Constructing Autonomous Systems: Major Development Phases

Nikola Šerbedžija
Fraunhofer FOKUS,
Berlin, Germany,
Email: nikola.serbedzija@fokus.fraunhofer.de

Annabelle Klarl, Philip Mayer
Ludwig-Maximilians-Universität München
Munich, Germany
Email: {klarl|mayer}@pst.ifi.lmu.de

Abstract—**Developing autonomous systems requires adaptable and context aware techniques. The approach described here decomposes a complex system into service components – functionally simple building blocks enriched with local knowledge attributes. The internal components' knowledge is used to dynamically construct ensembles of service components. Thus, ensembles capture collective behavior by grouping service components in many-to-many manner, according to their communication and operational/functional requirements. To achieve such high level of dynamic behavior a complete development life cycle for ensemble based systems has been defined and supported by rigorous analyses and modeling methods, linguistic constructs and software tools. We focus here on the analysis, modeling, programming and deployment phases of the autonomous systems development life cycle. A strong pragmatic orientation of the approach is illustrated by two different application scenarios. The main result of this work is an integrated view on developing autonomous systems in diverse application domains.**

Keywords-autonomous systems, component-based systems, context-aware systems

I. INTRODUCTION

Developing massively distributed systems has always been a grand challenge in software engineering [1], [2], [3], [4]. Incremental technology advances have continuously been followed by more and more requirements as distributed applications grew mature. Nowadays, one expects a massive number of nodes with highly autonomic behavior still having harmonized global utilization of the overall system. Our everyday life is dependent on new technology which poses extra requirements to already complex systems: we need reliable systems whose properties can be guaranteed; we expect systems to adapt to changing demands over a long operational time and to optimize their energy consumption [5], [6].

One engineering response to these challenges is to structure software intensive systems in ensembles featuring autonomous and self-aware behavior [7], [8]. The major objective of the approach is to provide formalisms, linguistic constructs and programming tools featuring autonomous and adaptive behavior based on awareness. Furthermore, making technical systems aware of their energy consumption contributes significantly to ecological requirements, namely to save energy and increase overall system utilization.

The focus here is to integrate the functional, operational and energy awareness into the systems providing autonomous functioning with reduced energy consumption. The rationale, expressing power and practical value of the approach are illustrated on the e-mobility and cloud computing application domains. The two complex domains appear to be fairly different. However, taking a closer look at the requirements of the two scenarios it becomes noticeable that the problem domains share numerous generic system properties, especially when seen from the optimized control perspective.

The paper presents integrative work focusing on the development lifecycle of complex distributed control systems. It binds together methods and techniques to model and construct systems with service components and ensembles. The rationale and development lifecycle (Sections II) of the approach is presented through close requirements analysis (Section III), ensemble modeling (Section IV) and programming/deployment (Sections V and VI) of two concrete application scenarios. A strong pragmatic orientation as well as the general nature of the approach is shown on two different case studies. Finally, the approach is summarized giving further directions for the work to come in Section VII.

II. DEVELOPMENT LIFECYCLE

The engineering of autonomous systems includes all of the challenges of non-autonomous complex systems plus the added problem of achieving self-* properties allowing for autonomy. An autonomous system needs to be self-aware and self-adaptive. That means it has to maintain the knowledge of its own functional and operational requirements and it should be capable of performing appropriate changes without human intervention. To implement such behavior, a number of feedback loops within the system are needed, to deal with changes both in the controlled environment and the system per se.

The method to develop autonomous systems thus needs to focus more on the runtime side than traditional engineering approaches, as both outside changes and system adaptive responses happen in operation (i.e., live). Within this approach, the iterative development processes that are standard in industry today have been extended to include two main loops; one focuses on the design time and one on the runtime of the system. Both loops are connected to allow feedback. The resulting Ensemble Development Life Cycle (EDLC) is shown in Figure 1. The first loop is the design loop which begins with the analysis of requirements, continues on to the modeling and programming phase, and finally to the verification and validation of the system. This loop runs iteratively until the result is satisfying. A connecting arrow which corresponds to the deployment of the system leads to the second loop which represents the runtime of the system. Usually, an ensemble-based system stays within this loop once deployed, using a feedback loop to achieve adaptation. This loop consists of a monitoring phase (both of the environment and itself), awareness built on the monitored information, and finally self-adaptation which leads back to monitoring.

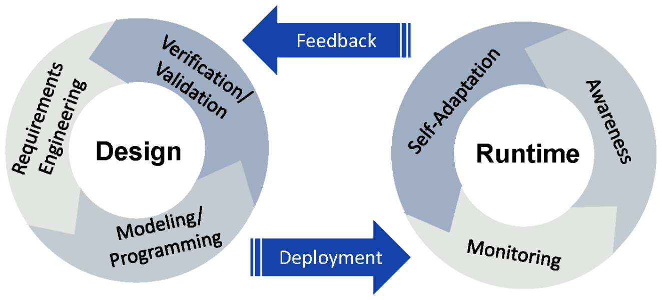

Figure 1. The Ensemble Development Life Cycle

The feedback arrow back to the design phase has two functions. First, feedback of normal system situations can be fed back to design to tweak the system or as input for a next version. Second, if a critical system encounters a non-adaptable situation, the feedback can be given immediately to human operators with the ability to reconfigure.

For each of the phases and transitions within the development life cycle a number of tools and methodologies have been developed, allowing for a formal and rigorous development of complex distributed autonomous systems [9].

Within this paper, we focus on two practical applications which are illustrated on the requirements and modeling phases, followed by programming and deployment. The other phases (validation and verification, awareness, monitoring, and self-adaptation) and the feedback transition are beyond the scope of this paper.

III. Requirements

To explore the system requirements, two complex application domains are closely examined: e-mobility control and cloud computing.

E-mobility is a vision of future transportation by means of an electric vehicles network allowing people to fulfill their individual mobility needs in an environmental friendly manner (decreasing pollution, saving energy, sharing vehicles, etc).

