A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase

Tomas Bures[1], Rocco De Nicola[2], Ilias Gerostathopoulos[1], Nicklas Hoch[3], Michal Kit[1],
Nora Koch[4], Giacoma Valentina Monreale[5], Ugo Montanari[5], Rosario Pugliese[6], Nikola Serbedzija[7],
Martin Wirsing[4], Franco Zambonelli[8]
[1]Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
[2]IMT Institute for Advanced Studies Lucca, Italy
[3]Corporate Research Group, Volkswagen AG, Wolfsburg, Germany
[4]Ludwig-Maximilians-Universität München, Germany [5]Dipartimento di Informatica, Università di Pisa, Italy
[6]Università di Firenze, Italy [7]Fraunhofer FOKUS Berlin, Germany [8]Università di Modena e Reggio Emilia, Italy

***Abstract*—Component ensembles are a promising way of building self-aware autonomic adaptive systems. This approach has been promoted by the EU project ASCENS, which develops the core idea of ensembles by providing rigorous semantics as well as models and methods for the whole development life cycle of an ensemble-based system. These methods specifically address adaptation, self-awareness, self-optimization, and continuous system evolution. In this paper, we demonstrate the key concepts and benefits of the ASCENS approach in the context of intelligent navigation of electric vehicles (e-Mobility), which itself is one of the three key case studies of the project.**

I. Introduction

The development of massively distributed and highly dynamic systems which interact with and control the physical world is one of the major challenges in software engineering [1]. This is because the dynamicity of these systems and the not well foreseeable context brought by the external physical environment demand that software operating in these systems is highly self-aware, autonomic and adaptive. While self-awareness and adaptivity has been relatively well mastered in case of small-scale localized control (especially in the field of control systems), it is still a major problem for large-scale distributed systems, which are open-ended and dynamic (meaning that components of the system may freely appear and disappear as well as change their communication partners).

Within the EU project ASCENS[1] an approach based on ensembles of components is pursued. Contrary to classical component-based software engineering, it features important concepts of knowledge and ensembles. The *knowledge* of a component is a structured repository of facts with well-defined relations. The facts in the knowledge change at runtime to reflect the state of the component and its belief about its environment, thus effectively addressing the self-awareness of a component. The *ensembles* are dynamic goal-oriented communication groups of components. The ensembles are formed on demand to reflect intentions of components with respect to the current state of their environment. This way, ensembles address the dynamicity and adaptivity of components.

In addition to providing basic concepts and their semantics, ASCENS wraps this into a holistic *ensemble development life cycle* (EDLC) framework, which covers the full development life cycle and addresses design and development for adaptation, self-awareness, self-optimization, and continuous system evolution.

In this paper we take a practitioner's approach and demonstrate the application of the EDLC on the development of one of the key ASCENS case-studies – the intelligent navigation of electric vehicles (e-Mobility).

The paper is structured as follows: Section II describes the e-Mobility case study and Section III outlines the EDLC and describes a high-level strategy of applying it to e-Mobility. Sections IV–IX demonstrate the particular EDCL steps applied. The evaluation and related work is presented in Section X, while Section XI concludes the paper.

II. E-Mobility Case Study

The e-Mobility scenario focuses on avoiding contingency situation in an open-ended systems of interacting electric vehicles. Such a scenario is highly dynamic. This stems mostly from the fact that it includes unforseeable human user actions which influence the availability of travel resources.

Technically, we assume in the case-study that travels are initiated by personal activities. A journey is thus defined as a sequence of trips, with each trip being initiated by a single activity. Trips may consist of multiple stages. A stage can be executed in different travel modes such as walking mode or driving mode. For example, consider a user that leaves for work in the morning. Work is the activity that initiates travel. The first trip contains a walking stage from home to the vehicle's parking lot, a driving stage from the parking lot at home to the one at work and lastly a walking stage to the office. The working time at the office is considered to be the activity duration. Throughout that time the vehicle is parked at the car park. If it has access to a charging station, it may recharge. After work the user continues his journey. The number of consecutive trips follows from the number of activities.

This work has been partially sponsored by the EU project ASCENS, FP7 257414.

