high-level declarative language for group specification, constructs for placing both structural and semantic constraints, as well for selecting the best possible group formation. As the groups are described in the specification language, formed by the framework, and connected with the rest of the application via well-defined abstractions, the formation mechanism is completely separated from the business logic of the application.

III. The Framework

A. Ensemble Definition Language

Before going into how the framework itself works, we first show the Ensemble Definition Language (EDL), which is used to specify what groups should be present in the system. Such specification is the main input provided by the framework's user. To better enable the understanding of the concepts used for modelling the specification of the target system, we show an EDL file that models the Smart parking scenario outlined in Section II and contains all the major features of the EDL. We assume that each car requests a parking place at most some distance from its target location, and that each car is also assigned a priority value representing how fast it needs to park. We further assume that the situation can be modelled using discrete coordinates, such as the regular streets in United States cities, making it easy to denote the position of entities for the solver, and that each parking lot has at most 20 slots.

An EDL document must start with a package declaration in a familiar Java format, enabling fully qualified identification of the types declared in the document. There are two concepts in the EDL that represent a type declaration, similar to classes in traditional programming languages—data contract and ensemble type declaration. *Data contract* declaration can be seen on lines 2 and 10 and serves essentially as an interface over component knowledge. However, unlike traditional interfaces, a data contract can only prescribe data fields, not methods or any kind of interaction paradigm. A component therefore satisfies (or implements) a data contract if it has all the required fields. In our case, there are two types of components in the system—

```
1  package SeamsDemo
2  data contract Car
3    priority : int
4    targetX : int
5    targetY : int
6    driveToX : int
7    driveToY : int
8  end
9
10 data contract ParkingLot
11   locationX : int
12   locationY : int
13   capacity : int
14 end
15
16 ensemble ParkingAssignment
17   id parking : ParkingLot
18   membership
19     roles
20       cars [0..20] : Car where (
(Abs(it.targetX - parking.locationX) +
Abs(it.targetY - parking.locationY)) < 10)
21     constraints
22       constraint sum cars 1 <=
parking.capacity
23     fitness sum cars it.priority
24   knowledge exchange
25       cars.driveToX = parking.locationX
26       cars.driveToY = parking.locationY
```

Figure 1. An example of an EDL specification.

cars and parking lots—so we declare a new data contract for each of them.

The other type declaration available is the *ensemble* type declaration, seen on line 16. With ensembles being a realization of a group of components, the ensemble declaration defines how a specific type of group is shaped, what kind of components can participate in it, and what semantic constraints must be enforced for the group to exist. In our running example, we use the ensemble to model the assignment of cars to parking lots, each parking lot having a group (ensemble) of its own containing all cars currently expected to park there.

The ensemble type declaration consists of several sections. Each ensemble type declaration must contain an *id* definition. The purpose of the id is to uniquely identify each instance of this ensemble type. Indirectly, it also restricts the domain of the possible instances, with its size corresponding to the size of the id type domain. As we want to form one ensemble per parking lot, we use the components with ParkingLot data contract to identify ensembles of this type.

The type of the id can be either an integer, or a data contract. In the case of integer identification, the number of instances is not limited by the id, and the created instances will be numbered sequentially starting from 0. In this mode, only the other properties of the ensemble type (such as the required structure) will restrict the domain of instances. Alternately, if the id type is a data contract, the domain of instances is restricted by how many components satisfying the contract are present in the system, as an ensemble instance must be associated with a unique instance of the data contract. Additionally, when used with a data contract, the ensemble id

also serves as a shared storage for all members of the ensemble instance—realizing the concept of ensemble knowledge. To avoid synchronization problems, the storage is expected to be read-only to ensemble members, with the ensemble id being the sole writer.

The *membership* section starting on line 18 groups together features that influence how components in the systems are selected to form ensembles of this type. First selection constraint is structural—following the *roles* keyword on line 19 is a list of roles in the ensemble that a component can participate in. Roles are analogous to class fields in traditional languages. In this case, we have a single role, named cars, consisting of all cars assigned to a specific parking lot. A role declaration must consist of the name of the role, multiplicity, and the required type. The type of the role must be a data contract, and only components satisfying the data contract are allowed to participate in the role.