Cloud computing is an approach that delivers computing resources to users in a service-based manner, over the Internet, thus reinforcing sharing and reducing energy consumption).

At a first glance electric vehicular transportation and distributed computing on demand have nothing really in common!

A. Common Characteristics

In a closer examination the two systems, though very different, have a number of common characteristics.

1) Massive Distribution and Individual Interest: E-mobility deals with managing a huge number of e-vehicles that transport people from one place to another taking into account numerous restrictions that the electrical transportation means imposes.

Each cloud computing user has also his/her individual application demands and interest to efficiently execute it on the cloud. The goal of cloud computing is to satisfy all these competing demands.

Both applications are characterized by a huge number of single entities with individual goals.

2) Sharing and Collectiveness: In order to cover longer distances, an e-vehicle driver must interrupt the journey to either exchange or re-charge the battery. Energy consumption has been the major obstacle in a wider use of electric vehicles. An alternative strategy is to share e-vehicles in a way that optimizes the overall mobility of people and the spending of energy. In other words: when my battery is empty – you will take me further if we go in the same direction and vice versa.

The processing statistics show that most of the time computers are idle – waiting for input to do some calculations. Computers belong amongst the fastest yet most wasteful devices man has ever made. And they dissipate energy too. Cloud computing overcomes that problem by sharing computer resources making them better utilized. In another words, if my computer is free – it can process your data and vice versa; or even better, let us have light devices and leave a heavy work for the cloud.

At a closer look “sharing and collectiveness” are common characteristics of both application domains!

3) Awareness and Knowledge: E-mobility can support coordination only if e-vehicles know their own restrictions (battery state), destinations of users, re-charging possibilities, parking availabilities, the state of other e-vehicles nearby. With such knowledge, collective behavior may take place, respecting individual goals, energy consumption and environmental requirements. Cloud computing deals with the dynamic scheduling of available computing resources within a wider distributed system. Maximal utilization can only be achieved if the cloud is “aware” of the users’ processing needs and the states of the deployed cloud resources. Only with such knowledge a cloud can make a good utilization of computers while serving individual users’ needs.

At a closer look “awareness” of own potentials, restrictions and goals as well as those of the others is a common characteristic. Both domains require self-aware, self-expressive and self-adaptive behavior based on a knowledge about those “self*” properties.

4) Dynamic and Distributed Energy Optimization: E-mobility is based on a distributed network that manages numerous independent and separate entities such as e-vehicles, parking slots, re-charge stations, and drivers. Through a collective and awareness-rich control strategy the system may dynamically re-organize and optimize the use of energy while satisfying users’ transportation needs.

Cloud computing actually behaves as a classical distributed operating system with the goal of maximizing operation and throughput and minimize energy consumption, performing tasks of multiple users.

At a closer look "dynamic and distributed optimization" is an inherent characteristic of the control environment for both application domains.

B. Common Approach

This set of common features serve as a basis for analysis and modeling of such systems leading to a generic framework for developing and deploying complex autonomous systems (Table I). Respecting these characteristics in constructing such systems will help meeting the common requirements and provide major behavioral principles: adaptation, self-awareness, knowledge, and emergence. These principles are actually very close and inter-related: namely knowledge is needed for awareness which is further needed for adaptation which further leads to emergent behavior.

Table I. COMMON CHARACTERISTICS

Common feature	Cloud computing	E-Mobility
Single entity	Computing resource	Car, passenger, parking lot, charging station
Individual goal	Efficient execution	Individual route plan
Ensemble	Application, CPU pool	Free vehicles, free parking lots, etc.
Global goal	Resource availability, optimal throughput	Travel, journey, low energy
Self-awareness	Available resources, computational requirements, etc.	Awareness of own state and restrictions
Autonomous and collective behavior	Decentralized decision making, global optimization	Reaching all destinations in time, minimizing costs
Optimization	Availability, computation task, execution	Reaching destination in time, vehicle/infrastructure usage
Adaptation	According to available resources	According to traffic, individual goals, infrastructure, resource availability
Robustness	Failing resources	Range limitation, charging battery, traffic resources

In this approach the adaptation is modeled as progress in a multi-dimensional space where each axis represents one aspect of system awareness (knowledge about its own functional, operational, or other states). Adaptation actually happens when the system state moves from one to another position within the space according to the pre- and post-condition on each of its awareness dimensions. This adaptation model is called SOTA (State Of The Affairs) [10].

The trajectory of an entity in the SOTA space is illustrated in Figure 2. Defining certain states in the SOTA space as desired goal states helps to understand and model goal-directed behavior of entities. We determine three kinds of (sets of) states: pre-conditions G^{pre}, utilities U, and post-conditions G^{post}. Starting from the desired state, G^{post} is the goal state that the entity should reach. This goal is only activated when the entity moves to the state G^{pre} so that it now has to try to move to G^{post}. On the way through the state space, the entity may have to adhere to specific constraints which are called utilities U.

The SOTA adaptation model is used to extract major application requirements and offers appropriate adaptation patterns

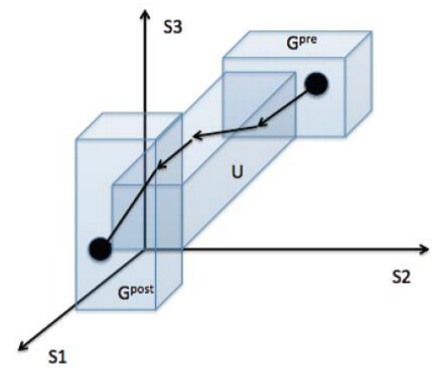

Figure 2. SOTA Adaptation Model

that effectively compose the system into numerous adaptation loops and guarantees the required behavior at run-time (more details on SOTA approach are described elsewhere, see [8] and [10]).

