# Emergent Software Services

Nicolás Cardozo
Systems and Computing Engineering Department, Universidad de los Andes
Bogotá, Colombia
n.cardozo@uniandes.edu.co

## Abstract

Services are normally composed following a structured model, or based on a particular goal that needs to be fulfilled. Such model is problematic for pervasive environments, since service components deployed in the environment are unknown beforehand. As a result, services may never execute due to the unavailability of one of the pre-specified components, or components missing to fulfill the service goal. This paper posits a new vision for service composition by inverting the control flow of service-oriented applications between users and the environment. Rather than having to request a particular service, services emerge from the environment based on interactions between available service components, and are pushed to be utilized by users. We present the architecture required to fulfill our vision in enabling service emergence in a pervasive environment. This vision architecture is realized by an initial prototype framework for software service emergence called Mordor. Early results of this vision are obtained from two examples demonstrating the feasibility of services emergence from previously unknown service components, and a case study demonstrating Mordor's usability in real world scenarios.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

***General Terms*** Design, Languages

***Keywords*** Dynamic service composition, Emergent services, Microservices

## 1. Introduction

The proliferation of sensors and actuators advocated by the Internet of Things (IoT) and Cyber Physical Systems (CPS) is driving a change in both the way users interact with and benefit from software services, and the autonomous interaction between software services [17, 18]. Such proliferation of devices, and the services associated to them will enable the provision of more and better services, for example, by improving services' Quality of Service (QoS) attributes such as response time, load, or availability.

IoT and CPS technologies are becoming of extreme importance in so called smart applications, such as ambient assisted living, smart buildings, or in general, smart city environments. Such environments are characterized by a multitude of sensors, actuators, and human users alike, all seamlessly interacting with each other. Sensor information and actuator capabilities can be used to enhance services deployed in the environment, for example by personalizing services to specific user needs, based on their preferences and information currently available in the environment. An example application domain that benefits from the aforementioned environments is that of routing users in a city. Users in a city are in continuous movement, requiring to travel from $A$ to $B$ to $C$. Users' routes could be affected by real-time traffic information, external sensor information, user preferences, or weather conditions. An example of a multi-modal routing service in a city composed of all such information can be defined as shown in Figure 1.

In order to cope with the requirements of new application domains spawn by such technologies (*e.g.,* multi-modal travel assistants in a smart city) it is necessary to update the current Service-oriented Computing (SOC) composition model. Considerable effort has been put in the composition of software service components defined by distributed providers in pervasive environments [25], as in our routing example. Notably, service composition approaches use a pull request model for service composition, in which services are always requested by users. Recent approaches, using a pull model, increase services' robustness by dynamically replacing service components if they fail or disappear from the environment [8]. These approaches, however, overlook two characteristics inherent to the pervasive environments in which they execute, giving rise to two main problems.

1. *Services must be foreseen.* In the pull service request model, users request a particular service they want to use. That is, software services are composed from service

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

*Onward!'16*, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...$15.00
http://dx.doi.org/10.1145/2986012.2986016

providers following a pre-defined structure between (abstract) service components [1, 31, 36]. Software services are normally composed according to an abstract workflow definition, in which each workflow node represents a service component, and information flows from the outputs of one node to the inputs of another (see Figure 1). While the specific components may be reified at run time by any service provider mapping the expected inputs to the outputs, the workflow structure defines the way in which components interact. Relying on such a workflow structure is problematic in dynamic environments as interaction between service components must be foreseen by developers. Service application developers must assume that all components required to satisfy the requested goal exist, interact in the foreseen way, and are available in the environment. Such composition models mismatch the reality of services deployed in pervasive environments, where mobility, heterogeneity, disconnections, and service failures are common. It is not possible to ensure that a particular service component will exist and have the expected interactions to fulfill a service request. This impairs services usability, as services may never execute (properly). In pervasive environments, service providers' availability is unknown at service definition time, therefore, service definitions cannot assume specific component interactions.

2. *Services are coarse-grained.* Services are seen as full-fledged pieces of software providing a particular functionality end-to-end. Services are normally composed by coarse components offering multiple functional aspects to meet an objective. Such a vision of service components, makes definition of services too rigid, as components' interchangeability is more difficult to achieve. Service components should be defined as independent process with a minimal set of functionality to foster their exchangeability [6, 35]. Available devices in the environment may not have the capabilities to support full services, or even process multiple services. Some devices deployed in the network may only be able to provide very fine-grained functionality according to their sensory capabilities. For example, an ambient sensor may only provide a data stream of the current temperature measurement, rather than a full weather service. Service definition should be maintained as fine-grained as possible in order to increase their flexibility, and foster interaction between available service components in the environment.

To tackle these problems, we propose a *push* service model for pervasive environments, in which services are offered to users from available service component providers in the environment (Section 2). In such model, services accessible by users are guaranteed to be available, increasing services' usability. Additionally, this model reduces the complexity of service components' definitions, as no prior knowledge about other services is required. The push service model is used to support our vision for software services emergence,

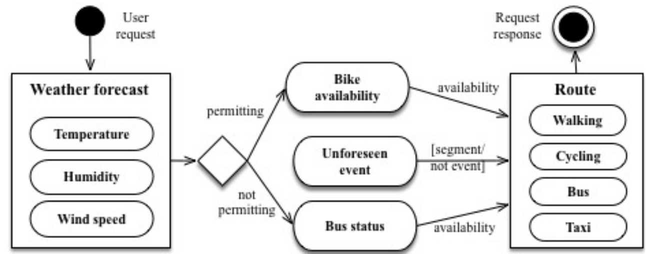



**Figure 1:** Service composition model specification for the routing example.

in which devices deployed in a pervasive environment can be associated with fine-grained service component definitions. From these definitions, new complex services emerge based on the autonomous interaction and composition between service components.

The feasibility of our vision is tested in Section 3 by implementing a prototype framework called *Mordor: where emergent service fellowships converge*. Mordor enables the emergence of software services from the definition of fine-grained stand-alone service component entities that do not require any prior information about their deployment environment or other components' availability. Mordor autonomously enables the broadcasting and discovery of service components, learning the interactions with those components that match in their semantics, inputs, outputs, and QoS attributes. Using components' learned interaction with other components, it is possible to push new emergent services, making them accessible to users. Software services emergence is validated by using two example applications: a color palette chooser, and a poker-hand discoverer. Each of these applications defines multiple service components. We use Mordor to discover all available services in the environment (both, the defined service components and the services that emerge from their interaction). In the simulation of our application examples, we observe emergence by comparing the ratio between the number of services discovered and the number of deployed service components. If it is possible to discover more services than the number of deployed service components, then services have emerged. The usability of our framework is evaluated through a case study that illustrates service components definition and composition in a real-world setting of a routing application for pedestrians in city. The exploration of Mordor's feasibility is complemented by a discussion of challenges and future lines of research to complement our vision (Section 5).