In addition to the structural and type conformance, EDL also allows specification of semantic constraints, such as saying that the number of assigned cars—specified using the *sum* expression aggregating the value 1 for each car in the ensemble—must not exceed the parking lot capacity. Constraints can be written in the *constraints* section starting on line 21. Each constraint is prefixed with a *constraint* keyword. A constraint is a logical expression over data visible in the ensemble, i.e., the knowledge accessible through the data contracts of the individual roles, as well as the ensemble knowledge represented by the id of the ensemble. As the current implementation of the constraint concept depends on an SMT solver, some limitations as to what field types can be used in the expression; this is discussed in Section VI.B.

In [2] we have argued for the need for additional language constructs that further restrict the set of components suitable for a specific role. One such construct that is fully realized in the current implementation is the *where* clause seen on line 20, which limits the component selection only to those satisfying a given constraint, for example allowing us to disregard parking lots that are further from the car's destination than a certain limit. Syntactically, the *where* clause must follow a role type declaration and consists of a single logical expression, similar to constraints. It is evaluated individually for each component that has passed the type validation step (corresponding data contract), with the aim of filtering out components that would result in unreasonable configurations. As such, it cannot depend on arbitrary knowledge available to such ensemble—its structure is not yet known. Instead, *where* is limited to the shared ensemble knowledge (i.e., the knowledge of the ensemble id), and the knowledge of the component under consideration, represented by the *it* keyword.

Finally, EDL allows its users to define a measure of how well an ensemble instance is suited to fulfil the ensemble's goal. This measure is defined following the *fitness* keyword on line 23. The fitness is an expression that must be evaluable to an integer value, having similar limitations as the constraints do w.r.t. its data fields. The purpose of the fitness clause is to enable the framework to prefer some configurations of an ensemble over others—if the framework has to choose between forming two instances of the same ensemble, it will prefer the one with higher fitness. In fact, the framework in its current implementation will optimize globally, forming ensemble instances that have the highest total sum of fitness values. In our case, we specify fitness as a sum of priorities of the cars in the ensemble, indicating that cars with higher priority should be preferred.

Finally, the ensemble contains a knowledge exchange section that can be used to model data transfer interaction between the members of the ensemble. EDL supports two modes of knowledge exchange specification. For simple scenarios, the data transfer can be directly written in the form of C-style assignment statements. This is sufficient in the running example, as we simply assign the location of the chosen parking lot to all the cars in the ensemble. Alternatively, knowledge exchange can be specified as external, meaning it will be defined in the platform-specific language, in this case Java. The framework then generates an additional Java class with a knowledge exchange method stub to be filled by the developer.

B. *High-level Architecture Overview*

In this section, we describe the technical architecture of the Intelligent Ensembles framework. The framework is built on top of a Java-based technology stack, consisting of the Eclipse Modelling Framework and the DSL development tools such as XText and XPand. As such, it is tightly integrated with Java and Eclipse. Additionally, the current implementation depends on Microsoft's Z3 SMT solver [3] for the ensemble formation strategy. Out of the box, the framework offers good integration with the jDEECo framework, a Java implementation of DEECo—an ensemble-based component system [4]—but is also designed to enable easy binding to other Java environments and frameworks.