IV. ENSEMBLE MODELING

Control systems for both the cloud computing and e-mobility domains share the idea of groups of entities collaborating towards specific goals. Those groups are formed dynamically while each group exhibits a collective and goal-directed behavior on the basis of complex interactions between members of the group. Well-known techniques in component-based engineering [11] are not enough to capture the particular characteristics of those highly dynamic and collaborative systems. Component-based modeling merely determines the architectural and dynamic properties of the underlying system while ensemble modeling focuses more on the cooperative features on top of the component-based models.

The HELENA approach [12] provides the formal foundations for rigorous ensemble modeling. Each group of entities collaborating towards a goal is abstractly modeled as an ensemble. The specific functionalities and interaction abilities in the ensemble are captured in roles played by the components, and connectors between those roles. The first distinguishing feature of the HELENA approach is the ability of components to dynamically adopt and give up roles. This feature supports adapting to changing conditions since appropriate components can contribute to the ensembles on demand. It also increases robustness since defective members can easily be replaced by new components taking over responsibility for an abandoned role. Lastly, it also helps to efficiently use resources since roles can be given up as soon as they are no longer needed. The second distinguishing feature is that components can adopt multiple roles in parallel so that they may play different roles concurrently in the same or different ensembles. Thus, the components of a single component-based system may take part in multiple collaborations playing task-specific roles in each group – or the other way round, that is multiple ensembles perform their tasks building on top of the same resources of the underlying system.

Figure 3 shows a snapshot of a component-based system on which two ensembles are imposed as described in the HELENA approach. The bottom level shows the pool of all components available in the system. They provide the core capabilities which are commonly available for all tasks. The components can be of different types, for example in the e-mobility scenario the basic components are persons, cars, and parking lots. However, they can also be of the same type as in the cloud computing scenario. The two upper levels of Figure 3 show different ensembles. Each requires specific roles to collaborate (here, we just model two roles R and R'). Each role provides additional capabilities which are required while performing the role-specific task. When a component adopts a certain role – depicted by an arrow in Figure 3 – it also adopts the role-specific capabilities. For example, a person may adopt the role of the driver of a car and thus gain the ability to decide on the velocity of the car, while in the role of a passenger she can only hop on and off.

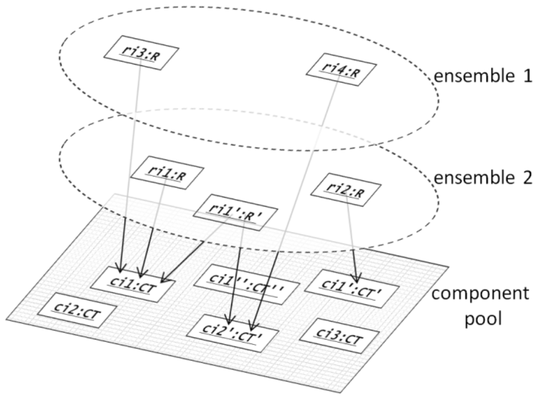

Figure 3. Ensembles in the HELENA Approach

At the same time, one component may participate in several ensembles playing the same or different roles. In each role, the component is equipped with particular properties and capabilities in addition to its core capabilities. Thus, these properties are the key features in the HELENA approach to enable task-oriented awareness. For example, adopting the role of the driver provides a person with the permission to retrieve the battery status of the vehicle. With this knowledge about the battery she can adapt her route accordingly.

Ensemble modeling on top of a component-based model is especially useful as a basis for subsequent development phases. Modeling with roles allows concentrating on the capabilities needed for a specific task. This increases coherence which leads to cleaner ensemble architectures. Furthermore, it decreases complexity of the models, thus providing a well-defined foundation for verification and validation as well as for detailed component-based designs implementing the required ensemble architecture.

Ongoing research in HELENA currently focuses on the derivation of component-based designs from ensemble architectures, description of ensemble behavior based on interacting roles, and checking goal satisfaction. For implementing ensemble architectures we investigate a systematic transition from HELENA models to abstract programming languages like SCEL [9] (cf. Section V-A).

The HELENA approach helps us to develop application scenarios based on top of a common basic component model. In both of the discussed application domains (e-mobility and cloud computing), basic components provide the core capabilities such that we can build appropriate ensemble architectures for each scenario exploiting and enhancing the underlying model.

A. Modeling E-Mobility with Ensembles

In the e-mobility domain, persons, cars, and parking lots team up in ensembles to manage a fleet of cars serving travelers' needs. Scenarios range from journey planning and execution to management of the car park. For example, in the "journey scenario", we envisage an ensemble structure enabling a group of people to travel to (maybe different) destinations (cf. Figure 4).

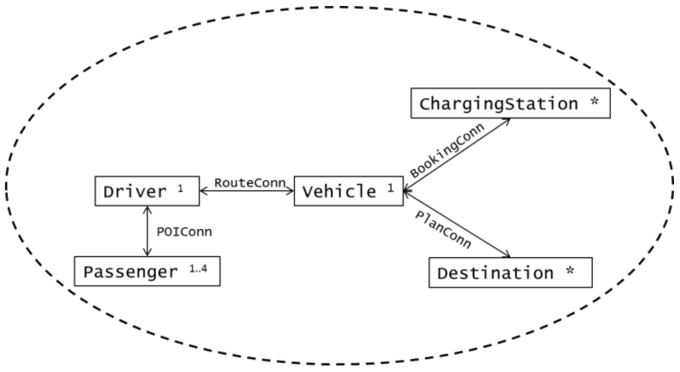

Figure 4. Ensemble Structure for the "Journey Scenario"

To implement the scenario, five roles of different multiplicities need to collaborate. The most important participants are the vehicle and the driver of the vehicle. They communicate with each other via the connector `RouteConn` to exchange, for example, destination requests and route points. It is crucial to recognize that the ensemble structure only mentions the particular roles that are needed for the collaboration and not the underlying components. The roles capture the role-specific capabilities; for example the need for a driver's license (driver) or passenger seats (vehicle). Particular components assume those roles, e.g., a person adopts the role of the driver, but in the case of self-driving cars a computer could also steer the vehicle.