## 2. Software Services Emergence

The software services emergence vision, put forward with Mordor, proposes to change the control flow in SOC applications, so that services are pushed to users from the environment, rather than being user-requested. The vision behind this model is that fine-grained service components

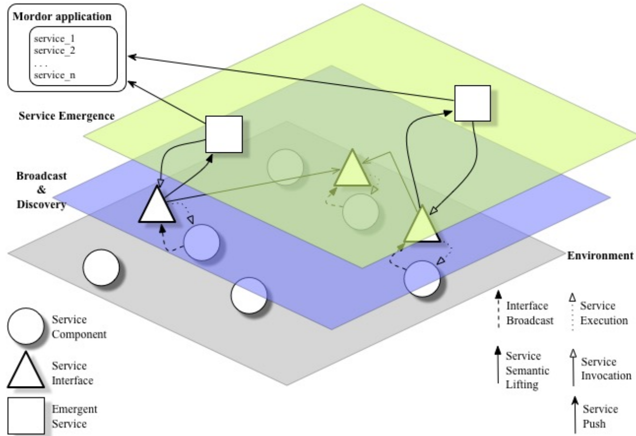



**Figure 2:** Service interaction model in Mordor

defined in a pervasive environment can interact, giving rise to new services from the combination of previously existing ones. We draw inspiration from, and combine three research areas in Mordor, in order to enable the emergence of full-fledged software services. (1) Service components are defined as fine-grained and independent pieces of functionality that *self-organize*. Interaction between service components is autonomous, without requiring previous knowledge about each other [12, 13, 27] (Section 2.1). (2) Service components should be able to compose autonomously by *learning* with which other type of components they can interact. Furthermore, components' learning process can be used to prioritize commonly used services over other possible compositions, so that the environment evolves organically (Section 2.3). (3) Finally, when executing a service, the execution should *adapt* to failing or disappearing components in order to increase services' usability. Adaptations can also be used to gear service emergence for particular situations and users' preferences, increasing the usability of such services [14, 16] (Section 2.4).

The self-organization, learning, and adaptation components in Mordor are coupled as part of the same single programming model. Figure 2 shows how these three components interact. Schematically, we show them in three different layers to help understand how and when each of the components is used. However, readers should keep in mind that they are all coupled. Service components are deployed in isolation in the environment (bottom layer). As part of their definition, components declare their service interface —that is, the functionality the component offers to the outside world. This, interface is used to broadcast and discover other components in the environment. Service components self-organize through interaction with each other's service interface. Interactions between service components are learned so that they can be composed directly in future interactions. Such compositions are lifted as emerging services in the top layer. Once a service emerges, this is pushed to the Mordor application to be used. In the following we explain how each of these components work in more detail.

## 2.1 Service Components Organization

CPS propose a symbiosis between physical elements and software systems by offering software services associated to a set of physical sensors or devices [17]. These environments are not fixed, but rather are in continuous change with the introduction and removal of devices, and the consequent effect on the software services such devices support. As a result, management of software services must be decentralized, where each service component is defined independently and with no prior knowledge about other services available in the environment.

Given that the nature and type of service components deployed in the environment is unknown, to promote interaction, all service components defined in Mordor have the capability to interact with each other. Services are broadcasted to, and discovered in the environment using a zero-configuration service such as multicast Domain Name System (mDNS). In this protocol, every service component deployed in a given environment is made available to all other components already deployed, and vice versa. Upon discovery, service components start an interaction process to see whether they can be combined into bigger services. As service components are defined in isolation without prior knowledge about other components, and may even be developed by different providers, interaction between components must start by communicating the nature (*i.e.,* semantics) and type (*i.e.,* syntax) of each component to each other.

The semantics of a service component are defined by its intended use, the component's physical conditions (*e.g.,* deployment environment), the external information gathered by sensors, or possible interactions. In our routing example from Figure 1, the FORECAST component can be defined with the "routing" semantics as the service is intended to be used for this purpose, while the BIKEAVAILABILITY component can be defined with the "transport" semantics, as the sensors it uses are associated to a transport mode (*e.g.,* physical bikes).

Service components initially exchange their semantic information conveying what kind of application domain they can be used for (*e.g.,* routing, environmental, video). Semantics for a service component follow an ontology definition, such that semantics for related application domains are associated in an ontology graph. The advantage of using such semantics definition, is that correspondence between service components is not required to be exact (*i.e.,* the components may not be part of the same application domain). More general or more specific semantics could be used for correspondence in case there is no exact match [23]. For example, the "transport" semantics can be used as a more general application domain to the "bus" semantics. Ontology semantics definition is normally used in centralized settings, where a global ontology structure can be maintained and accessed by multiple components. Nevertheless, this is not the case in

pervasive environments. Rather than using one big semantics ontology definition, service components in Mordor keep their own slice of the ontology [5], keeping track of only those semantics relevant to the component. Service components semantics can evolve by learning new interactions with other service components (*cf.,* Section 2.3), for example, by taking into account user-driven actions to compose services.

After service components have matched, such components proceed to exchange their syntactic information, to check if interaction is indeed possible. In order to have a successful interaction and collaboration between service components, the functionality of one of the services should extend, or be used by the other. That is, the outputs of one component can be used as the inputs of the other one. In Mordor, we follow the service composition requirements normally used in SOC [8]. Given two service components $C_1$ and $C_2$ with corresponding inputs and outputs $In_1$, $In_2$ and $Out_1$, $Out_2$, we say that, for example, $C_1$ is usable by $C_2$ if $Out_1 \subseteq In_2$. That is, the outputs of the first service component satisfy the inputs of the second one.

In order for service components to interact it is not sufficient to match their semantic and syntactic interfaces. In addition, the QoS requirements of the first component ($C_1$) must be satisfied by the component using it ($C_2$). Service components' QoS interface is as flexible as possible, and any kind of attribute can be defined for a given service component.

