An Architecture Framework for Experimentations with Self-Adaptive Cyber-Physical Systems

Michal Kit, Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, and Frantisek Plasil
Charles University in Prague
Faculty of Mathematics and Physics
Department of Distributed and Dependable Systems
Prague, Czech Republic
{kit, iliasg, bures, hnetynka, plasil}@d3s.mff.cuni.cz

***Abstract*—Recent advances in embedded devices capabilities and wireless networks paved the way for creating ubiquitous Cyber-Physical Systems (CPS) grafted with self-configuring and self-adaptive capabilities. As these systems need to strike a balance between dependability, open-endedness and adaptability, and operate in dynamic and opportunistic environments, their design and development is particularly challenging. We take an architecture-based approach to this problem and advocate the use of component-based abstractions and related machinery to engineer self-adaptive CPS. Our approach is structured around DEECo – a component framework that introduces the concept of component ensembles to deal with the dynamicity of CPS at the middleware level. DEECo provides the architecture abstractions of autonomous components and component ensembles on top of which different adaptation techniques can be deployed. This makes DEECo a vehicle for seamless experiments with self-adaptive systems where the physical distribution and mobility of nodes, and the limited data availability play an important role.**

***Index Terms*—Component framework, self-adaptation, cyber-physical systems**

I. Introduction

Adaptation to different situations and environments has become a common necessity in smart Cyber-Physical Systems (CPS) [1] – i.e., systems closely interacting with physical world entities. Such systems are typically open-ended and have to be capable of supporting new requirements and needs. At the same time, these systems are deployed in heterogeneous and ever-changing (sometimes even hostile) environments and thus have to promptly react to these changes. Due to limited connectivity, smart CPS typically have to perform adaptation in a decentralized manner, which adds to the overall complexity of designing the adaptation. Moreover, in complex enough systems such as modern smart CPS, the mutual dependencies of different *local* adaptation strategies may have an unexpected *global* impact – a behavior often referred to as *emergent*.

To correctly design complex self-adaptive smart CPS is thus a challenging task, which is only partially addressed by existing software engineering models and approaches. This stems from the fact that a correct design of smart CPS has to apply a holistic view that takes into account the overall system goals, the operational models of the system and its environment (along with the uncertainty present in these models), and the employed communication models (along with issues related to latencies and communication unavailability).

In this paper, we present DEECo (Dependable Emergent Ensembles of Components) [2] – a model and framework for developing complex smart CPS. In its model, DEECo provides the holistic view that combines the goals of a system, the system's operational model (including real-time constraints), and realistic communication model (including limited communication and latencies). With its framework, DEECo allows large-scale simulations of complex CPS. Combined with the real-time perspective of DEECo and the network-accurate simulation of communication, DEECo offers accurate insight into the effects of adaptation strategies in complex smart CPS.

The structure of the paper is as follows: Section II presents the running example of self-adaptive vehicles. Section III discusses the DEECo component model, while Section IV presents its reification delivered by the JDEECo simulation framework. Section V presents the IRM design method used in DEECo, while Section VI concludes the paper.

II. Running Example

To illustrate the DEECo models and significant features, we rely in this paper on a smart parking scenario. In the frame of this scenario, vehicles are equipped with vehicle-to-vehicle (V2V) communication and smart sensors to detect available parking spaces along the streets and exchange their knowledge about the available parking capacity (Figure 1).

From the architectural perspective, vehicles are represented as autonomous components, each consisting of a belief and real-time processes. While the belief (*knowledge* in DEECo) reflects the components' perspective about the available parking spaces, the real-time processes take care of sensing the current position, free parking spaces in visible range, etc. In addition to direct sensing, component enrich their belief by exchanging the belief with other components – i.e., the information about the available parking capacity is exchanged between vehicles that are in proximity (typically via limited-range V2V communication).

This information allows vehicles to adjust their route to effectively find a parking space. Also, the inter-component communication may be used to negotiate with other vehicles for selecting and reserving a parking space.