[1]http://www.ascens-ist.eu

In this scenario the main components are the user, the electric vehicle, the parking lot and charging station. Parking lot and charging station are commonly referred to as infrastructure components. Component temporarily form ensembles. These ensembles include (i) collection of charging stations, (ii) collection of parking lots, (iii) collection of users and electric vehicles and (iv) collection of at least one user, one electric vehicle and one infrastructure component, etc.

Throughout runtime, contingency situations may occur. Components and ensembles require self-adaptive actions to resolve these situations. Examples of contingency situations that need to be resolved by the electric vehicle component include (i) unavailability of a reserved parking lot, (ii) unavailability of a reserved charging station, (iii) falling below minimum battery energy level and (iv) missing a scheduled arrival time. Examples of contingency situations that need to be handled by the parking lot or charging station component include (i) early or late arrival of a vehicle at a parking lot or charging station, (ii) early or late departure of a vehicle from a parking lot or charging station, (iii) missed initiation of a scheduled charging action and (iv) deviation from the expected power profile during charging.

III. Applying EDLC to e-Mobility – Big Picture

Within the scope of the ASCENS project we propose a "double-wheel" *ensemble development life cycle* (EDLC) – see Fig. 1 – for autonomic systems such as the e-Mobility. The aim is to provide a conceptual framework that covers the main aspects of the engineering process required for such systems. The "first wheel" is used for representing the phases that are performed *offline*, which are mainly those related to *design*. The "second wheel" focus on the phases related to *online* activities that are performed at *runtime*. Both are connected by the transitions *deployment* and *feedback* from design to runtime, and vice versa, respectively. This software life cycle is designed to specifically support the development of ensembles characterized by their complexity and self-* properties, such as self-awareness, self-expression and self-adaptation.

The *offline* activities are grouped into requirements engineering, modeling and programming, and verification and validation phases. In addition to model functional and non-functional requirements as in traditional requirements engineering, the focus is on modeling aspects of self-adaptation and self-awareness. These specific requirements need to be validated and verified as well.

The *online* activities comprise monitoring, awareness and self-adaptation. Monitoring consists on the observation of the environment and the behavior of the systems. Reasoning on the collected data is also a key aspect. Finally, adaptation is performed in order to change the system according to the knowledge acquired during monitoring and the reasoning performed by the awareness engine.

In this paper we describe how we have applied the EDLC on the e-Mobility case study. In the spirit of EDLC, we have employed several interrelated methods developed in ASCENS to address the application life cycle of the e-Mobility. In particular, we start with the specification of requirements and

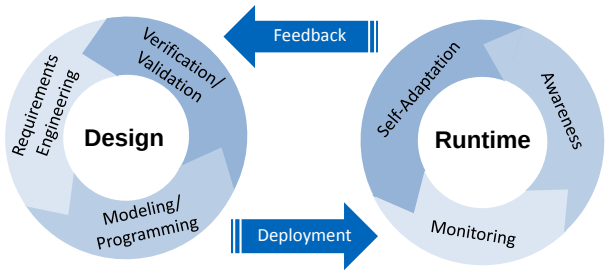

Figure 1. Ensembles Development Life Cycle (EDLC).

their reflection in the operation space of the system and system's self-awareness (described in Section IV). Following the requirements specification, we focus on the high-level architectural design, both in terms of adaptation patterns (Section V) and in goals decomposable into individual components and ensembles (Section VI). Next, emphasis is put on low-level design of component activities. For that, we employ the SCEL language, which is specifically intended for ensemble-based description of communication and coordination-oriented concerns (Section VII). Along with SCEL we also employ (Section VIII) the SCLP (soft-constraints logic programming), which provides a natural way of describing optimization related tasks, which are very frequent in self-adaptive systems. The final step is the implementation of components and the deployment of the system (Section IX), for which we use a dedicated ensemble-based component model and component runtime (called DEECo).

IV. Requirements Engineering with SOTA

Requirements engineering is of paramount importance to understand the adaptation needs of a system-to-be [3]. In the area of adaptive systems, and more in general of open-ended systems, the most appropriate approach is to adopt a goal-oriented requirements engineering one, and accordingly model requirements in terms of *goals* [4].

A goal represents a desirable state of the affairs that an entity, let it be a software component or an ensemble, aims to achieve. The idea of goal-oriented modeling of requirements naturally matches goal-oriented and intentional entities, such as organizations and multi-agent systems. However, self-adaptation too is naturally perceivable as an "intentional" quality. In fact, a self-adaptive component/ensemble should be engineered not simply to "achieve" a functionality or state of the affairs, but rather to "strive to achieve" such functionality, i.e., be able to take self-adaptive decisions and actions so as to preserve its capability of achieving despite unexpected contingencies and environmental changes.