In Mordor, the QoS requirements policy is open for service component developers to define. Such policy should express how many and up to what extent QoS requirements should be satisfied. For example, in the routing application, we could define `timeToResolve` to be the most relevant QoS attribute, requiring service components to respond within a given threshold. If the components do not satisfy this policy, then they do not match.

If both the semantic and syntactic interfaces of two components match, they are deemed to interact. The resulting service of the compositions is then pushed to users, so that it can be called an executed. If, on the contrary, the components do not match, then there is no interaction between the components, and no service emerges to be called by users.

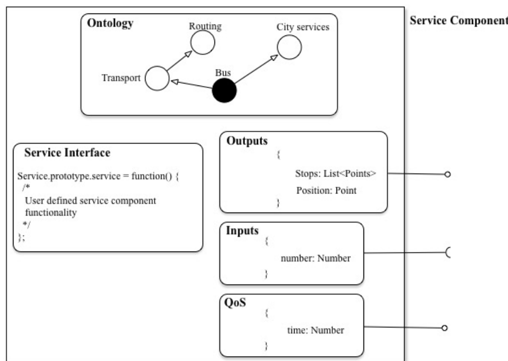



**Figure 3:** Service component, interaction perspective

Figure 3 shows the main elements in a service component. The generated ontology slice, and the definition of the inputs, outputs, and QoS elements, used to broadcast and discover components in the environment. The service interface, as we explain next, presents the actual service functionality.

### 2.2 Fine-Grained Service Definition

The service emergence vision follows the trend of microservice architectures [35], which foster the flexibility of service-oriented architectures by combining multiple small services. Such flexibility is appropriate for IoT and CPS environments, where software services are associated with specialized and small devices (*e.g.,* sensors). We envision Mordor to offer equally specialized and small services. Along side services' semantic and syntactic interface, developers provide the main functionality of a service component. This functionality is exactly the behavior other service components will use when composing full-fledged services. In the routing example, the definition of the ROUTE service component is as in Snippet 1, where `transportMode` corresponds to a service component providing information about a particular transport mode (*e.g.,* bike, bus), and `origin` and `destination` are input parameters of the component.

```
Route = Service({
  var schedule= transportMode.getSchedule
      ();
  var route= _.filter(schedule, function(
      node) {
    return node >= origin && node <=
        destination;
  });
  return route;
});
```

**Snippet 1:** Definition of the route service component

The importance of having a fine-grained definition of service components, is that failing or disappearing components can be replaced at run time (*cf.,* Section 2.4), improving provided services availability [8, 14, 16].

### 2.3 Emerging Service Composition

The previous sections discussed the definition and interaction of service components. This section turns the discussion into the emergence of full-fledged services from such components. In order to compose services, each service component keeps track of all other components it can interact with. Mordor achieves this using a model-free learning technique, specifically Q-learning.

Q-learning [34] is a reinforcement learning algorithm used to optimize decision-making processes in environments where there may not be complete knowledge about the system states and transitions beforehand. This can be applied to dynamic environments where the complete information about the environment is usually unknown. Given a system, Q-learning records (*i.e.,* learns) the actions taken to reach

one system state from another. In addition, each action taken by the system is associated with a reward function, in Equation (1), calculating the long term cumulative reward from a particular state $s$, where $r_{t+1}$ is the reward after $t$ transitions have executed. $\gamma$ defines the discount factor providing a weight of the decisions for each action over time.

$$r(s_t) = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{1}$$

As time progress, the system specializes its actions more and more, so that with every new change of state, the system chooses the optimal action with respect to the long-term pay-off. Optimal actions are chosen based on states' q-value. These values are associated to each state and represent the overall pay-off of choosing the action to get to that state. q-values are updated with every action, using the reward function.

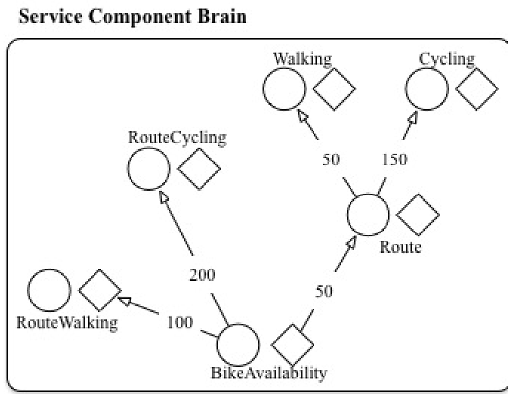



**Figure 4:** Service component, learning perspective

In order to learn interaction between service components we modified the Q-learning process. Whenever two service components are deemed to match, say $C_1$ and $C_2$, the component extending the behavior of the other component (*i.e.,* the component such that $Out_1 \subseteq In_2$) becomes the service *leader* and keeps track of the interaction with the other component. In our case, $C_1$ is deemed the leader, so it learns that $C_2$ is one of its reachable states (states are represented as circles in Figure 4) and the actions to reach such state (actions are represented as rhombi in Figure 4). Each learned state has a reward associated to it. Every time a service is used, the q-value of the state associated with such service increases using the reward function, increasing the likelihood of using that service in the future.

Figure 4 depicts the "brain" of service components in Mordor —that is, the entity in charge of learning which *states* (*i.e.,* service components) and *actions* (*i.e.,* service combinators) a given component can interact with. In the routing example, the BikeAvailability component can interact with a ROUTE component, which in turn can interact with the Cycling and Walking components. Therefore, the components learned by BikeAvailability correspond to the Route service component, and the services emerging form this component (*i.e.,* the compositions with the services it interacts, namely, RouteCycling and RouteWalking).

Once interaction between service components is learned, brand new (*i.e.,* emergent) services are pushed to users. Such services, however, need to provide information about their functionality and specification (*i.e.,* name, service specification, and description). This responsibility rests on service leaders, as the starting point of a service execution. Service leaders can generate a semantically meaningful service interface that is pushed to users. The alternative chosen in Mordor to generate the full service semantics interface reuses the service components' semantics ontology. For example, for the BikeAvailability component, a cycling route emergent service could be pushed to users as a "Cycling transport routing" since the components of this service are respectively defined "transport mode" (for the BikeAvailability component), "routing" (for the Routing component), and "bike" (for the Cycling component). Such a name could be given to the service by, for example, concatenating each of their semantics. The definition of meaningful semantics for emergent services relates to semantic web definitions [19], and is part of our avenues of future work (Section 5.2).