The high-level architecture of the framework, as well as the typical workflow, can be seen on Figure 2. The process starts with an EDL specification file describing all types of groups present in the system, such as the one described in Section III.A. An important point here is that the specification does not explicitly state which ensembles should be formed, but only what constraints they should satisfy. The actual architecture of the system is thus up to the framework to

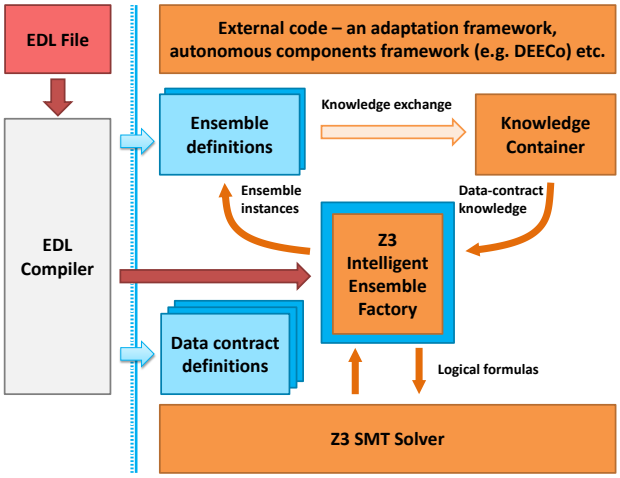

Figure 2. Framework high-level architecture.

decide, making the system dynamic and able to respond to changes.

The input EDL file is processed by the EDL compiler, which is provided in the form of an Eclipse plugin. The compiler parses the file into an Ecore model and then generates Java classes corresponding to the ensemble types defined in this model. All ensemble classes share a common interface providing a knowledge exchange method, which is invoked either by the external framework (e.g., jDEECo), or by user code.

The most complicated and important part of the framework is the ensemble formation mechanism itself. The formation strategy is encapsulated in the Ensemble Factory component, whose purpose is to take the Ecore model (representing specification) and the current component knowledge, and then use the components available in the system to build the best ensembles that satisfy the specification. The specification takes the form of the same Ecore model that is gained from the EDL file and used for code generation. This means that the specification is easily mutable at runtime, essentially being an application of the Models@run.time [5] approach and lending itself well to self-adaptation. The current version of the Ensemble Factory is built on top of the Microsoft Z3 SMT solver, forming ensembles by (i) translating the knowledge data and the specification model into logical formulas, (ii) feeding them to the solver, and (iii) interpreting the resulting valuation of variables as a specific architectural configuration.

C. Reusability

Despite being developed with jDEECo integration in mind, the Intelligent Ensembles framework was always designed as a separate entity, having only a minimal common interface with the goal of easy reuse. In fact, the framework has only one critical dependency in the form of the *KnowledgeContainer* interface, representing a data storage of components in the system—how exactly the container manages the instances is not important, as long as the invariants of the interface are maintained. This abstraction allows the framework to be easily integrated with many environments. As storing the knowledge in the form of data class instances is anticipated to be a common scenario, the framework is shipped with an implementation of such a knowledge container. If needed, a different ensemble formation strategy (e.g., based on another solver) can be easily introduced via a new Ensemble Factory implementation.

IV. Artifact Structure

The artifact accompanying this paper can be found either at http://d3s.mff.cuni.cz/software/deeco/files/seams-2017.zip or at http://dx.doi.org/10.4230/DARTS.3.1.6. The artifact comprises of a single archive containing both the source code of the framework, as well as precompiled eclipse plugins and an eclipse project containing the implementation of the example discussed in Section III.A. Due to the platform requirements of the current implementation and the licensing of the Windows OS, we are unable to provide a complete virtual machine image. Instead, the archive provides a modified Eclipse instance containing all the necessary configuration out of the box. This instance can be used on any Windows 10 x64 machine.

V. User's Perspective

A. Current Features

Faced with a problem that can be solved by applying the Intelligent Ensembles framework, an architect needs to take several steps. First, they should analyze the problem and identify the actors in the system to be realized as components, and specify their data contracts. Next, they need to design the groups representing assignments or cooperation, and decide how exactly the groups are structured, i.e., what components participate in it and how, what are the semantic constraints of the group, and which groups should be preferred. Together, these design decisions form the description of the group, realized as ensemble types in the EDL code.

The data contracts and ensemble types can be written in the EDL editor, and then translated into Java code via the automatically invoked EDL compiler. The resulting Java classes can then easily be used in the user code, as described further on.