Within the ASCENS development life cycle, SOTA [5] proposes itself as an extension of existing goal-oriented requirements engineering approaches that integrates elements of dynamical systems modeling, so as to account for the general needs of dynamic self-adaptive systems and components.

SOTA, which stems for "state of the affairs", models the entities of a self-adaptive system as if they were immersed in n-dimensional space $\mathbf{S}$, each of the n dimensions representing

a specific aspect of the current situation of the entity/ensemble and of its operational environment. As the entity executes, its position in $\mathbf{S}$ changes either due to its specific actions or because of the dynamics of environment. Thus, system evolution can be seen as movements in $\mathbf{S}$.

The activity of requirements engineering for self-adaptive systems in SOTA requires identifying the dimensions of the SOTA space, which means modeling the relevant information that the different components and ensembles of a system have to collect to become aware of their location in such space. In e-mobility, the space $\mathbf{S}$ includes the spatial dimensions related to the street map, but also dimensions related to the current traffic conditions, the battery conditions, etc.

Once the SOTA space is defined, a goal in SOTA can be expressed in terms of a specific state of the affairs to aim for, that is, a specific point or a specific area in $\mathbf{S}$ which the component or ensemble should try to reach in its evolution, in spite of external contingencies that can move the trajectory farther from the goal. For instance, a goal for a vehicle could imply reaching a position in the SOTA space that, for the dimensions representing the spatial location, trivially represents the final destination and for the dimension representing the battery condition may represent a charging level ensuring safe return.

V. From SOTA to High-level Design with Adaptation Patterns

The SOTA modeling approach is very useful to understand and model the functional and adaptation requirements, and to check the correctness of such specifications (as described in [5]). However, when a designer considers the actual design of the system, it is important to identify which architectural schemes need to be chosen for the individual components and ensembles.

To this end, in previous work [6], we defined a taxonomy of architectural patterns for adaptive components and ensembles of components. This taxonomy has the twofold goal of enabling reuse of existing experiences and providing useful suggestions to a designer on selecting the most suitable patterns to support adaptability.

At the center of our taxonomy is the idea that self-adaptivity requires the presence (explicit or implicit) of a *feedback loop* or control loop. A feedback loop is the part of the system that allows for feedback and self-correction towards goal achievement, i.e., self-adjusting behavior in response to changes in the system. Feedback loops provide a generic mechanism for adaptation as they provide the means for inspecting and analyzing the system at the component or ensemble level and for reacting accordingly.

However, when it comes to choosing among a variety of possible architectural schemes that can be employed for feedback loops [6], it becomes clear that the specific characteristics of goals identified in the requirements engineering phase directly guides the choice of specific feedback loop patterns. In particular, the choice of a specific pattern depends on how (and to which extent) the components of the system have component-specific goals with very different characteristics,

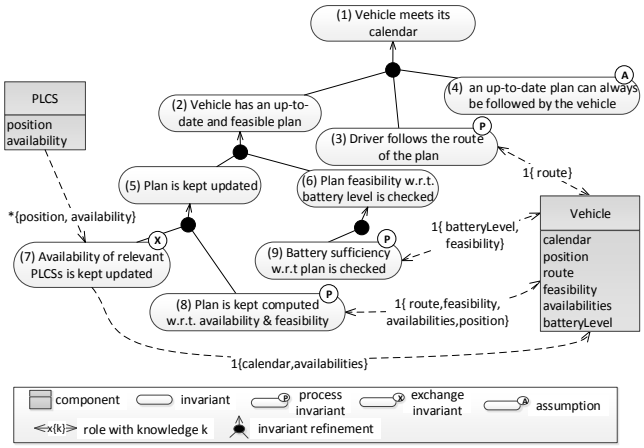

Figure 2. E-Mobility system level graph – IRM method.

or rather they share the same ensemble-level goals or goals with very similar characteristics.

As an example, the *goal-oriented component* pattern considers autonomic components with internal control loops. Goal oriented behavior is explicit in the actions determined by the control loop, which can select actions to actively pursue goals in an adaptive way. A pattern of this kind, in which the feedback loop is embedded within the component, is suitable for those components that have very specific goals, that no other components share.