### 2.4 Service Execution

Once service components are composed to generate new full-fledged software services, the latter can be pushed to the user for their subsequent execution. Users get access to emergent services via the Mordor application, a service discovery application receiving all available services that emerge from the environment. Figure 5 shows a mockup interface of the Mordor application. In this figure, emergent services are displayed using the concatenation of its components' names, and are grouped by application domain according to the semantics extracted from all its components.

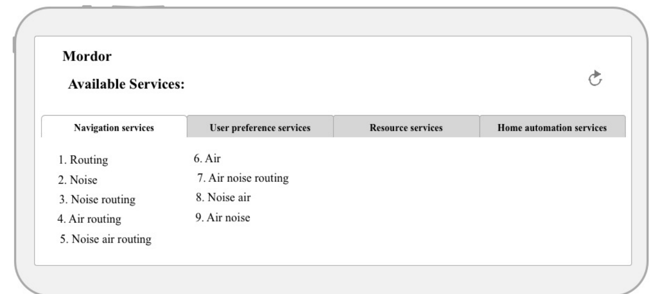



**Figure 5:** Mordor emergent services mockup interface

Selecting one of the services in the Mordor application trickles down the execution of each of the components functionality. In order to manage possible errors during service execution due to service components disappearing from the environment or failing, the concrete components to execute are not defined. Before a particular service component ex-

ecutes, its availability is verified. In case the component is no longer available, the execution can adapt to replace this component with a component providing similar functionality.

We manage service adaptation by means of adapting the problematic service component. In existing approaches, services are adapted by changing missing or failing components for new components providing an equivalent functionality [8, 16]. We take advantage of the fine-grained definition of service components' functionality to drive their adaptation by means of a programatic adaptation technique, as proposed by Context-oriented Programming (COP) [5, 10]. The main idea behind COP is to modify the behavior of an application according to the situations of its surrounding execution environment (*e.g.,* available service components). Using this technique, it is possible to adapt the behavior provided by a single service component with that provided by other component.

This type of behavioral adaptation at the component level can also be used to refine the type of services that emerge for a given user, customizing the behavior of services according to users preferences (*cf.,* Section 5.4).

## 3. Emerging Services in Mordor

This section uses our prototype Mordor implementation to define service components in an environment and observe service emerge. We define two proof-of-concept service scenarios with two purposes. First, we demonstrate the feasibility of service emergence through the simulation of two small example applications. Second, we illustrate the usability and appropriateness of Mordor for the definition of service-oriented systems using a routing scenario employing real city services and service components.

The first scenario uses two applications, a color palette offering *color* combination services. This application offers users a palette to chose a given color, from those available in the environment. Each service component represents a color, defined using either the RGB or CMYK color system. Colors belonging to the same color system can be combined (*i.e.,* interact), offering a color palette services consisting of all possible combinations of colors available in the environment. Selecting one such service executes the composed service behavior —that is, the composed color is used in the palette. The second application defines a discoverer application offering *poker-hand* services. In this application each card represents a fine-grained service component. Offered services to users are constrained to sets of five components, making the different poker hands to be played (*e.g.,* two of a kind, straight, or royal flush). Selecting one of the available services, plays the given hand one card at a time.

The second scenario consists of a routing application in a city. The application is composed by a routing service present in users' mobile device, and different service components deployed around the city offering route enhancements according to the surrounding environment. This scenario is used to show the definition of service components (*i.e.,* their semantic and syntactic interfaces), as well as to show how multiple heterogeneous service components interact without developers' involvement.

### 3.1 Current State of Affairs

Mordor is currently implemented in ECMAScript. In this section we describe the state of affairs of the Mordor implementation with respect to the full model described for our vision in Section 2.

Currently, Mordor offers developers an API to instantiate service components, as well as a textual version of the Mordor application to discover services (Figure 5). Service components are broadcasted into a network using the mDNS protocol, where all service components defined in Mordor can be discovered. Components self-organize taking into account those service components belonging to the same application domain —that is, service components are discovered according to their semantic interface (*i.e.,* ontology slice), and exchange only their syntactic interface (inputs, outputs, and QoS). Service components are deemed to match based on a user-defined syntactic matching policy (*i.e.,* a `conditions` function) defining additional conditions for service matching.

Service components only learn about other components they can interact with, and keep track of them as new states in their Q-learning brain. While service components are learned as they appear, their q-values do not yet condition the behavior of the system. All components have the same reward function. Each component keeps track of all reachable services from it by means of a `serviceCombinator` function, which determines the concrete interaction between them — that is, how the outputs of a service component are used by the inputs of another one. Finally, to push emergent services, we use the plain-text name of the combination of all service components in them.

### 3.2 Service Emergence Feasibility

To investigate the feasibility of software service emergence, we use the color palette and the poker-hand application scenarios. For each of the applications we created a simulation as follows:

1. Service components are created and deployed in the environment on regular $5s$ time intervals.
2. Service components interact with each other —that is, they broadcast their interface and interact with other (unknown) service components every $1s$.
3. The Mordor application is refreshed to display pushed services on fixed time intervals.

For each of the simulations we measured the number of services discovered by Mordor with respect to the number of service components deployed in the environment. This measurement was taken in regular time-steps (*i.e.,* as per the Mordor display rate).

### 3.2.1 Color Palette

In the color palette application, we generate two simulations shown in Figure 6. The first simulation consists of 10 RGB colors (with randomly generated values). In the color palette application, the number of colors that can emerge varies with respect to the initial values used for each color and the following restrictions: (1) colors are only applied once to a given combination, and (2) the maximum saturation value for a basic color is 255 (*e.g.,* green = {`R: 0, G: 255, B: 0`}. With this restrictions in mind, the number of services that can emerge in an environment in which $n$ ($n \geq 2$) service components are deployed, is in the range $[n+1, 2^n+1]$. The lower bound corresponds to an environment in which any pair-wise composition of colors reaches black {`R:255, G:255, B:255`}, and the upper bound to whenever all possible combinations of all service components are possible. Figure 6a, shows the growth (in logarithmic scale) in the number of services as the number of service components increases taken in 2*s* intervals. The bottom gray line shows the estimate number of services to discover —that is, one per each service component (the identity function). The top blue line shows the number of services discovered in the simulation. Note that, for this particular simulation, around 5 service components the amount of services that emerge is double, by the time 10 service components are deployed, the services that emerge has increased to over 4 times the number of deployed service components.

