Towards Dependable Emergent Ensembles of Components: The DEECo Component Model

Jaroslav Keznikl[1,2], Tomáš Bureš[1,2], František Plášil[1,2], Michal Kit[1]

[1] Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
{keznikl, bures, plasil, kit}@d3s.mff.cuni.cz

[2] Institute of Computer Science
Academy of Sciences of the Czech Republic
Prague, Czech Republic
{keznikl, bures, plasil}@cs.cas.cz

***Abstract*—In the domain of dynamically evolving distributed systems composed of autonomous and (self-) adaptive components, the task of systematically managing the design complexity of their communication and composition is a pressing issue. This stems from the dynamic nature of such systems, where components and their bindings may appear and disappear without anticipation. To address this challenge, we propose employing separation of concerns via a mechanism of dynamic implicit bindings with implicit communication. This way, we strive for dynamically formed, implicitly interacting groups – ensembles – of autonomous components. In this context, we introduce the DEECo component model, where such bindings, as well as the associated communication, are managed in an automated way, enabling transparent handling of the dynamic changes in the system.**

Keywords—component; ensemble; adaptation; dynamic architecture; implicit communication; implicit bindings

I. INTRODUCTION

In component-based software architecture design, we still face many challenges, particularly in the case of large, distributed and dynamically changing applications, where both components and bindings may appear/disappear without anticipation. Therefore, components are often designed as autonomous [1] so that they stay operable regardless of the changes in their operating environment. This in turn implies the need for a (self-) adaptation mechanism [2].

In this context, a challenge is to find suitable paradigms for engineering such systems to overcome the design complexity of their communication and composition, specifically in terms of their autonomic and dynamic nature.

As for autonomy and (self-) adaptation, these have been partially addressed by agent-based approaches [3][4] where actors leveraging on messaging establish explicit bindings for data and code exchange. As for coping with dynamism, techniques utilizing implicit bindings while focusing on explicit communication have been proposed [5]. Furthermore, separation of concerns was to some extent achieved by introducing implicit communication (driven by a third-party entity) via explicit bindings [6]. Intuitively, it is desirable to combine all of these approaches in order to take advantage of the benefits they offer simultaneously.

The work was partially supported by the EU project ASCENS 257414, the Grant Agency of the Czech Republic project P202/11/0312. The work was partially supported by Charles University institutional funding SVV-2012-265312.

Contributing to the ASCENS project [7], our goal is to respond to this challenge by elaborating on the idea of dynamic implicit bindings with implicit communication. To do so, we introduce the DEECo component model (Dependable Emergent Ensembles of Components).

The basic idea of DEECo is to facilitate separation of concerns by extracting component bindings and communication from the component implementation, expressing them implicitly, and managing them in an automated way via the DEECo runtime framework. Specifically, we consider bindings to be declaratively-expressed first-class entities, capturing component communication by implicit data exchange. This way, a component can be considered as an autonomous and self-aware entity, relying solely on its local data, which are modified in the background by the runtime framework according to the implicit component bindings. Similar to self-organizing architectures [8], such bindings facilitate dynamic forming of implicit groups – ensembles – of autonomous components.

Moreover, stemming from the need for autonomy while allowing for dependability, in DEECo we aim at supporting (self-) adaptation, code mobility, and verification of safety (reachability) properties.

The rest of this paper is structured as follows: Section II describes our motivating case study, in Section III the requirements for effectively addressing the outlined goals, demonstrated by the case study, are summarized, Section IV provides a brief description of the DEECo component model, while the concluding Section V outlines a long term DEECo vision and identifies the key challenges to be addressed.

II. CASE STUDY

As our motivating case study we consider a robotic playground scenario, stemming from the e-mobility demonstrator [9] in ASCENS. Basically, it pertains to several autonomous robots moving on roads with crossings. When approaching a crossing, all the robots in the same situation, or already on the crossing, have to cooperate in order to avoid collision. An assumption is that the robots can communicate only with those within a short range, since they typically have limited communication signal coverage. Thus the architecture of the system of robots and crossings is dynamic, determined by their actual positions.

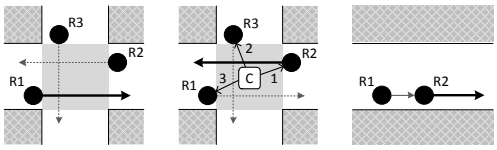

Figure 1. Robot Case Study: (a) autonomous robots, (b) robots advised by a crossing, (c) convoy

In the basic case (Fig. 1.a), we assume that the robots give priorities according to the "right-hand rule" (e.g., R1 has the highest priority). Furthermore, we consider also other (more elaborate) variants for the crossing strategy (Fig. 1.b), where the robots are advised by the crossing itself (similar to crossings with specific arrangements of traffic lights; e.g., the robot R2 is advised by C to continue as the first). These variants are handled by self-adaptation of the robots, including both short-term and permanent adaptation. As an example of the former case, the crossing provides the corresponding robots with a strategy for interacting with it only for the time the robots are at/in the crossing; in the latter case, robots exchange strategies for interacting with new variants of crossings, and these strategies are adopted permanently. Finally, we also expect the robots to form dynamic convoys (Fig. 1.c); i.e., if two robots drive in the same direction, one behind the other, the robot behind (e.g., R1) should adjust its speed to the one in front (e.g., R2). We will use this robot playground case study as the running example throughout this paper.