Additionally, up to four passengers may join the collaboration who want to travel to some destinations and therefore communicate with the driver (via the connector `POIConn`) to announce their target locations. Usually, one of the passengers will also be the driver of the vehicle. This is where we benefit from the clear separation of roles and components. In the HELENA approach, one person is able to take different roles at the same time: on the one hand, she is a mere passenger just announcing her destination; on the other hand, she takes on responsibility for steering the vehicle. This dualism increases complexity when we try to model both responsibilities simultaneously in one component. Separating them into two roles as proposed in the HELENA approach facilitates the ensemble model by far. The collaboration is completed by an arbitrary number of destinations the passengers want to reach and an arbitrary number of charging stations needed to load the battery of the vehicle. Both roles can be taken by parking lots and are filled by appropriate parking lots on demand.

Note that other ensembles may be considered in parallel to the “journey ensemble”. For example, another ensemble may take care of relocating a car back to its home base after a one-way journey. This calls for an ensemble structure composed of different collaborating roles in comparison to the “journey ensemble”. However, even several instances of the same ensemble structure can run concurrently if we think of multiple journey groups each traveling to different destinations.

B. Modeling Cloud Computing with Ensembles

In a similar manner, computing entities may team up to provide a seamless platform of resources to users. We envision a cloud computing platform that allows executing applications in the cloud on a PaaS level. Managing the cloud system necessitates various collaborative tasks for distributed deployment and fail-safe execution which we want the system to perform in self-organizing teams. In Figure 5, the basic structure of an ensemble for the deployment of an application is given.

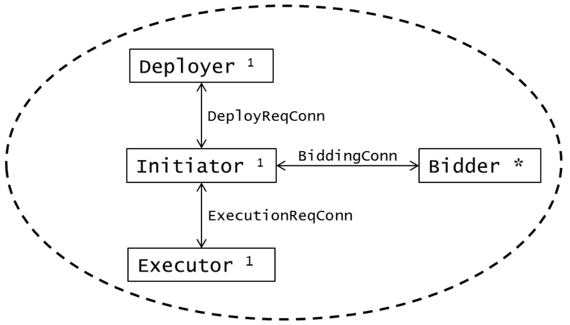

Figure 5. Ensemble Structure for the “Deployment Scenario”

In this scenario, a user wants to deploy an application in the cloud. As the user is outside the network, she needs to address her request to a deployer node inside the network which, from now on, serves as the origin of the request for the collaboration. The application comes with a set of requirements for the executing node such that the cloud has to search for an appropriate node. This search is managed by the initiator. The initiator has three responsibilities: it announces the application with its requirements for execution in the network, calls for bids from possible execution nodes, waits for bids and finally selects one node to serve as executor for the current application. The possible execution nodes – and therefore the bidders for execution – are a highly dynamic and application-specific set of nodes which cannot be determined from the available resources beforehand. This is where the notion of a role facilitates the description of collaboration. The role of a bidder abstractly defines that any node adopting this role has to meet the requirements of the application to be executed. By adopting the role, the node assumes the behavior of the bidder role such that it is only equipped with the appropriate bidding capabilities on demand. This applies to the role of the executor as well. The ensemble structure simply defines that an executor is needed for this task; the role of the executor is then filled dynamically at runtime, the chosen node adopting the appropriate behavior for execution of the application.

The above example of ensembles which take care of executing an application, shows that multiple ensembles of the same structure can be run in parallel, sharing resources through different collaborations. For each application to be run in the cloud, a new collaboration needs to be established. However, nodes can join different ensembles at the same time. For example, a node can be initiator for one application while being the executor for another application. It can also adopt the same role twice in different ensembles, e.g., executing two different applications in two different ensembles. The same node can even take responsibility for two roles in the same ensemble like being initiator and executor at the same time.

V. E-Mobility Deployment

Finding a way from high-level modeling to development and deployment of software intensive systems is a complex endeavor. Reasoning and validation often require high-level abstractions, while implementation calls for detailed programming and low-level deployments. To bridge this gap a number of intermediate tools are being developed that assist in the engineering process [8].

A. SCEL Language Programming Abstractions

The challenge for developers of complex distributed systems is to find proper linguistic abstractions to cope with individual vs. collective requirements of system elements and their need to respond to dynamic changes in an autonomous manner. A set of semantic constructs has been proposed [9], [13] that represent behaviors, knowledge and composition supporting programming of awareness-rich systems. It provides linguistic abstractions for describing the behavior of a single component as well as the formation of ensembles.

The basic ingredient of SCEL – Software Component Ensemble Language – is the notion of an autonomic component $\mathcal{I}[\mathcal{K}, \Pi, P]$ that consists of:

- An interface $\mathcal{I}$ providing structural and behavioral information about the autonomic component in the form of attributes visible to other components.
- A knowledge repository $\mathcal{K}$ managing information about the component interface, requirements, major state attributes etc. Managing such knowledge allows for self-aware behavior and dynamic interlinking with other system components.
- A set of policies Π which controls internal and external interaction.
- A set of processes P defining component functionality specific to the application and internal management of knowledge, policies, and communication.

The structure and organization of the SCEL notation is illustrated in Figure 6.

The code in Table II shows a fraction of the SCEL syntax (with notation for S - systems, C - components, P - processes, a - actions and c - targets); a fully detailed presentation of SCEL syntax and semantics can be found in [9], [13], [14].