The second simulation deploys 50 color service components with a random distribution for the amount of colors in the RGB and CMKY color systems. Figure 6b shows the number of emergent services, as the number of service components in the environment increases, in logarithmic scale. As before, the bottom gray line shows the identity in function of the number of deployed service components, and the top blue line shows the values obtained in the simulation. Note that rate of growth of discovered services is more erratic than in the previous case, with periods of rapid growth and periods of more steady growth. This might be due to the types of services deployed in the environment. The periods of rapid growth may see a larger number of colors using one particular system. Given that these are able to interact, the number of emerging services increases. As the distribution between color systems evens up, the growth rate decreases, as one new color can only interact with a portion of the deployed service components.

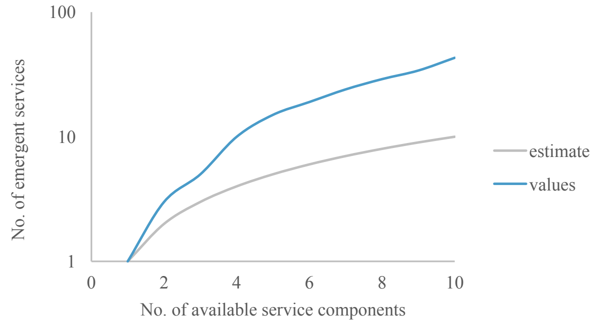
From both graphs in Figure 6 is possible to see that the number of services available has a higher growth rate than the linear growth of the number of service components deployed in the environment. This is because new services do emerge as deployed components start interacting. The deployment of a new service component in the environment will prompt its interaction with all other services deployed in the system, generating about $n+1$ services. In the worst case scenario, where no new interactions occur, the growth is linear. If there are interactions, the growth will be faster. The simulation results shown here, are not absolute for the applications used, but rather present the trend of growth for emergent services. As a matter of fact, simulations may variate with respect to the particular services deployed for that simulation

Note that the exact number of services that are generated depends on the distribution of service components having the same semantic and syntactic interface in the environment. In the color palette example, there are two types of services, those representing RGB colors and those representing CMKY colors (syntactically, each component represents a different color). In this case, all service components presenting the same semantic interface (*e.g.,* color system) will interact. Mordor does not discriminate between services providing the same semantic interface, interaction takes place between all of them. However, if two services provide the same semantic and syntactic interface (*i.e.,* are functionally the same service), the first service with which an interaction is stablished is learned, while the second service will be used without any learning process taking place. Equivalent service components can be used to replace failing components for their equivalent.

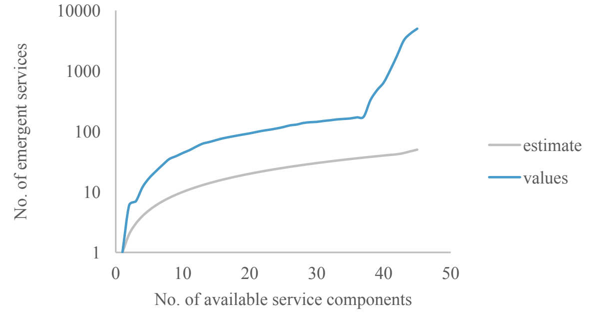
### 3.2.2 Poker-Hand

In the poker-hand application we generate a simulation in which the 52 cards of a playing deck are deployed in the environment, deploying a card every 5*s*. In this example, services discovered in Mordor consist of sets of 5 cards. As in the previous application, to investigate emergence we measure the ratio between available services (*i.e.,* sets of five cards) and the deployed service components (*i.e.,* individual cards). Analytically, if there are $n$ service components there will be $\binom{n}{5}$ services. Figure 7 shows the effective numbers of emergent services in relation to the available service components taken every 5*s*. As it is possible to see from the graph, the simulation values (in the blue top line) surpass the expected values (in the bottom gray line). This is because in our poker composition playable hands [10♡][J♡][Q♡][K♡][A♡] and [J♡][K♡][Q♡][10♡][A♡] are considered different, as the order in which they are composed may influence the outcome of the service. Therefore, the upper bound for service combinations is ${}_nP_5$. The poker application reaches the upper bound as we do not check for semantic matches between service components.

Figure 7 shows the results gathered from the simulation. As it is possible to see, the amount of services pushed in the simulation is slightly lower than the upper bound. This is because processes deploying service components and the one measuring components' interactions are asynchronous and processing all interactions for a given number of components lags behind the the time to measure pushed services. Furthermore, not all components can be successfully deployed in the experiment. This is because after 17 components the system stack fills as there are over 730000 services created, and no more data can be collected. This example exhibits the complexity of managing emerging services as multiple interacting service components appear in the environment.




**(a)** Single color system emergence




**(b)** Multiple color systems emergence

**Figure 6:** Color palette emergent services to service components simulation

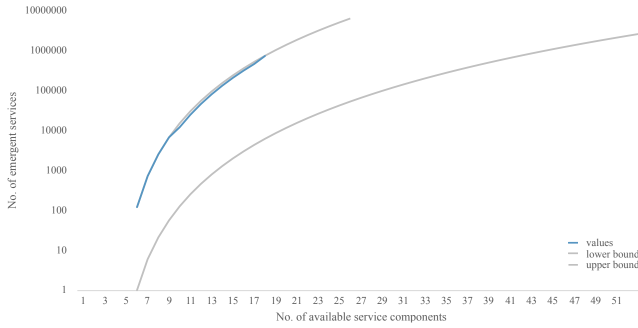



**Figure 7:** Poker-hand emergent services to service components simulation

### 3.3 Service Component Development

The push model envisioned for service emergence aims to reduce the complexity of developing software services (*i.e.,* the definition of service discovery, composition, interaction, and execution) in pervasive environments. Using Mordor, application developers are to focus on the main functional logic of service components, while the other aspects of service definition should be inferred autonomously from the surrounding execution environment (*e.g.,* available services in the network, or previously learned interactions with other components).

Now that we have shown that services effectively emerge from the environment, we present a real-world case study, illustrating the definition and interaction of service components. For this purpose, we use a routing CPS application as case study. Now a days, cities are equipped with a diversity of IoT devices providing information about the city, such as pollution (*e.g.,* noise, air), ambient conditions (*e.g.,* temperature, humidity), or mobility (*e.g.,* public transport schedules, traffic density, available transport modes) to mention a few. Based on such information routes can be enhanced to take users around the city. Unlike existing routing applications fully providing enhanced routes to users [21, 24], based on information gathered by sensors around the city. Our case study differs from these applications by providing the enhanced services autonomously as more IoT devices and service components are deployed in the environment, rather than requiring developers to continuously integrate such devices and components.