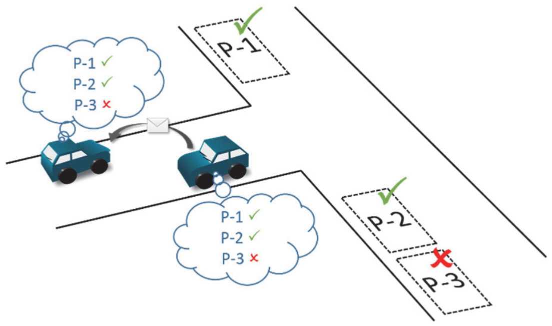

Figure 1: Vehicles sharing data about parking space capacity.

III. DEECo Component Model

DEECo offers straightforward support for development of self-adaptive CPS thanks to the following aspects (structured in sub-sections) that it offers to system architects and developers.

A. Dynamic ensemble-based component model

DEECo is built on the concepts of autonomous *components* and *ensembles* [3]. A component is an independent unit of computation and deployment, while an ensemble is a group of components cooperating to achieve a particular goal.

Ensembles are established/disbanded dynamically at runtime depending on the state of the environment and the state of the components. They can overlap, reflecting the fact that a component may take on multiple roles and pursue multiple goals at the same time (e.g., the goal of having up-to-date information about parking spaces and the goal of making sure that selected parking space is reserved).

The concept of ensembles thus allows forming dynamic component architectures and provides a straightforward reflection of operational goals in the application architecture. The concept of ensembles is further backed by theoretical research in coordination logics [4], which makes it possible to apply related results from statistical model-checking [5] (outside the scope of the artifact presented in this paper).

Figure 2 illustrates the concepts of DEECo's autonomous components and ensembles. It shows the specification (in DEECo DSL) of the smart parking scenario. A vehicle is captured as a `Vehicle` component – subject to multiple instantiations. It consists of its *knowledge* (i.e., the state of the component and its belief about other components – lines 5-11), and of real-time *processes* (lines 12-16). Processes in DEECo are periodic (time-triggered and event-triggered). This makes it easy to implement both real-time CPS control logic and adaptation logic as MAPE-K [6] loops.

Communication between components is not direct but happens via knowledge exchange between components in an ensemble. Figure 2 shows the `CapacityExchangeEnsemble`, which reflects the goal of vehicles having up-to-date information about the available parking spaces.

Technically, an ensemble is defined by its *membership* condition and its *knowledge exchange*. Membership determines which components to involve in an ensemble (e.g., all vehicles in proximity), whereas knowledge exchange specifies which

```
1.  role LinkCapacityAggregator:
2.    linkCapacities, position
3.
4.  component Vehicle features LinkCapacityAggregator
5.    knowledge:
6.      ID = V1
7.      linkCapacities = [(Link_21, 1), (Link_21, 2), …]
8.      position = {50.075306, 14.426948}
9.      speed = 54.2
10.     destination = Link_126
11.     selectedParking = P23
12.   process measureSpeed
13.       out speed
14.     function:
15.       speed ← SpeedSensor.read()
16.     scheduling: periodic( 500ms )
17.   … /* other process definitions */
18.
19. ensemble CapacityExchangeEnsemble:
20.   coordinator: LinkCapacityAggregator
21.   member: LinkCapacityAggregator
22.  membership:
23.   distance(member.position, coordinator.position) < ENSEMBLE_RADIUS
24.  knowledge exchange:
25.     coordinator.linkCapacities ← { m.linkCapacities | m ∈ members }
26.     for(m ∈ members)
27.       m.linkCapacities ←{ coordinator.linkCapacities }
28.  scheduling: periodic( 1000ms )
```

Figure 2: Examples of a DEECo component and an ensemble.

knowledge should be exchanged among these components. An ensemble may be instantiated multiple times if the situation described by membership occurs at different places (potentially involving different components).

To ease the structural specification of an ensemble, DEECo features two principal roles – ensemble *coordinator* and ensemble *member*. Membership is then expressed as a condition over the knowledge of the coordinator and the knowledge of the members (lines 22-23). Similarly, knowledge exchange is specified as assignment from the knowledge of members to the knowledge of the coordinator and vice-versa (lines 24-27).