SCEL aggregates both semantics and syntax power to express autonomic behavior. At one side, being abstract and rigorous SCEL allows for formal reasoning about system behavior; at another, it needs further programming tools to

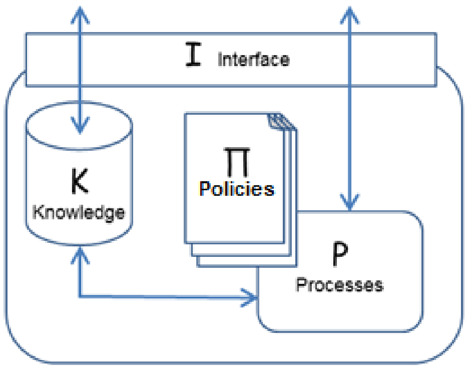

Figure 6. SCEL Elements

Table II. SCEL SYNTAX

SYSTEMS:	S	$::=$	$C \mid S_1 \parallel S_2 \mid (\nu n)S$
COMPONENTS:	C	$::=$	$\mathcal{I}[\mathcal{K}, \Pi, P]$
PROCESSES:	P	$::=$	$\mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[P_2] \mid X \mid A(\bar{p})$
ACTIONS:	a	$::=$	$\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$
TARGETS:	c	$::=$	$n \mid x \mid$ self $\mid$ P $\mid$ p

support system development and deployment. Formal reasoning, modeling and validation are covered in referenced articles about SCEL. Here, the focus is more on the pragmatic orientation on the given application scenario.

B. Java Framework for SCEL Programming and Model Checking

To execute SCEL programs, the jRESP framework has been developed. jRESP is a Java runtime environment providing means to develop autonomic and adaptive systems without any centralized control programmed in SCEL [15]. By relying on the jRESP API, a programmer can embed the SCEL paradigm in Java applications.

A prototypic statistical model-checker running on top of jRESP simulation environment has been implemented. Following this approach, a randomized algorithm is used to verify whether the implementation of a system satisfies a specific property with a certain degree of confidence. The statistical model-checker is parameterized with respect to a given tolerance t and error probability p. The used algorithm guarantees that the difference between the computed values and the exact ones is greater than t with a probability lower than p.

The model-checker included in jRESP can be used to verify reachability properties. These properties allow one to evaluate the probability to reach, within a given deadline, a configuration where a given predicate on collected data is satisfied [15].

C. Programming E-Mobility

Programming in jRESP is self-explaining and elegant. Considering the e-mobility application, we can easily program a vehicle supporting a user in her daily travel obligations. The vehicle is controlled by four modules:

```
VEHICLE =
ContactParkingLots[Planner[Book[MonitorPlanExecution]]]
```

Each of the modules has its own responsibility. The module `ContactParkingLots` retrieves all appointments of the passenger and searches for parking lots close to the points of interest. Afterwards, the module `Planner` plans a route how to reach all appointments in time using the available parking lots. Booking of the appropriate parking lots is done by the module `Book`. At last, the module `MonitorPlanExecution` monitors the current progress on the route and displays information about the reservation of the next parking lot on the route. The jRESP code (enriched with comments explaining the processes) is given in the following boxes.

```
ContactParkingLots =
//read the size of the calendar
//(i.e., the list of appointments)
qry("calendarSize", ?n)@self .
//scan the calendar
for(i := 0 ; i < n ; i ++){
  //read an appointment of the calendar
  qry(calendar, i, ?poi, ?poiPos, ?when, ?howLong)
    @self .
  //contact the parking lots near to the POI
  //(this illustrates attribute-based communication
  //typical in SCEL)
  put("searchPLot", self, poi)
    @{ I.type="PLot" & walkingDistance(poiPos,I.pos)} .
    //ensemble predicate
}
//signal completion of the phase of data request
// from the parking lots
put("dataRequestSent")@self
```

```
Book =
//wait for the completion of the planning phase
get("planningCompleted")@self .
//read the size of plan list
//(i.e., the PLots to be booked)
get("planListSize", ?n)@self .
//scan the plan
for(i := 0 ; i < n ; i ++){
   //read an entry of the plan list
   get("plan", i, ?pLot, ?when, ?howLong)@self .
   //send the booking request to the PLot
   put("book", self, when, howLong)@pLot .
   //wait for the reply of pLot
  //   (we assume that booking requests always succeed)
   get("bookingOutcome", true)@self .
   //store the reservation in the list of reservations
   put("reservation", i, pLot, when, howLong)@self .
}
//close the list of reservations
put("reservationListSize", n)@self .
//signal completion of the booking phase
put("bookingCompleted")@self
```

```
Planner =
//wait for the completion of the phase of
//requirement of data to the parking lots
get("dataRequestSent")@self .
  //we intentionally leave this process unspecified
    //input: collection of tuples of the form
    // (poi,pLotId,pLotInfo) received from the PLots
    //output: list of chosen planned PLots,
    // i.e., (planListSize,n)
    // (plan,0,pLotId0,when0,howLong0)...(plan,n-1,...)
   //signal completion of the planning phase
put("planningCompleted")@self
```

```
MonitorPlanExecution =
//wait for the completion of the booking phase
get("bookingCompleted")@self .
//read the size of the reservation list
//   (i.e., the PLots to be visited)
get("reservationListSize", ?n)@self .
//scan the reservation list
for(i := 0 ; i < n ; i ++){
   //read a reservation
   get("reservation", i, ?pLot, ?when, ?howLong)@self .
   //display the info about the next reservation
   //to the user
   put("reservation", ?pLot, ?when, ?howLong)@screen .
   //wait for the arrival at the parking lot
   //    (signaled by the user)
   get("arrivedAt", pLot)@self .
}
//signal completion of the plan execution phase
put ("planExecuted")@self
```

The jRESP processes illustrate the expressive power of the language to cope with huge systems with complex interactions. A distinguishing feature of SCEL which is directly implemented in jRESP is implicit ensemble building by attributed-based communication. For example, while searching for parking lots close to a particular point of interest in the module ContactParkingLots, an ensemble of appropriate parking lots is implicitly formed by using the following predicate:

```
I.type="PLot" &  walkingDistance(poiPos,I.pos)
```

This directly addresses the search request to the appropriate parking lots.