Routing users through a city should offer a routing service, for example, accessible from mobile devices. This service, defined as a Mordor service component, provides a route, given its origin and destination.

Mordor service components are created as stand-alone instances of service entities, describing their semantics, and functionality. A `routing` service component (Snippet 2) should be aware of the city, represented as a graph, where each edge corresponds to a street. The component is defined with its inputs, outputs, QoS attributes, and functionality. The `routing` component receives as inputs, the `origin` and `destination` of the route (given by users), and `weightingData`. The `weightingData` is used by the graph algorithm, and corresponds to the weight given to every edge according to the purpose of the route. For example, the shortest route would weight edges according to street lengths, while the quietest route, would weight them by the noise levels perceived in the street. The component's outputs correspond to the path to follow by users. Finally, in our application we rank weighting data according to its reliability, giving a priority to the data to use. The more reliable data is, the highest its priority (1 being the highest priority posible). Our case study will consider highly reliable data sources, having priority 1 or 2, as expressed in the `conditions` function, Lines 5-8 in Snippet 2. Note that Snippet 2 is only showing the basic service definition: its main functionality, the A* graph traversal algorithm, providing a path between two nodes (Lines 9-15), and the graph.[1]

The `routing` component offers the possibility to combine other services through its `weightingData`. Components' compositions are managed in the `combinator` function expressing

[1] the graph and auxiliary algorithms are omitted for conciseness. These are defined in the component's private interface (*i.e.,* methods not shared with other components)

how the composition takes place. For example, the `routing` component composes with components providing weighting data, *i.e.,* IoT devices. As shown in the `combinator` function (Lines 16–26), this is done by associating the new weight with each edge, so that edges have tuples of weights. Edges are ordered lexicographically on each dimension of the tuple.

```
1  var routing = new Service({
2   inputs: [origin, destination,
       weightingData],
3   outputs: path,
4   qos: { "priority": function(p)
5                          {return p < 3;}
                             },
6   conditions: function(interface, state)
       {
7    return this.qos.priority(state.
        priority);
8   },
9   serviceInterface: {
10   service: function() {
11     //build the path using the A*
         algorithm
12     //Edges in the city graph are
         weighted with the weightingData
13     return astar.runAlgorithm();
14   }
15  },
16  combinator: function(serviceStates) {
17   _.each(serviceStates, function(state)
        {
18    _.each(state.data, function(edge, w)
         {
19      _.map(this.graph.edges, function(e
          ) {
20        if(e.id == edge.id)
21          e.weight.push(w);
22        });
23      });
24    }, this);
25   return this.graph;
26  }
27 });
```

**Snippet 2:** Routing service component definition

Suppose now, that while the `routing` service is in use, the city deploys sensors measuring different environmental conditions such as `transportModes`, `traffic`, `noise`, or `airPollution`, exporting gathered weighting data as service components. The service components providing weighting data based on the city's environment, such as `noise`, or `traffic`, are defined similarly. Snippet 3 shows the definition for the `noise` service component. The functionality of this component is to output all information collected in the city, by giving each street its weight. In the case two weighting services compose, they are combined similarly than in Snippet 2 —that is, creating weighting data tuples for every edge.

```
1  var noise = new Service({
2   inputs: [],
3   outputs: [[e1, nw1], [e2, nw2], ...],
4   qos: { "priority": 1 },
5   serviceInterface: {
6    service: function() {
7      _.each(this.graph.edges, function(e)
           {
8       client.get('fwk',function(err,reply
           ){
9         this.outputs.push[e, reply];
10      });
11     });
12   }
13  },
14  combinator: function(serviceStates) {
15   //As the combinator in Snippet 2
16  }
17 });
```

**Snippet 3:** Noise service component definition

Service components are accesible only in the vicinity of their respective devices. For the purpose of this example, assume that we have two environments, one with only the `noise` component and one with both the `noise` and `airPollution` components. As users enter the first environment, the `routing` and `noise` components interact, pushing all services in the environment (*i.e.,* `routing`, `noise`, and `noiseRouting`). Similarly, in the second environment all service combinations are pushed (those shown in Figure 5). Note that service compositions as `noiseAir` and `airNoise`, displayed last in Figure 5, are not particularly useful services. These services emerge as their components are defined with empty inputs, so other components answering to the same semantics can be combined with them too. Executing these services will give the user a list of streets with two associated weights.

This case study illustrates how real world applications are defined in Mordor by means of the independent definition of service components, demonstrating that complex services do emerge from the environment.

Note that, service components are defined without any prior knowledge about the environment, or services previously defined. All components can be defined and deployed by different developers at different times. Normally, each ordering to execute services components would require a different composition strategy. In Mordor, regardless of the deployment order, all possible combinations of services emerge autonomously.

Beyond the illustrative emergent services in the case study, new services can appear as more basic city services (as the ones shown in the case study) are deployed. One may think of fitness routing services, which combine the different transport modes in the city and its infrastructure, `bikeRacks` or `joggingLanes`. Deploying services providing the weighting information about these modes, and the location and availabil-

ity of the infrastructure, the JOGGINGROUTE, BIKEROUTE, or even DUATHLON (combining jogging and biking) services emerge. Each of these services would require a completely different definition (by interested parties) without Mordor.

As more and more of these service components are deployed in, and removed from the environment, new and interesting services emerge, increasing their availability fluctuation. Yet, their definition does not increase in complexity. Developers are still defining fine-grained components. Nonetheless, not all services that emerge will be useful to users. Over time, such services are relegated to the end of the pushed services list, giving priority to more frequently used (*i.e.,* relevant) services (based on the q-value of each service), reducing too the complexity of choosing a given service.

### 3.4 Extension to Heterogeneous Services

The case studies presented in this section provide initial results supporting our vision for software services emergence. As demonstrated with our simulation, it is possible to observe new services emerge from the interaction of previously unknown service components deployed in a pervasive environment. However, a note of warning must be raised about the obtained results. While it is possible to observe services emerging in the simulations, the results are not yet generalizable to any two types of services interacting in the environment. All service components used in the examples belong to the same application domain (*i.e.,* colors, poker playing cards, or routing), and so, interactions between service components are restricted and well defined by their `combinator` function.