As another example, the *autonomic service component* pattern is characterized by an explicit external feedback loop. That is, the control loop is realized by "attaching" via appropriate interfaces an external controller that turns a simple service component into a component whose activities can be externally controlled to make the component itself goal-oriented and adaptive.

In a coordinated system for e-mobility, the e-vehicles of a car-sharing company may all share the same basic adaptation goals, thus making it suitable to model them as simple components all sharing the same class of external controller. Also, at the level of the fleet of e-vehicles, the presence of a single stakeholder makes it possible to exploit a pattern of an ensemble with a global control loop to orchestrate the overall behavior of the fleet.

VI. High-level Design – Architecture

In order to guide the design of an ensemble-based system from high-level strategic goals, requirements and patterns (described by SOTA) to their low-level realization in terms of system architecture (components and ensembles) we use the *Invariant Refinement Method* (IRM) [7].

The main idea of IRM is to capture the high-level system goals and requirements in terms of interaction *invariants*. In compliance to SOTA's notion of "striving to achieve", invariants describe the desired state of the system-to-be at every time instant. In general, invariants are to be maintained

by the coordination of the different system components. At the design stage, by *component* we refer to a participant or actor of the system-to-be. A special type of invariant, called *assumption*, describes a condition that is expected to hold about the environment; an assumption is not intended to be maintained explicitly by the system-to-be.

As a design decision, identified top-level invariants are decomposed into more concrete sub-invariants forming a decomposition graph (Figure 2). The decomposition is essentially a refinement, where the composition of the children exhibits all the behavior expected from the parent and (potentially) some more. By this decomposition, we strive to get to the level of abstraction where the (leaf) invariants represent detailed design of the particular system constituents – components, component processes, and ensembles. Two special types of invariants, namely the *process invariants* (denoted by "P") and *exchange invariants* (denoted by "X"), are used to model the low-level component computation (processes) and interaction (ensembles), respectively.

A possible system-level graph corresponding to the simplified e-Mobility scenario is depicted in Figure 2. In this case, the IRM design mainly captures the necessity to keep the vehicle's plan updated (invariant (5)) and to check whether the current plan remains feasible with respect to measured battery level (invariant (6)). The identified leaf invariants are easily mappable to component activities, which are further formally captured by SCEL or SCLP.

VII. Modeling Computational Activities: SCEL

To complement the high-level, architectural design, we have proposed specific linguistic and programming abstractions aiming at dealing with the challenges posed to language designers by massively distributed and highly dynamic systems. Our starting points have been the notions of *autonomic components* (ACs) and *autonomic-component ensembles* (ACEs) that are used to structure systems into independent and distributed building blocks that interact and adapt in different ways. Based on the notions of ACs and ACEs, we have introduced a number of specific abstractions and linguistic constructs that permit building up ACs, defining ACEs and programming their behaviors and interactions. The proposed abstractions are the basis of SCEL (Software Component Ensemble Language) [8], [9].

ACs are entities with dedicated knowledge units and resources that can cooperate while playing different roles. Each AC is equipped with an *interface*, consisting of a set of *attributes*, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc. Attributes are used by the ACs to dynamically organize themselves into ACEs.

Indeed, one of the main novelties of SCEL is the way sets of partners are selected for interaction and thus how ensembles are formed. Individual ACs not only can single out communication partners by using their identities, but they can also select partners by exploiting the attributes in the interfaces of the individual ACs. Predicates over such attributes are used to specify the targets of communication actions, thus providing a sort of *attribute-based* communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates.

Starting from the IRM model presented in Figure 2, we can identify two kinds of SCEL components: PLCSs ad Vehicles. A PLCS identifies a *parking lot/charging station* and is characterized by a *position* and by its *availability*. These are the attributes that are exposed in the component interface and that respectively identify the location of the PLCS and the number of available slots in the area. As expected, Vehicle components identify cars involved in the scenario and will expose in their interface a set of attributes describing the state of the component (*position*, *batterylevel*,...). Attributes of both PLCSs and Vehicles are obtained as the projection on the interface of the local knowledge of each component.