VI. Science Cloud Deployment

Cloud computing refers to provisioning resources such as full machines, storage space, processing power, or even applications to consumers “on the net”: Consumers can use these resources without having to install hard- or software themselves and can dynamically add and remove new resources. Common use cases include renting virtual machines, external disk space, or ready-made applications for traditional office tasks. Cloud solutions are software products which offer this ability. They may be installed by dedicated cloud companies which only offer the cloud end results to users; however, a company working in a non-IT branch (for example, manufacturing) can also install a cloud solution in-house, thus creating a private cloud for its own employees. The same applies to universities and research institutions.

A. A Voluntary, Peer-to-Peer Platform as a Service

In the science cloud case study [16], [17], the focus is on implementing a cloud in a fully distributed, peer-to-peer, voluntary computing fashion. The cloud is intended for use by the scientific community; each scientist – or university – can contribute to the cloud with computing power and storage space, but can also retract their resources if they are required elsewhere which corresponds to the voluntary aspect of the cloud. Furthermore, there is no centralized control in the cloud; rather, individual nodes communicate in a peer-to-peer fashion to organize themselves.

The cloud itself offers services on a PaaS level, that is, it provides a platform for executing applications. Each application may have its own requirements (or service level agreement) which the cloud must do its best to satisfy while in general keeping all applications running and conserving energy. These aspects make such a distributed cloud-based systems complex and hard to design, build, test, and verify.

To take part in the science cloud, each partner must run an instance of the Science Cloud Platform (SCP). Such an instance, running on a physical or virtual machine, is considered to be a service component in the previous described sense.

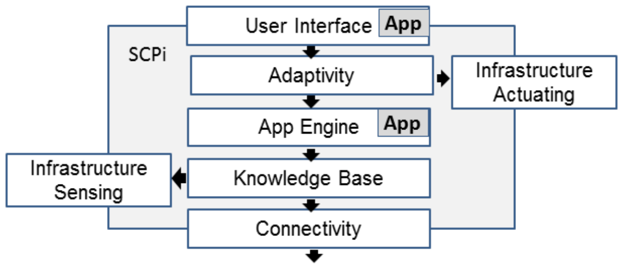

Figure 7.   Science Cloud Platform Architecture

Figure 7 shows the logical components which make up a Science Cloud Platform instance (SCPi). We explore three of them in more detail here: Connectivity, Knowledge, and Adaptivity.

a) Connectivity: Each SCPi has a connectivity component which enables it to talk to other SCPs over the network, and deals with the overlay structure the cloud imposes on the lower-level network layers. The protocol followed by these communications must enable SCPis to find one another and establish links, for example by manually entering a network address or by a discovery mechanism. Furthermore, SCPis must be able to query others for knowledge and at the same time distribute their own knowledge. Finally, the protocol must support exchange of data and applications.

There are different options for implementing such an overlay which in this case is built on top of the TCP/IP (Internet) network. As pointed out above, the science cloud takes a peer-to-peer approach to communication, and thus re-uses classical algorithms for peer routing (for example, DHT-based protocols like Pastry [18] are useful in this context).

b) Knowledge: Each SCPi has knowledge consisting of (1) its own properties (set by developers), (2) its infrastructure

(CPU load, available memory), and (3) other SCPis (acquired through the network). Since there is no global coordinator, each SCPi must build its own view and act upon the available knowledge. The SCPi may acquire knowledge about its infrastructure using an infrastructure sensing plug-in which provides information about static values, such as processor speed, available memory, available disk space, number of cores etc. and dynamic values, such as currently used memory, disk space, or CPU load.

SCPi properties are important when specifying conditions as Service Level Agreements (SLAs) [8], [16] for the applications. For example, when looking for a new SCPi to execute an application, low latency between the SCPis might be interesting. Other requirements may be harder: For example, an application may simply not fit on a SCPi because of the lack of space whereas another may require a certain amount of memory.

c) Adaptivity: As already outlined in the Ensemble Development Life Cycle (EDLC) in the second section of this paper, monitoring, awareness, and self-adaptation are key to managing autonomous systems. For this reason, each SCPi contains its own adaptivity component which uses the information available in the knowledge base.

Adaptation in the science cloud means several things. Applications will be deployed on the cloud and, depending on their SLA, must be executed on nodes which are able to fulfill these. If nodes become overloaded, or leave the system (which they may do at any time), applications need to be moved to different machines and restarted. This requires planning ahead for such situations, i.e., keeping redundant copies of both the applications' executable code and its data. The science cloud may, as indicated in Figure 7, work with an additional IaaS solution below it which allows the cloud to start new virtual machines and migrate to these machines if necessary. In the other direction, using such an IaaS allows shutting down machines if idle, thus conserving energy.

The adaptivity logic is exchangeable, application-independent, and has a direct relation to the SLAs of applications. The adaptivity logic itself can be written in a standard programming language or custom domain-specific languages or rules. It may take into consideration elements such as the reputation of nodes (previously established through their uptimes or capabilities), past performance, peak times experienced, and so on. Besides the connectivity, knowledge, and adaptivity components, each SCPi contains components for sensing the environment (for example, load, attached storage devices, etc.) and for acting on it (in the case an IaaS solution is available). Furthermore, an application engine executes applications locally; both the application interfaces and the SCPi meta-interface are available through a user interface component.

The science cloud is formed by connecting multiple SCPis together over a network. Within this cloud, we consider a subset of SCPis with certain properties as an ensemble which we call a Science Cloud Platform Ensemble (SCPe). The set of properties may be based on attributes of the SCPis and/or the SLAs of applications. In other words, an ensemble consists of SCPis which work together to run one application in a fail-safe manner and under consideration of the SLA of that application, which may require a certain number of SCPis, certain latency between the parts, or have restrictions on processing power or on memory. An example of a science cloud with five SCPis grouped in two (overlapping) ensembles is shown in Figure 8.