B. Working with the EDL editor

The Intelligent Ensembles framework is intended to be used in the context of the Eclipse platform, with the compiler, UI and the language representation all provided as Eclipse plugins. The EDL editor currently supports highlighting, syntax validation, and a simple type validation that checks signatures of functions, compatibility of assignments and other type-based restrictions.

C. Using the Framework with General Java Code

In order to invoke the group formation logic from Java, only a few steps are needed—a concise example of the code is shown in Figure 3. First, the application code must create a new instance of a Factory class (line 27), generated by the EDL compiler and prefixed with the source EDL file name. Next, the state of the components must be provided via a knowledge container. Unless a special kind of component data handling is needed (e.g., support for data rollback), the provided *DataclassKnowledgeContainer* implementation can be used. After the container is created (line 28), component data must be stored in the container (line 29) for each component in the system. A simple *createInstances* method call on the factory (line 30) then creates the groups specified in the EDL file, returned as a collection of the EnsembleInstance interface implementors (a specific class is generated for each ensemble type). If any kind of data flow

```
27 seamsEdlFactory factory = new seamsEdlFactory();
28 DataclassKnowledgeContainer container = new
       DataclassKnowledgeContainer();
29 container.storeDataClass(new CarComponent(…));
       ...
30 Collection<EnsembleInstance> formed =
       factory.createInstances(container);
31 for (EnsembleInstance instance : formed) {
       instance.performKnowledgeExchange();
   }
```

Figure 3. Using the framework from application code.

was used in the knowledge exchange specification, the returned groups must be iterated and a knowledge exchange method called on all of them (line 31). In a typical scenario, the group formation calls would be handled by the framework, and periodically scheduled to ensure adaptivity.

VI. Discussion

A. Inherent Problem Complexity

Since selecting an optimal configuration from a space of all possible configurations is clearly an NP-complete problem, this inherent complexity impacts the scalability of the framework. The scalability challenge has been elaborated on in [2] where we have considered some assumptions on the common sCPS scenarios that allow us to mitigate the issue by introducing filtering concepts, such as the *where* clause mentioned in Section III.A. These concepts allow the user to specify additional domain-specific knowledge, which makes it possible to reduce the size of the problem. While still NP-complete, the reduced problem is much smaller, allowing us to obtain approximate solutions to otherwise infeasible use cases. In order to further improve scalability, it is possible to move from running the solver in each formation step to only running it when the system performance degrades below a certain threshold, and using more lightweight ensemble formation strategies otherwise (e.g., trading components between ensembles, or applying predefined group adaptation tactics).

B. Data Type Limitations

As the underlying mechanism of the ensemble formation depends on an SMT solver, this naturally introduces some limitations on the possible form of the expressions used in the constraints and fitness function. As Z3 can deal with integer and boolean variables, these are supported in all EDL expressions. However, we currently do not have any way of representing floating point and string values in Z3, so the compiler has been configured to not allow them in any expression that must be passed to the solver. Of course, these values can still be used in other parts of the specification, such as when declaring knowledge fields, or in knowledge exchange. In the future, it may be possible to introduce simple expressions over strings, such as equality, and possibly prefixes and suffixes, by hashing the value of the string and storing the hash in the solver as a representation of the string. In fact, a similar method could be used to enable at least equality expressions for composite types.

C. Future Features

In addition to addressing the challenges mentioned above, some features of the framework are still to be implemented or improved. Apart from improving the user experience of the library, such as by introducing fully functional code completion in addition to the already present type validation, the expressivity of the framework is the main area that we would like to focus on, primarily contained in the following aspects.

***General aggregation functions*—**We currently support the sum aggregation which was chosen as the most critical and common form of aggregation. Supporting general aggregation reminiscent of functional languages would allow a wider range of situations to be captured in constraints and fitness functions.

***Function extensibility*—**EDL supports defining functions (e.g., absolute value) both for general and solver-related expressions and the functions are not hardcoded but stored as data, making the function set extensible. However, a definition of a new function currently requires that the library and the plugins are recompiled. Having the ability to load function definition classes from the class path of the project would therefore much improve the ease of use of the framework.