The user associated to a vehicle is modeled by a process that, according to the local Vehicle interface, will interact with PLCSs in order to identify the next stop in the travel. This task is largely simplified thanks to the use of attribute based communication. Indeed, if *poi* is the next *point-of-interest* to visit in the travel, then the next PLCS to use can be identified by sending a *reservation request* to all the PLCSs components that are close to *poi* up-to a given *walking distance* and that can be reached with the current battery level.

However, when the battery level of a vehicle decreases under a given threshold, the actual behaviour can be adapted so to force the reservation of a PLCS that can be used to recharge the battery and then continue with planned trip.

VIII. Adaptation via Soft-Constraints Solving and Optimization

As a complement to SCEL specifically targeting intuitive specification of optimization problems that frequently appear in self-adaptive systems, we have used our approach on Soft Constraint Logic Programming.

Constraint logic programming (CLP) [10] extends logic programming (LP) by embedding constraints in it. However, only classical constraints can be handled. So, in [11], a further extension has been proposed to also handle soft constraints. This has led to a high-level and flexible declarative programming formalism, called *Soft CLP* (SCLP), allowing to easily model and solve real-life problems. Roughly speaking, SCLP programs are logic programs where logical constants and operations are replaced by those of the semiring (a structure representing the levels of satisfiability or the costs of a constraint). Consequently, assignments of variables to the items of the Herbrand universe yield the levels of satisfiability or the costs of the constraints.

We have applied the SCLP framework [12] to the e-Mobility travel optimization problem described in [13], by modelling in Ciao [14][2] two scenarios: the (i) trip; and (ii) journey optimization problems. A solution to (i) finds the best trip in terms of travel time and energy consumption, while (ii) determines the optimal sequence of trips, guaranteeing that the user reaches each appointment in time and that the state

[2]We would like to thank the Clip group for its technical support.

of charge of the electric vehicle never falls below a given threshold.

Besides optimizing trips and journeys of single users, that we can call local problems, the e-Mobility case study aims at solving global problems, involving large ensembles of vehicles. For such large problems, solution is often unfeasible, with both SCLP and more efficient tools. To tackle these, we propose a coordination of declarative and procedural knowledge: the global problem is decomposed into several local problems, which can be separately solved by the SCLP implementation (e.g. [14]), and whose parameters can be iteratively determined by a programmable coordination strategy. The latter guarantees a suboptimal, yet acceptable global solution.

Let us consider for example the parking optimization problem, which consists in finding the best parking lot for each vehicle of an ensemble in terms of three factors: the distance from the current location of the vehicle to the parking lot, the distance from the parking lot to the appointment location and the cost of the parking lot. Solving a global optimization procedure which assigns the best parking lot to each vehicle of the ensemble would be rather expensive, and also not flexible (replanning could require lots of time). So we propose to use SCLP to solve the local problems and some procedural language to programme the orchestrator. In this setting, SCLP is convenient since the orchestrator will be able to access much more easily the parameters of its fact/clause-based declarative implementation than an ordinary imperative module structure.

In particular, the orchestrator could be programmed using an extension of SCEL or simply Java. The orchestrator, after receiving the requests from the vehicles which want to park, asks the SCLP tool to solve the local optimization problems, determining the best parking lot for each vehicle. Then, it verifies if the local solutions all together form an admissible global solution, that is, if each parking lot is able to satisfy the requests of the vehicles planning to park in it. If it is so, the problem is solved, otherwise the orchestrator queries the declarative knowledge again, but now by increasing the costs of the parking lots which received too many requests. The procedure is repeated, with suitable variations, until a global solution is found. Notice that in this way the orchestrator has a hypothetical, transactional behavior, with the options of committing (a solution is found) or partially backtracking (on the parkings which are overfull).

IX. Implementation and deployment

Next steps in the EDLC, following the architectural design and detailed specification of component activities, is implementation and deployment. For these steps, we employ our DEECo (*Dependable Emergent Ensembles of Components*) component model [15] to provide us with the relevant software engineering abstractions that ease the programmers' tasks.