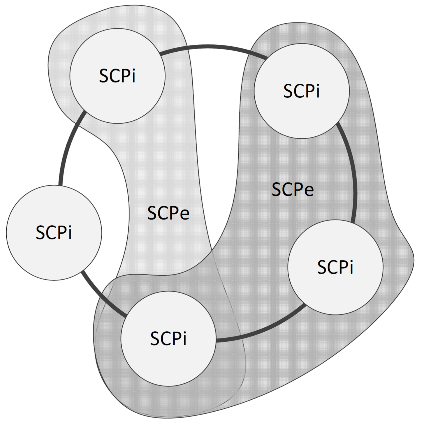

Figure 8. SCP Ensemble

At runtime, an ensemble may gain new SCPis or lose them depending on the behavior of the SCPis themselves; the load generated by applications, and the physical state of the underlying node (which may join and leave the network).

B. Programming the Science Cloud

Currently, a prototype of a science cloud platform is being developed and tested in a physical network connecting two universities [8]. The experimental platform features ad-hoc and voluntary behavior supporting dynamic re-configuration of physical layers and application migration on an upper level.

High-level SCEL modeling and model checking provide formal means for property proofs while a prototype implementation offers pragmatic means to test deployment and effectiveness of autonomous and self-aware behavior. The prototype we are currently investigating is based as much as possible on existing projects and scientific results. In particular, we are re-using the Pastry peer-to-peer substrate [18] and accompanying protocols for implementing the peer-to-peer and voluntary computing part, and in addition an interpretation of the ContractNET [19] protocol for the upper layer of application execution.

Our prototype is split into three layers which correspond roughly to handling the network addressing logic, data management, and application execution.

The first, i.e., the network layer, is based on the Pastry protocol. This protocol uses a hash-based addressing scheme similar to that of a Distributed Hash Table (DHT): Each node is assigned a random hash within a certain (wrapped) range; thus a network position agnostic overlay ring is formed. Routing works by sending messages to a certain hash target; the node whose hash is closest to the target hash receives the message. While routing is possible along the ring, Pastry also introduces shortcuts for reaching the target in $O(logn)$ routing steps. It is important to note here that for each conceivable hash, exactly one node is the closest node which is an interesting property exploited in upper layers. Pastry includes mechanisms for self-healing in case nodes drop out of the overlay network, and automatically integrates newly arriving nodes. Since Pastry only supports unicast routing, the SCRIBE protocol [20] is added on top which provides multicasting support using a form of publish/subscribe mechanism. In the science cloud platform, each node is implemented as a Pastry node.

The second layer, i.e., the data layer, is implemented by the PAST protocol [21] which implements the remaining features for realizing an actual DHT. Thus, it is possible to store data packages given their hash; again, the data is stored at the nearest node. However, PAST also supports redundancy since it allows storing not only one but k copies of a data package clustered around the nearest node. Thus, if the nearest node fails, another automatically takes its place.

The data layer is used in the science cloud platform for storing the applications to be executed (as byte code) as well as the data they keep during runtime. Both must be stored in a redundant fashion.

The application layer deals with the actual execution of applications. Here, we employ an implementation of the ContractNET protocol which is based on a bidding-like process. After deployment of an application by a user, an initiator node is chosen (based on hash nearness of the application code data package) which is from now on responsible for the execution of the application. Note that if this node drops out of the network, another node takes its place automatically according to changing hash nearness. The initiator will now request bids from other nodes through a SCRIBE-based communication channel, sending the application name and requirements to enable other nodes to evaluate whether they are capable of executing the app. This process is shown in Figure 9.

Having received all bids, the initiator decides on an executor node and sends further instructions to this node. The initiator then switches to a monitoring mode: If the executor fails, the initiator starts a new bidding process. The SCP implementation is open-source and can be downloaded from the ASCENS web site [8].

VII. Conclusion

This paper presents a unified approach to model, validate and deploy complex distributed systems with massive number of nodes that respect both individual and global goals. The non-centralized character of the approach allows for autonomic and self-aware behavior which is achieved by introduction of knowledge elements and enrichment of compositional and communication primitives with awareness of both system requirements and individual state of the computing entities.

The essence of the approach is to de-compose a complex system into a number of generic components, and then again compose the system into ensembles of service components. The inherent complexity of such ensembles is a huge challenge for developers. Thus, the whole system is decomposed into well-understood building blocks, reducing the innumerable interactions between low-level components to a manageable number of interactions between these building blocks. The result is a so-called hierarchical ensemble, built from service components, simpler ensembles and knowledge units

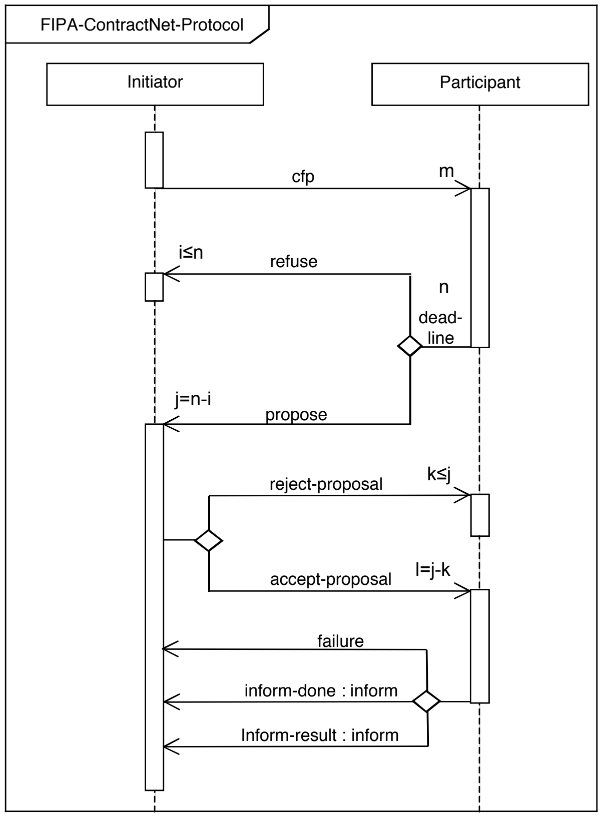

Figure 9. FIPA Contract NET Protocol (from [19])

connected via a highly dynamic infrastructure. Ensembles exhibit four main characteristics: adaptation, self-awareness, knowledge and emergence, yielding a sound technology for engineering autonomous systems [6], [8]. A number of linguistic constructs and validation and programming tools are under development and are being tested in different application scenarios.