In addition to the robot scenario, we also seek inspiration from a more elaborate case study of a self-aware and self-adaptable cloud platform [9]. We consider several client applications running on a cloud platform, continuously storing their logging data via a logging service. An important part of the scenario is that the application processes, as well as the processes implementing the logging service, can migrate between the nodes of the cloud according to various optimization criteria. These processes should migrate autonomously and be able to adapt the migration strategy according to the impact of previous migrations. During migrations, client applications should not be affected by the changes in the cloud architecture.

III. Overview of Requirements

Based on the case studies, we have identified several general requirements to be met by the DEECo component model. These include the capability to:

- allow for convenient design with a suitable level of abstraction and proper concepts, coping efficiently with dynamic and parallel activities.
- provide appropriate means for continuous self-adaptation of the system. This implies the need for separation of concerns, so that adaptation is separated from business logic.
- achieve dynamic updates of behavior by means of both (self-) adaptation and code mobility.
- ensure a high level of dependability by supporting methods for formal verification of safety (reachability) properties.

As the requirements are also partially targeted by the SCEL [10] specification language proposed for ASCENS, we will reuse some of its related concepts. However, since SCEL is a low-level generic theoretical model, it does not provide any higher-level abstractions for system design. Supporting only low-level primitive operations for component communication without considering any programming environment, it is not, as such, suitable for direct development of non-trivial software systems.

IV. DEECo Component Model

In this section, we target the requirements identified in Section III by introducing the DEECo component model. Its basic idea is to manage the dynamism of a system by externalizing the (potentially distributed) communication among components. Specifically, a component accesses only its local data, which are communicated in the background to other components in an automated way by the DEECo runtime framework. Hence, a component is logically an autonomous unit, oblivious to the way data are exchanged, which makes it robust and adaptable with respect to dynamism. Moreover, the DEECo data exchange mechanism supports code mobility and adaptation.

A. Component Structure

A component is a unit of design and deployment, consisting of knowledge and processes.

Knowledge represents the internal state and functionality of the component. It is a hierarchical data structure, similar to a tuple space [10], mapping identifiers to (potentially structured) values. Values are either statically-typed data or functions; both being first-class entities. Only pure functions with no global variables are considered.

A *process*, being essentially a "thread", operates locally upon the knowledge by calling a specific function (being a part of the knowledge) to perform its task. Since global variables are disallowed, a process assigns a part of the knowledge to the actual parameters of the function (*input knowledge*), and on its completion updates a part of the knowledge (*output knowledge*) by the return value.

The example from Fig. 2 describes the component `Robot` (a singleton instance; multiple instances are expected to be created by cloning) in the DEECo DSL. It illustrates that a component is defined solely by its initial knowledge, which also syntactically contains the definition of the component's processes. Here, the `Robot` component's knowledge contains

```
component Robot = {
   id: RobotId = "R1";
   info: RobotInfo = {
      position: Position = { x = 1, y = 1};
      path: list Position = [];
   };
   otherRobots: map RobotId -> RobotInfo = {};
   stepf: fun(inout i: RobotInfo, in o: map RobotId->RobotInfo) = {
      ... };
   processes = {
      step: Process = {
         function = stepf;
         inputKnowledge=[info, otherRobots];
         outputKnowledge=[info];
         scheduling = PERIODIC(100ms);
}; }; };
```

Figure 2. Example of a DEECo component

```
interface IRobot = {
   id: RobotId;
   info: RobotInfo;
   otherRobots: map RobotId -> RobotInfo;
};
ensemble AutonomousRobotsEnsemble {
   member-interface: IRobot;
   coordinator-interface: IRobot;
   membership: fun(in r: IRobot, in c: IRobot, out ret: Boolean) = {
      ret = proximity(r.info.position, m.info.position) <= TRESHOLD;
   };
   coordinator-to-member: fun(inout m: IRobot, in c: IRobot) = {
      m.otherRobots=m.otherRobots.merge(c.otherRobots).except(m.id);
   };
   member-to-coordinator: fun(in m: IRobot, inout c: IRobot) = {
      c.otherRobots[m.id] = m.info;
}; };
```

Figure 3. Example of an ensemble prescription

the actual position of the robot and the list of remaining waypoints the robot has to drive through (`info`), and similar information about the robots in its close perimeter (`otherRobots`). The `Robot`'s only process `step` moves the robot (via the `stepf` function) by updating its `info.position` according to the `info.path` while considering `otherRobots` in order to avoid a crash (by applying the right-hand rule).