A component in DEECo, features execution model based on the MAPE-K autonomic loop [16]. In compliance with SCEL, it consists of (i) well-defined knowledge, being a set of knowledge items and (ii) processes that are executed periodically in a soft real-time manner. The component concept is complemented by the first-class ensemble concept. An ensemble stands as the only communication mechanism between

```
component Vehicle features AvailabilityAggregator:
  knowledge:
    batteryLevel = 90%,
    position = GPS(...),
    calendar = [ POI(WORKPLACE, 9AM−1PM), POI(MALL, 2PM−3PM), ... ],
    availabilities = [ ],
    plan = { route = ROUTE(...), isFeasible = TRUE }
  process computePlan:
      in plan.isFeasible, in availabilities, in calendar, inout plan.route
    function:
      if (!plan.isFeasible) plan.route ← planner(calendar, availabilities)
    scheduling: periodic( 5000ms )
  ...
ensemble UpdateAvailabilityInformation:
  coordinator: AvailabilityAggregator
  member: AvailabilityAwareParkingLot
  membership:
    ∃ poi ∈ coordinator.calendar:
      distance(member.position, poi.position) ≤ TRESHOLD &&
      isAvailable(poi, member.availability)
  knowledge exchange:
    coordinator.availabilities ← { (m.id, m.availability) | m ∈ members }
  scheduling: periodic( 2500ms )
```

Figure 3. Examples of identified DEECo components & ensembles.

DEECo components. It specifies a *membership* condition, according to which components are evaluated for participation. The evaluation is based on the components' knowledge (their *attributes* in SCEL). An ensemble also specifies what is to be communicated between the participants, that is, the appropriate knowledge exchange function. Similar to component processes, ensembles are invoked periodically in a soft real-time manner. (See Figure 3 for an excerpts of components and ensembles descriptions as found in the e-Mobility case study.)

In order to bring the above abstractions to practical use we have used jDEECo[3] – our reification of DEECo component model in Java. In jDEECo, components are intuitively represented as annotated Java classes, where component knowledge is mapped to class fields and processes to class methods. Similarly, appropriately annotated classes represent DEECo ensembles.

Once the necessary components and ensembles are coded, they are deployed in jDEECo runtime framework, which takes care of process and ensemble scheduling, as well as low-level distributed knowledge manipulation.

X. Evaluation and Related Work

Having described the application of EDLC to the e-Mobility case study, we relate it in this section to other approaches having the same aim and we describe benefits that we have observed in performing the case study. In particular, we structure this section along three main topics addressed in the case study, namely (i) requirements engineering and architectural design, (ii) modeling of activities, (iii) programming and deployment.

As to requirements engineering and architectural design in the area of autonomic systems, the most recognized approaches are KAOS [17] and Tropos [18]. Similar to our approach, they fall into the category of goal modeling and elaboration, especially in the area of agent-based systems. In our experience,

[3] http://github.com/d3scomp/JDEECo

Table I. SUMMARY OF METHODS/TOOLS USED.

Requirements engineering:	SOTA
High-level design:	IRM
Process/activities modeling:	SCEL
Adaptation/optimization modeling:	SCLP
Implementation/runtime:	DEECo / jDEECo

they provide a very solid ground for requirements engineering, but fall short to an extent when continuous control with self-adaptivity (as in the case of e-Mobility case study) is sought for. For this reason, we have employed SOTA and IRM, which are centered around the notion of continuously "striving to achieve" and thus make the reasoning about a guided evolution of a system easier.

As for the activity modeling, our approach builds on the body of work carried out in coordination languages (e.g., KLAIM [19]) and process algebras. However, it extends it by providing a tailored semantics to describe and reason about cooperating groups of components (i.e. ensembles). In the same vein, SCLP builds on the experience with constraint solving, but adds the option of soft-constraints and integration with SCEL. Indeed, in the e-Mobility case study, we found the interplay of SCEL with SCLP very useful for description of mutually related activities of interaction and coordination among vehicles combined with finding a tradeoff between local-global optimums (reflecting the need of harmonizing the selfish and cooperative concerns of vehices in the case-study).

Finally, at the programming and deployment stage, our approach has been backed up by DEECo component model and its Java-based reification jDEECo. In this respect, it is possible to find a plethora of component models and SOA-based approaches (e.g. SCA, Fractal, OSGi). However, these typically fall short in well-defined dynamicity (as captured by the ensembles) as well as in autonomicity and self-adaptation capabilities (as featured by the special design of components as distributed MAPE-K based entities). Similar problems apply even to the agent-based approaches with their Belief-Desire-Intention model (e.g., JADE). On the other hand, the explicit support of DEECo for ensembles and components – based on knowledge and cyclic activities – proved to make the transition from SOTA/IRM-based design (together with activities captured by SCEL/SCLP) to runtime very smooth.