The contract between ensembles and components is carried by component *roles* (lines 1-2). The role specifies the knowledge of a component that an ensemble can assume.

An ensemble periodically (in real-time manner – line 28) evaluates the membership condition and executes the knowledge exchange. In evaluating the ensembles, the model takes into account network latencies, which means that the knowledge of a component is available to other components only after certain random time, which is further correlated with the intensity of other network traffic and the geographical distance (in case of multi-hop communication in V2V networks).

B. Openness and extensibility for adaptation strategies

DEECo is open to deployment of different adaptation algorithms or strategies. They can be implemented as component processes and seamlessly integrated with a DEECo application. They can be also dynamically switched in response to: (i) values that are directly sensed by a component (e.g., free parking spaces around a vehicle), (ii) the state of a component (e.g., destination where a vehicle wants to park), and (iii) the belief about the knowledge of other components, including their perception of the environment (e.g., free parking spaces in another street).

```
@Component
public class VehicleComponent {
   public String id;
   public Map<…> linkCapacities;
   public Coord position;
   public Double speed;
   public Link destination;
   public Parking selectedParking;

   @Process
   @PeriodicScheduling(period=500)
   public static void measureSpeed(
      @Out("speed") Double speed)
         {…}
}

@Ensemble
@PeriodicScheduling(period=1000)
public class CapacityExchangeEnsemble {

   @Membership
   public static boolean membership(
      @In("coord.position") Coord cPos,
      @In("member.position") Coord mPos)
         {…}

   @KnowledgeExchange
   public static void exchange(
      @InOut("coord.linksCapacities") Map<…> cLinksCapacities,
      @InOut("member.linksCapacities") Map<…> mLinksCapacities)
         {…}
}
```

Figure 3: Code snippets from DEECo component and ensemble specification in Java.

While (i) and (ii) can be obtained by a component directly, the belief about the knowledge of other components (iii) comes as the result of ensemble evaluation and thus is subject to network latencies and limited network connectivity. This contributes to the realistic simulation of the adaptive behavior in decentralized smart CPS.

The switching of the strategies is facilitated by DEECo's "models@runtime" [7] approach, which makes it possible to inspect the architecture of an application at each time instant and modify the architecture as the result of an adaptation strategy.

The runtime model further provides a global view on a DEECo application (including knowledge of components and grouping of components in ensembles). This makes it possible to easily evaluate adaptation strategies by comparing (i) the adaptation taken by a component based on its incomplete (and potentially outdated) belief of the system and (ii) the ideal adaptation that should have been taken if the complete knowledge of the complete up-to-date state of the system and its environment were available.

C. Component and Ensemble Development

DEECo provides a mapping of its concepts (as exemplified by the DSL in Figure 2) to the Java programming language. Figure 3 gives an example of implementing the smart parking scenario in Java. All metadata are captured by annotations, which removes the necessity of having any accompanying specification (in DSL or XML) additional to the Java implementation of components and ensembles.

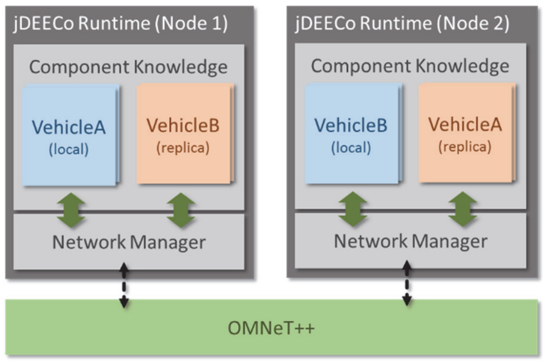

Figure 4: JDEECo runtime.

In the mapping, each component becomes a single class. Its knowledge becomes the class fields and its processes become the static methods. The knowledge that is consumed and produced by a process is specified as process parameters. This processes-knowledge separation allows DEECo runtime to manage snapshotting and atomic updating of knowledge. Similarly, each ensemble is represented as a Java class with a method for membership and a method for knowledge exchange.