In order to generalize the results obtained in this section to a set of heterogeneous service component types, to manage interactions between components in multiple application domains we must extend Mordor so that: (1) service components can exchange messages to promote interaction, even if their semantic interfaces belong to different application domains, (2) the semantics of a service should be elevated from its components, (3) service components learn with which other components they can interact, and (4) service components learn which actions must be taken to interact with such components (*i.e.,* the combinator function used).

Service components belonging to different application domains may still reach coherent and complete services when combined. To allow such compositions, it should be possible for components to "probe" each other's services, and observe if sound functionality can be reached. In conjunction with this, the system should be able to generate what the service semantics are, so that the service is meaningful for users. For example, in the routing case, a TEXTTOSPEECH component could be combined with a ROUTE component in order to generate an *audible route* stream.

There may be multiple ways to successfully combine two service components, for example in two different application domains. In addition to learning what these actions may be, the learning component in Mordor should also estimate which of the combinators is best suited for the application domain at hand. These and other challenges to be tackled in order to realize the full vision of Mordor are discussed in Section 5.

## 4. State-of-the-Art

To the best of out knowledge, there are currently no similar approaches proposing the emergence of software services by means of their autonomous interaction in a push-service model. However, as previously mentioned, our proposal draws inspiration from existing research areas. Here, we discuss the similarities, shortcomings, and advantages of our approach in the perspective of the following four areas.

***Decentralized Service Composition*** In SOC development, services and their availability are assumed to be globally known [31]. However, this is no longer the case in pervasive environments, where services can be deployed into, or removed from the environment unannounced, due to the mobility of their containing devices. In response to this, Chen et al. [8] propose a new service composition model, GoCoMo. This model is the closest to our proposal from existing approaches. GoCoMo addresses the problem of service components availability by using a decentralized composition model in which unavailable service components can be replaced by other equivalent (set of) components through dynamic adaptation. Service components appropriate to satisfy a given service request, and to replace failing components, are selected from all available components in the environment using a goal-driven algorithm. There is no defined structure for service components in GoCoMo; rather, components self-organize (*i.e.,* define their interactions) upon a service request sent by users, as the initiator of the execution process. Such a request can come from a pre-defined service of the application in use, or as a query expressing the goal to be fulfilled. Regardless of the method used, at the moment of a service request, it is unknown if the requested service can be executed due to components' unavailability. Therefore, services can still be requested by users and be unavailable in certain environments. This fact constitutes the main novelty and difference of Mordor over GoCoMo. Using Mordor, by definition, all services available to users (*i.e.,* can be requested) are executable, as these are exactly the services pushed to users from the environment. Additionally, while both approaches enable service components to self-organize, in Mordor such organization is independent from service requests. That is, service components interactions are determined at components' connection/disconnection time (*i.e.,* discovery), and remain untouched so long service components availability does not change.

The difference between the two approaches may impact the execution of services in pervasive environments in two ways. First, the success ratio of a service execution should improve using Mordor, due to the assurance that all components of pushed services are available in the environment. The way in which service components self-organize will have an impact on the delay-time of answering for a service request

depending on the volatility of the environment and the stress on services' use. In an environment with low volatility of service components in which (the same) services are called often, GoCoMo services may take longer to execute than in Mordor. In highly-volatile environments where services are not called as often, Mordor services may take longer to execute. A full evaluation of the tradeoff between the two approaches is left for future work.

***(Model-free) Learning*** Bringing learning techniques into SOC is not an idea exclusive to Mordor. It is important to note, however, that in order to allow service emergence, it is necessary to use a model-free learning algorithm. Having a model would require previous knowledge about the system, similar as when using an abstract workflow specification to define a service. Notwithstanding, learning has been used extensively in SOC to optimize service composition with respect to certain QoS requirements [20, 33]. Learning has also been used for service adaptation using dynamic composition planning [37].

Composing software services from multiple available providers is a difficult task because not all services may provide the same execution properties —that is, satisfy the same QoS requirements. As a result, services may be composed in sub-optimal fashions. Optimizing service composition with respect to QoS attributes can be improved by means of reinforcement learning. Here, the system learns which service component combinations yield a better system state [33], even when multiple QoS attributes are evaluated simultaneously [20]. Service composition learning in Mordor draws inspiration from these approaches in that we also learn which service components can be composed with one another. However, unlike existing approaches, in Mordor, the states of the learning engine are generated at run-time.

Service adaptation through learning uses a central composition engine, where a composition plan is made according to the rewards obtained from previous compositions. In case the execution fails, a new plan is drawn using different service components [37]. This approach is somewhat similar to ours in that service components analyze possible compositions with every new component that is deployed in the environment.

***(Context-aware) Mashups*** Mashups are introduced to integrate data from multiple sources and provide new services from the combination of such data. Mashups proved useful as visualization services, by combining geo-location with service data. For example, a mashup can be used to display all social media posts in a map, during a particular period of time. Furthermore, mashups can use sensor data to specialize provided services to the current situation in the surrounding environment [4]. Mashup definition is sequential, following a defined structure in which data can be combined. Such sequential definition makes the applicability of mashups to pervasive environments complex. This is due to the uncertainty about resources availability associated with these environments. Dynamic mashup solutions are explored to tackle the aforementioned problems. Dynamic mashup definitions [32], introduce service types as a means to group services providing similar functionality and abstract the interaction with concrete mashup services. This model also permits the dynamic selection of mashups according to the surrounding environment.

Mashup development is similar to SOC in that it intends to compose services from already existing ones. Mashups run into many of the same problems regular SOC approaches face, and so are unfit for our purpose. However, the idea of service types introduced in mashups could be used as inspiration to extend the ontology classification of service components as a means to group them.

***Symbiotic Interfaces*** Symbiotic Interfaces are introduced in the context of human-machine interaction as a means to enable ad hoc collaboration between users. This technology is used, in particular, in the presence of wearables, where the devices can both take advantage of the situation to improve their functionality, and use the system to help human behavior [2, 29]. While introduced in a different setting, symbolic interfaces share some commonalities with our approach, as they enable the possibility to learn information about the executing system in order to predict future behavior.

## 5. Challenges

In Section 3.4, we already discussed some of the extensions required in Mordor to enable the emergence of software services in environments with heterogeneous service components. In this section, we further discuss some of the current and future challenges faced to completely fulfill software service emergence. This section also serves as a roadmap for the future research avenues to explore in, and spawn from our vision.