XI. CONCLUSIONS AND FUTURE WORK

We have presented the EDLC framework for development of self-aware autonomic adaptive systems applied to the e-Mobility case study, a driving case-study in the ASCENS FP7 project. We have particularly shown the offline processes of EDLC, starting from requirements modeling and pattern identification with SOTA, to refinement of system invariants with IRM, ending in activity modeling with SCEL and SCLP formalisms. We have also outlined the programming and deployment phases using DEECo/jDEECo. The summary of methods and tools used in provided in Table I.

Due to space constraints and present work organization we have focused on requirements analysis, modeling and programming phases and deployment transaction of EDLC. The further phases (i) verification and validation of functional and non-functional properties at design and runtime and (ii) system evolution, where historical data monitored at runtime are used to improve the system design, are subject of the current and future research.

REFERENCES

[1] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Kwiatkowska, J. Mcdermid, and R. Paige, "Large-scale complex IT systems," *Commun. ACM*, vol. 55, no. 7, pp. 71–77, Jul. 2012.

[2] S. Bensalem, T. Bures, J. Combaz, N. Koch, R. D. Nicola, M. Hölzl, M. Loreti, P. Tuma, M. Wirsing, and F. Zambonelli, "A Life Cycle for the Development of Autonomic Systems," 2013, submitted.

[3] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM TAAS*, vol. 4, no. 2, pp. 1–42, 2009.

[4] J. Mylopoulos, L. Chung, and E. S. K. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM*, vol. 42, no. 1, pp. 31–37, 1999.

[5] D. B. Abeywickrama and F. Zambonelli, "Model Checking Goal-Oriented Requirements for Self-Adaptive Systems," in *Proc. of ECBS*. IEEE, Apr. 2012, pp. 33–42.

[6] G. Cabri, M. Puviani, and F. Zambonelli, "Towards a Taxonomy of Adaptive Agent-based Collaboration Patterns for Autonomic Service Ensembles," in *Proc. of CTS*. IEEE, May 2011, pp. 508–515.

[7] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, and N. Hoch, "Design of Ensemble-Based Component Systems by Invariant Refinement," in *Proc. of CBSE '13*. Vancouver, Canada: ACM, 2013.

[8] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "SCEL: a Language for Autonomic Computing," IMT Lucca, Tech. Rep., January 2013. [Online]. Available: http://rap.dsi.unifi.it/scel/pdf/SCEL-TR.pdf

[9] R. De Nicola, G. L. Ferrari, M. Loreti, and R. Pugliese, "A Language-Based Approach to Autonomic Computing," in *Revised Selected Papers of FMCO*. Springer, 2011, pp. 25–48.

[10] J. Jaffar and J. L. Lassez, "Constraint Logic Programming," in *POPL*. ACM Press, 1987, pp. 111–119.

[11] S. Bistarelli, U. Montanari, and F. Rossi, "Semiring-Based Constraint Logic Programming: Syntax and Semantics," *ACM TOPLAS*, vol. 23, no. 1, pp. 1–29, 2001.

[12] G. V. Monreale, U. Montanari, and N. Hoch, "Soft Constraint Logic Programming for Electric Vehicle Travel Optimization," *CoRR*, vol. abs/1212.2056, 2012.

[13] N. Hoch, K. Zemmer, B. Werther, and R. Y. Siegwarty, "Electric Vehicle Travel Optimization - Customer Satisfaction Despite Resource Constraints," in *Proc. of IEEE IVS*. IEEE, 2012.

[14] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla, "The Ciao Prolog System. Reference manual," School of Computer Science, Technical University of Madrid (UPM), Tech. Rep. CLIP3/97.1, 1997.

[15] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "DEECo – an Ensemble-Based Component System," in *Proc. of CBSE '13*. Vancouver, Canada: ACM, 2013.

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

[17] A. V. Lamsweerde, "Requirements Engineering: from Craft to Discipline," in *SIGSOFT '08/FSE-16*. ACM, 2008, pp. 238–249.

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

[19] R. De Nicola, G. Ferrari, and R. Pugliese, "KLAIM: A Kernel Language for Agents Interaction and Mobility," *Software Engineering, IEEE Transactions on*, vol. 24, no. 5, pp. 315–330, 1998.