B. Component Composition

In DEECo, the composition of components is flat, in the form of component *ensembles* – groups of components, consisting of a single (unique) coordinator and multiple member components. At the same time, a component may play the role of a coordinator or member in several ensembles.

Supporting separation of concerns, an ensemble mediates communication between the coordinator and members. In consequence, two components can communicate only if they are involved in the same ensemble and one of them is the coordinator (direct communication among the members is not possible). Most importantly, such an involvement is expressed implicitly via a membership condition, evaluated in an automated way by the runtime framework.

Similarly, the inter-component communication is realized by implicit knowledge exchange (i.e., a part of the knowledge of one component is copied to the other component in an automated way). Such exchange may also include a knowledge transformation.

In compliance with the principle of knowledge exchange solely between the coordinator and a member, an ensemble is described pair-wise, defining the couples coordinator – member. Syntactically, an *ensemble prescription* consists of the desired knowledge *interface* of the coordinator (*coordinator interface*), the desired interface of a member (*member interface*), *membership function*, and *mapping function*.

Interface constitutes a structural prescription for a partial view on a component's knowledge. Specifically, it is associated with the knowledge by means of duck typing (structural subtyping); i.e., if a part of the component's knowledge matches the structure prescribed by the interface, then the component reifies the interface. For example, `Robot` from Fig. 2 reifies the `IRobot` interface from Fig. 3.

Membership function declaratively expresses the membership condition, under which two components form a pair

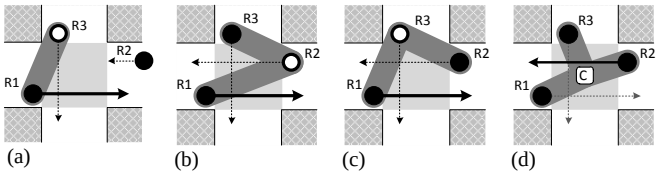

Figure 4. Ensemble Examples: (a) two-robot ensemble with coordinator R3, (b) autonomous robots ensemble with coordinator R2, (c) autonomous robots ensemble with coordinator R3, (d) crossing ensemble

coordinator – member of the ensemble. This condition is defined upon the knowledges of the components and is to be evaluated by the runtime framework (potentially in a distributed fashion). For example, in Fig. 3 the components `r` and `c`, reifying the `IRobot` interface, have to be in the proximity lower than `THRESHOLD` in order to form a coordinator-member pair.

Mapping function determines the knowledge exchange between the coordinator and a member. Specifically, it describes which part of the knowledge of one component is to be transferred to the other and how it is potentially transformed. We assume a separate mapping for each of the directions, coordinator-to-member and member-to-coordinator. Also, the mapping function is to be executed by the runtime framework. This basically ensures that relevant knowledge changes in one component are propagated to the other in the background. As an example, consider the `coordinator-to-member` and `member-to-coordinator` mapping functions from Fig. 3 which ensure an exchange of knowledge necessary to avoid robot collisions (i.e., the positions and remaining paths of the robots in a close perimeter).

In general, components form an ensemble whenever they satisfy the *ensemble condition* of an ensemble prescription, i.e., one of them reifies the coordinator interface, the other components reify the member interface, and the membership condition holds for each coordinator – member pair. Therefore, multiple ensembles based on the same prescription can be formed simultaneously.

As an example, consider an ensemble prescription of autonomous robots where the membership condition requires the member robots to be in close proximity to the coordinator robot. In Fig. 4.a, R2 is too far from the coordinator R3 so it is not (yet) included in the ensemble [R1, R3]. After R2 reaches the required proximity, all three robots will form a single ensemble as shown in Fig. 4.b and Fig. 4.c (bigger ensembles are preferred to smaller ones and the coordinator is selected randomly if multiple candidates are eligible). Assuming the crossing strategy, where components are advised by the crossing, the ensemble will potentially look like the one in Fig. 4.d, where the crossing component is the coordinator.

In the situation where a component satisfies the ensemble condition for multiple ensembles (Fig. 4.b and Fig. 4.c), we envision a mechanism for deciding whether all or only a subset of the candidate ensembles should be formed. Currently, we employ a mechanism based on a partial order over ensembles (the ensemble with higher order is preferred; incomparable ensembles are formed simultaneously).

C. Computational Model

The computational model of DEECo is based on asynchronous knowledge exchange and process execution, stemming from the asynchronous nature of dynamic distributed systems. Specifically, the processes of all components execute in parallel as independent threads either periodically or when triggered by modification of (a part of) their input knowledge. In a similar vein, a binding in an ensemble is accomplished by a separate activity, running the mapping function again either periodically or when triggered by a change in the knowledge of the coordinator/member.