This paper presents an integrated view (from high level modeling to application deployment) of a complex approach which has been described by a number of referenced papers, each focusing on different aspects of the work: Modeling ensembles using Helena [12] and SCEL [9], [13], system validation [15], adaptation aspects [10], knowledge management and deployments [10], [16] and engineering aspects [6], [8]. Further contribution of this paper is in optimized and energy-aware control based on autonomous behavior. Optimized distributed control with improved throughput and utilization of the cloud and e-mobility frameworks contribute significantly to the overall strategy to reduce energy consumption. Using the sharing principle instead of exclusive use of the transportation and computing means represents a significant challenge (requiring significant changes in our perception of vehicles and computers) in the application domains under consideration. This principle will undoubtedly play an important role in extending the application area.

Acknowledgment

Most of the work presented here has been performed in the context of the ASCENS project (project number FP7-

257414) [8], funded by the European Commission within the 7th Framework Programme, pervasive adaptation initiative. Special thanks go to the developers of SCEL and jRESP language (Rocco De Nicola from IMT Lucca and his group).

References

[1] N. Serbedzija, “Autonomous Systems: from Requirements to Modeling and Implementation,” in *International Conference on Autonomic and Autonomous Systems (ICAS)*, 2013, pp. 1–6.

[2] The InterLink Project. Accessed: 2013-18-11. [Online]. Available: http://interlink.ics.forth.gr

[3] I. Sommerville, D. Cliff, R. Calinescu, J. Keen, T. Kelly, M. Z. Kwiatkowska, J. A. McDermid, and R. F. Paige, “Large-scale complex it systems,” *Commun. ACM*, vol. 55, no. 7, pp. 71–77, 2012.

[4] M. Hölzl, A. Rauschmayer, and M. Wirsing, “Engineering of software-intensive systems: State of the art and research challenges,” in *Software-Intensive Systems and New Computing Paradigms*, ser. Lecture Notes in Computer Science, M. Wirsing, J.-P. Banâtre, M. M. Hölzl, and A. Rauschmayer, Eds. Springer, 2008, vol. 5380, pp. 1–44.

[5] L. Xu, G. Tan, X. Zhang, and J. Zhou, “Energy Aware Cloud Application Management in Private Cloud Data Center,” in *Proc. Cloud and Service Computing (CSC)*, 2011, pp. 274–279.

[6] C. Seo, “Energy-Awareness in Distributed Java-Based Software Systems,” in *Automated Software Engineering (ASE)*. IEEE Computer Society, 2006, pp. 343–348.

[7] M. M. Hölzl and M. Wirsing, “Towards a System Model for Ensembles,” in *Formal Modeling: Actors, Open Systems, Biological Systems*, ser. Lecture Notes in Computer Science, G. Agha, O. Danvy, and J. Meseguer, Eds., vol. 7000. Springer, 2011, pp. 241–261.

[8] The ASCENS Project. Accessed: 2013-11-18. [Online]. Available: http://www.ascens-ist.eu

[9] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, “SCEL: a Language for Autonomic Computing,” IMT Institute for Advanced Studies Lucca, Italy, Tech. Rep., 2013.

[10] D. B. Abeywickrama, F. Zambonelli, and N. Hoch, “Towards simulating architectural patterns for self-aware and self-adaptive systems,” in *Self-Adaptive and Self-Organizing Systems (SASO) Workshops*. IEEE Computer Society, 2012, pp. 133–138.

[11] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] R. Hennicker and A. Klarl, “Foundations for Ensemble Modeling - The Helena Approach,” in *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi (SAS 2014)*. LNCS, in press.

[13] R. De Nicola, G. L. Ferrari, M. Loreti, and R. Pugliese, “A language-based approach to autonomic computing,” in *Formal Methods for Components and Objects (FMCO)*, ser. Lecture Notes in Computer Science, B. Beckert, F. Damiani, F. S. de Boer, and M. M. Bonsangue, Eds., vol. 7542. Springer, 2011, pp. 25–48.

[14] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, “Meld: A declarative approach to programming ensembles,” in *Intelligent Robots and Systems (IROS)*. IEEE, 2007, pp. 2794–2800.

[15] M. Loreti, “jRESP: a Runtime Environment for SCEL programs,” IMT, Institute for Advanced Studies Lucca, Italy, Tech. Rep., 2013. [Online]. Available: http://rap.dsi.unifi.it/scel/

[16] P. Mayer, C. Kroiss, and J. Velasco, “The Science Cloud Case Study: Overview and Scenarios,” Ludwig-Maximilians-Universitä München, Munich, Germany, Tech. Rep. TR20120500, 2012.

[17] P. Mayer and J. Velasco, “The Science Cloud Case Study: Overview and Scenarios,” Ludwig-Maximilians-Universitä München, Munich, Germany, Tech. Rep. TR20130300, 2013.

[18] A. I. T. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Distributed Systems Platforms (Middleware)*, ser. Lecture Notes in Computer Science, R. Guerraoui, Ed., vol. 2218. Springer, 2001, pp. 329–350.

[19] FIPA Contract Net Interaction Protocol Specification. Accessed: 2013-11-18. [Online]. Available: http://www.fipa.org/specs/fipa00029/SC00029H.html

[20] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. T. Rowstron, “Scribe: a large-scale and decentralized application-level multicast infrastructure,” *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[21] A. Rowstron and P. Druschel, “Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility,” *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 188–201, Oct. 2001.