IV. JDEECo Simulation Framework

DEECo component model comes with two runtime frameworks – one in Java[1], and one in C++[2]. While the C++ implementation targets actual deployment on embedded devices (e.g., STM32F4 MCU), the Java implementation (which constitutes the artifact presented in this paper) serves primarily for experimentations with autonomous components and self-adaptation. The Java implementation (*JDEECo*) provides a simulation framework which allows experimentations with decentralized adaptive behavior of smart CPS.

The simulation framework is integrated with OMNeT++[3] network simulator. All knowledge exchange passed between components is routed through OMNeT, which provides realistic estimates of network latency w.r.t. to network topology, geographical position of components, network collisions and packet drops, etc. By employing the INET and MIXIM extensions of OMNeT, JDEECo allows for simulating deployments in mixed network environment combining IP networks and mobile/vehicle ad-hoc networks (MANETS / VANETS), as found in modern smart-* systems. Figure 4 illustrates the JDEECo runtime and OMNeT integration.

A simulation of a DEECo application typically requires simulation of responses of the environment (e.g., simulation of car movement in a city). In rapid prototyping, this can be realized at the architectural level by including an "Environment" component, which, by means of models@runtime manipulation, gathers actuation from all components and feeds them with sensing. For more systematic simulations, JDEECo offers a generic sensor/actuator interface and access to the simulation

[1] JDEECo: http://github.com/d3scomp/JDEECo
[2] CDEECo: http://github.com/d3scomp/CDEECo
[3] OMNeT++ : http://www.omnetpp.org/

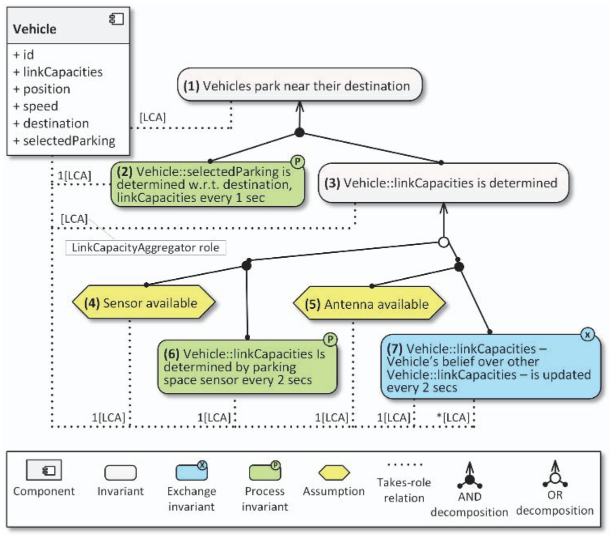
Vehicle
+ id
+ linkCapacities
+ position
+ speed
+ destination
+ selectedParking
(1) Vehicles park near their destination
[LCA]
1[LCA]
(2) Vehicle::selectedParking is determined w.r.t. destination, linkCapacities every 1 sec
P
(3) Vehicle::linkCapacities is determined
[LCA]
LinkCapacityAggregator role
(4) Sensor available
(5) Antenna available
(6) Vehicle::linkCapacities is determined by parking space sensor every 2 secs
P
(7) Vehicle::linkCapacities – Vehicle's belief over other Vehicle::linkCapacities – is updated every 2 secs
X
1[LCA]
1[LCA]
1[LCA]
1[LCA]
*[LCA]
Component
Invariant
Exchange invariant
Process invariant
Assumption
Takes-role relation
AND decomposition
OR decomposition

Figure 5: IRM tree for the smart parking scenario.

scheduler, which allows plugging-in existing simulators. To date, we have integrated MATSim traffic simulator this way.