Due to the asynchrony, it is necessary to ensure that knowledge is accessed consistently. Thus, at its start, a process is atomically provided with a copy of its input knowledge so that its computation is not affected by later-occurring knowledge modifications. When finishing, the process atomically updates its output knowledge. The same atomic copy-on-start and update-on-return semantics also applies to the membership and mapping functions of ensembles. Technically, this semantics can be implemented for instance via messaging.

For the time being, we envision employing the "single writer, multiple readers" rule for knowledge access, meaning that at any time each value in the knowledge of a component has at most one writer while being accessed by potentially multiple readers. Note that this rule applies to obtaining input and writing output knowledge of component processes, as well as to knowledge exchange via mapping functions. Since all the readers and writers are well defined, we envision that compliance with this rule will be verified.

Consequently, based on the computational model, an ensemble is created when the ensemble condition starts to hold, and is discarded when the condition gets violated. Technically, as the whole system is asynchronous and potentially distributed, techniques for handling inherent delays, while creating/discarding ensembles, have to be carefully chosen.

V. Discussion: Vision and Challenges

We assume DEECo will be employed in the design of systems of autonomous self-adaptive components, such as a self-managing cloud platform and self-organizing car sharing [9], where it aims at simplifying the design process.

Specifically, we expect DEECo to effectively handle knowledge exchange among distributed components, including code mobility in support of adaptation, while putting a strong emphasis on separation of concerns. Although similar to software connectors [11], DEECo ensembles capture component composition implicitly and thus allow for handling of dynamic changes in an automated way. Similar benefits result from the implicit knowledge exchange.

Currently, we foresee two possible methods for handling distributed knowledge exchange: message passing and distributed tuple spaces, both already adopted by the state-of-the-art agent-oriented frameworks such as [3] and [12], respectively. Although supporting dynamic features such as code mobility, these frameworks lack high-level abstractions allowing for implicit dynamic composition and communication. Nevertheless, since DEECo components resemble agents with respect to autonomy, we consider partially employing these frameworks in the DEECo runtime framework. Currently, we already have prototypes for both types of these methods for handling knowledge exchange[1].

In order to support controlled architecture evolution, we aim to incorporate mechanisms for dynamic addition, modification, and removal of ensemble prescriptions.

In addition, we envision supporting formal verification of DEECo applications. As for model checking of temporal properties, we assume a mapping of applications to SCEL and intend to exploit its means [12] for this purpose. Moreover, we anticipate also employing stochastic model checking [13][14] for quantitative verification.

Finally, inspired by the cloud and e-mobility case studies, we intend to introduce, in addition to abstractions for performance awareness, other forms of implicit knowledge-based communication such as distributed consensus.

References

[1] J. O. Kephart, and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, IEEE CS, 2003, pp. 41-50.

[2] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation," Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA 2009), 2009, pp. 171–180.

[3] F. Bellifemine, G. Caire, and D. Greenwood, "Developing multi-agent systems with Jade," John Wiley & Sons, 2007.

[4] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi, "Modeling adaptation with a tuple-based coordination language," Proc. of 27th Symposium on Applied Computing (SAC 2012), 2012, in press.

[5] C. Escoffier and R. S. Hall, "Dynamically adaptable applications with iPOJO service, " Software Composition, 2007.

[6] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," Proc. of Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), 2006, pp. 3-12.

[7] ASCENS [Online], http://www.ascens-ist.eu.

[8] C. Villalba, M. Mamei, and F. Zambonelli, "A self-organizing architecture for pervasive ecosystems," Self-Organizing Architectures, volume 6090 of LNCS, pp. 275–300, 2010.

[9] N Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther, "Requirement specification and scenario description," ASCENS Deliv. D7.1, November 2011.

[10] R. De Nicola, G. Ferrari, M. Loreti , and R.Pugliese, "Languages primitives for coordination, resource negotiation, and task description," ASCENS Deliv. D1.1, 2011, http://rap.dsi.unifi.it/scel/.

[11] R.N. Taylor, N. Medvidovic, and E.M. Dashofy: "Software architecture: foundations, theory, and practice," Wiley, 2010.

[12] L. Bettini et al. "The Klaim project: theory and practice," In global computing. Programming Environments, Languages, Security, and Analysis of Systems, volume 2874 of LNCS, 2003, pp. 88-150.

[13] M. Z. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2010), Springer, 2010, pp. 23-37.

[14] J. Barnat, L. Brim, I. Cerna, M. Ceska, and J. Tumova: "ProbDiVinE, a parallel qualitative LTL model checker," Quantitative Evaluation of Systems (QEST 07), IEEE, 2007.

[1] http://d3s.mff.cuni.cz/projects/components_and_services/deeco/