D. Related Work

As mentioned in Section I, the concept of autonomous component ensembles has been introduced within the scope of the ASCENS project. It appears to be suitable for modelling sCPS scenarios and has been applied to sCPS scenarios, such as the e-mobility use case [6]. Till now, the concept was realized in several component models and frameworks such as DEECo [4], JRESP (http://jresp.sourceforge.net), and Helena [7]. DEECo, JRESP and Helena offer the same concepts of components with roles and ensembles, however they do not provide self-formation of ensembles based on high-level specification. Ab^{a}CuS [8], which is a Java-based implementation of AbC [9] and can be seen as a kind of successor to JRESP, is not an ensemble-based framework, but it also employs components and dynamic opportunistic communication among them. However, as above, it does not provide self-formation of communication groups based on high-level specification. The concept of ensemble was also utilized in our previous work [10] as part of the simulation testbed implementation. Whereas the purpose of the testbed was to provide a case study driven sCPS simulation environment, this artifact delivers a general group formation library backed by the intelligent ensemble concepts and EDL.

VII. Contributions

In this artifact paper we have presented the Intelligent Ensembles, a framework for dynamic group formation based on a high-level specification. Due to its adaptive architecture capabilities and its high-level description language, we envision the framework to be useful for implementing all kinds of group-based use cases, as a part or spring board for implementation of another framework focused on group-wise self-adaptation, or as an example adaptive architecture framework in university software architecture courses. The framework is designed to be easily integrated with existing code, and general enough to be applied even outside the sCPS domain for which it is primarily intended.

Acknowledgment

This work was partially supported by the project no. LD15051 from COST CZ (LD) programme by the Ministry of Education, Youth and Sports of the Czech Republic, partially supported by Charles University Grant Agency project No. 390615, and partially supported by Charles University institutional funding SVV-260451.

References

1. Wirsing, M., Hölzl, M., Tribastone, M., Zambonelli, F.: ASCENS: Engineering Autonomic Service-Component Ensembles. In: Beckert, B., Damiani, F., Boer, F.S. de, and Bonsangue, M.M. (eds.) Proceedings of FMCO 2011 (Revised Selected Papers), Turin, Italy. pp. 1–24. Springer (2011).
2. Krijt, F., Jiracek, Z., Bures, T., Hnetynka, P., Plasil, F.: Automated Dynamic Formation of Component Ensembles. In: Proceedings of Modelsward 2017, Porto, Portugal. SCITEPRESS (2017).
3. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of TACAS'08, Budapest, Hungary. pp. 337–340. Springer (2008).
4. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F.: DEECo: An ensemble-based component system. In: Proceedings of CBSE 2013, Vancouver, Canada. pp. 81–90. ACM (2013).
5. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. Computer. 42, 44–51 (2009).
6. Hoch, N., Bensler, H.-P., Abeywickrama, D., Bureš, T., Montanari, U.: The E-mobility Case Study. In: Wirsing, M., Hölzl, M., Koch, N., and Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems. pp. 513–533. Springer (2015).
7. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling – The Helena Approach. In: Iida, S., Meseguer, J., and Ogata, K. (eds.) Specification, Algebra, and Software. pp. 359–381. Springer (2014).
8. Alrahman, Y.A., Nicola, R.D., Loreti, M.: On the Power of Attribute-Based Communication. In: Proceedings of FORTE 2016, Heraklion, Crete, Greece. pp. 1–18. Springer (2016).
9. Alrahman, Y.A., Nicola, R.D., Loreti, M.: Programming of CAS Systems by Relying on Attribute-Based Communication. In: Proceedings of ISOLA 2016, Corfu, Greece. pp. 539–553. Springer (2016).
10. Matena, V., Bures, T., Gerostathopoulos, I., Hnetynka, P.: Model Problem and Testbed for Experiments with Adaptation in Smart Cyber-physical Systems. In: Proceedings of SEAMS 2016, Austin, USA. pp. 82–88. ACM (2016).