V. Invariant Refinement Method

In order to reason about self-adaptation during the design phase, DEECo framework provides the Invariant Refinement Method (*IRM*). IRM is based on goal-oriented requirements elaboration that stems from methodologies such as KAOS [8] and Tropos/i* [9]. IRM captures goals and requirements of the system as *invariants* that describe the desired state of the system-to-be at every time instant. This corresponds to the operational normalcy of the system-to-be and thus aligns well with the need of continuous operation of autonomic component ensembles. IRM is based on iterative decomposition of abstract goals. It is primarily a top-down method, where top-level invariants constitute high-level (general) goals of the application and are further decomposed into more specialized (fine-grained) invariants, which eventually map into concrete component processes (reflecting component responsibilities) and ensembles.

To induce self-adaptivity in architecture design so that the system would react to changing situations in the environment at runtime, IRM captures and exploits architecture variability (in certain potentially overlapping situations) by OR-decompositions. In particular, the designed architecture configurations corresponding to distinct situations that can be encountered at runtime are further elaborated to produce alternative realizations of system requirements.

In Figure 5, the IRM decomposition tree for the example scenario is depicted. There, a simple example of self-adaptation in practice is given. To determine available parking space a Vehicle has two possibilities: (i) to use its own parking space sensor, which is however constrained and provides only readings in the immediate proximity to the vehicle (i.e., current road link), or (ii) to use the information exchanged with other vehicles. As shown in the figure, in order to take advantage of those methods certain assumptions need to hold at runtime – i.e., the reading sensor or the antenna need to remain operational and available for method (i) or (ii), respectively. In the best case, both assumptions hold and as such both methods are used at the same time. However, in cases where there is a problem with ensuring either of those assumptions, the component remains operational and exploits only one of the possible alternatives. By this, Vehicle adapts itself to the situation in the deployment environment and tries to achieve the best possible output, selecting among the available parking spaces near to its destination, always according to the available information.

VI. Summary

To correctly design complex self-adaptive smart CPS is a hard task stemming from the fact that a correct design of such systems has to apply a holistic view that takes into account multiple aspects, many times even conflicting ones.

In this paper, we have briefly introduced DEECo framework, which is intended for development and simulations of such complex self-adaptive smart CPS. In contrast to other frameworks, DEECo (i) is open and easily extensible, (ii) offers a dynamic component model based on ensembles, (iii) has two implementations for experimenting with smart CPS and self-adaptivity, (iv) provides a goal-based design method taking into account self-adaptation, and (v) allows for simulations of real-life deployment by evaluating the system behavior under different network configurations and settings (taking into account also network latency and limited connectivity).

The paper comes together with the artifact containing the JDEECo implementation of the example from Section II and integrated with the JDEECoSim tool. It can be accessed from http://self-adaptive.org/exemplars/v2v-DEECo.

References

[1] *Cyber-Physical Systems: Driving Force for Innovation in Mobility, Health, Energy and Production*. Munich, Germany: National Academy of Science and Engineering, 2011.

[2] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECo: An ensemble-based component system," in *Proceedings of CBSE '13*, Vancouver, Canada, 2013, pp. 81–90.

[3] ASCENS, "Autonomic Service-Component Ensembles D4.2: Second Report on WP4," 2012.

[4] 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, F. de Boer, and M. Bonsangue, Eds. Springer Berlin Heidelberg, 2013, pp. 25–48.

[5] R. De Nicola, A. Lluch Lafuente, M. Loreti, A. Morichetta, R. Pugliese, V. Senni, and F. Tiezzi, "Programming and Verifying Component Ensembles," in *From Programs to Systems. The Systems perspective in Computing*, vol. 8415, S. Bensalem, Y. Lakhneck, and A. Legay, Eds. Springer Berlin Heidelberg, 2014, pp. 69–83.

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

[7] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[8] A. van Lamsweerde, "Requirements Engineering: From Craft to Discipline," in *Proceedings of FSE '08*, Atlanta, Georgia, USA, 2008, pp. 238–249.

[9] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An Agent-Oriented Software Development Methodology," *Auton. Agents Multi-Agent Syst.*, vol. 8, no. 3, pp. 203–236, May 2004.