### 5.1 Self-Organization

One of the main challenges in service emergence, arising as a consequence of the interaction between multiple service components, is the definition of services as semantically meaningful objects understood by users. Such semantic definition is required to convey the intention of all components making up the service. The research question to explore here is how to give meaning to an aggregated object. Our current exploration combining the semantic ontology associated with each of the used components could be extended by exploring emergent semantic techniques in the data-base community [11], the use of markup techniques in the semantic web [19], or even the creation of meaning in linguistics [9, 22].

### 5.2 Learning Interactions

In the learning area, we could also consider using a dynamic reward function for the learning phase of service components' interactions. As mentioned in Section 2.3, every service composition has an associated reward for a particular service

component. The more a service component is used, the more likely this service is to be used in the future. However, given the environment's dynamicity, the conditions in which services execute may change, requiring to modify the reward function involving that service. For example, because a component's QoS decreases, or one component is no longer available for the execution of a service. To deal with these situations, Mordor's reward function should proportionally track QoS attributes, and each time they change adjust the currently learned services, so that the likeliness of using services with deteriorating QoS values, until the service is fully un-learned.

### 5.3 Communication

IoT and CPS environments are composed of tens and hundreds of devices. If a service component is defined for each of these devices, the messages exchanged to organize them may quickly overflow the network. Further research in Mordor is required to optimize network usage. Rather than allowing all service components in the environment to interact with every other component, we could apply message passing algorithms used in distributed and peer-to-peer networks. Among the possibilities to explore we include the use of tuple spaces [7], gossip algorithms [28], and ultrapeer communication [30].

### 5.4 Dealing with Choice

Emergence of services from available service components can potentially result in an excess of services that can be pushed to users. This is specially the case in IoT and CPS environments where the number of devices potentially defining fine-grained service components is ever at a rise. As described by *the paradox of choice* [26], offering too many services to users might actually prevent users from using any service, in particular because users are not previously aware of the services that might be available.

To attain this concern, in Mordor we already envision emergent services to be organized according to their semantic application domain, as shown in Figure 5. Additional refining of available services could be done by means of filtering results. More over, we can use the q-value of services to order them according to their relevance (*i.e.,* services used more are shown first as they are "preferred"). Nonetheless, we foresee these measures not to be enough. Two research areas can be explored to ease this problem.

First, it is clear that not all services are of interest to all users. For example, in the poker-hand application, users may only want to see winning hands —that is, hands containing three of a kind, or higher. Behavioral adaptations have been used in the past for the specialization and customization of application behavior with respect to users' preferences. Extending the service component adaptation technique introduced in Section 2.4, it is possible to customize the kind of services that emerge for a particular user. For example, sight impaired users may want to filter emergent services exclusively to those services whose outputs can be transformed into an audio signal.

A second possibility is to refine the services gathered in Mordor according to a semantic ontology, similar to the way semantic web refinements work. The idea behind this approach, similar to the customization of observed services, is to refine the kind of services that are pushed to users. For example, a decision tree could be built based on user-defined ontologies, specifying what kind of services are of interest to users [15].

### 5.5 Reliability and Robustness

Coupled to the problem of dealing with choice presented in Section 5.4, come the challenges of the system's reliability and robustness.

The type and amount of services pushed to users solely depends on the environment of use. Users may find discomfort in an application, using Mordor, offering a variable set of services, as services previously used may not be available in new environments. User acceptance to the new service provision approach proposed with Mordor is key for its realization. In future work, several user studies must be conducted to evaluate the acceptance of users when faced with such a system. Such a user study would consist of a user group using a fixed restricted set of services in different environments, some of which are not able to satisfy all services' requests. A second user group will use Mordor in the same environments as the previous group, using the services that emerge from each environment. Then, the reactions of both users groups can be compared with respect to service availability, frustration of using the system, and conformance to the set of services used.

Additionally, emergent services may present side-effects or behavior that does not map to users' initial expectations. The fact that a set of service components can interact and be combined into a service does not guarantee that this service would be useful, or be accepted by users. This problem can be addressed from two perspectives. Semantically, the emergent service could express the effects of executing a service [3]. This way users would be better informed about a the services they execute. In the poker example, this is done by presenting the service as the combination of all five cards, rather than just the name of the play *e.g.,* "three of a kind." A second alternative, would be to use a user defined functional verification process during service components' composition, such that the effects of a service can be verified upfront its execution. This alternative, for example, could be specified not to consider service combination of triplets of cards where the remaining two cards have a denomination below seven. The drawback of this alternative, is that users are being burden again with the definition of verifications for multiple services they may not know during development. Both of these alternatives, would need to be explored in the future.

## 6. Conclusion

This paper proposes a new vision for Service-oriented Computing (SOC) applicable to pervasive environments as realized by the IoT and CPS. In our vision we reverse the control-flow of service composition and execution by enabling the system to dictate what services can be executed in a given environment. Software services emerge from the interactions of service components deployed in the environment, and are subsequently pushed to the user. The advantage of using such SOC model is that service developers are not required to have any previous knowledge about their deployment environment, or other services deployed in it. Rather the development focus can be on components' logic. Keeping components' functionality as fine-grained as possible, the flexibility and availability of services increases, being able to compose a full-fledged service in various ways (*i.e.,* (re-)using different service component providers).

We have implemented a first prototype framework for software services' emergence, dubbed Mordor. Mordor currently contains the basic functionality to define service components that can learn and interact with other components. In order to validate the feasibility of our vision, we developed three applications using Mordor. These applications define multiple services components, confirming that new composed services emerge, and that real-life complex applications can be build with Mordor. Furthermore, we discussed avenues for future work required to fulfill the overall vision of emergent software services.


## Acknowledgments

We thank the anonymous reviewers for their comments on earlier versions of this paper. Special thanks go to Ivana Dusparic for discussing the ideas, and reviewing early drafts of this paper.


## References


[1] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A service creation environment based on end to end composition of web services. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 128–137, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9.

[2] D. Ashbrook and T. E. Starner. Enabling ad-hoc collaboration through schedule learning and prediction. In *Proceedings of the CHI Workshop on Mobile Ad-Hoc Collaboration*. ACM, 2002.

[3] N. Bencomo, K. Welsh, P. Sawyer, and J. Whittle. Self-explanation in adaptive systems. In *17th International Conference on Engineering of Complex Computer Systems*, ICECCS'12, pages 157–166, July 2012.

[4] A. Brodt, D. Nicklas, S. Sathish, and B. Mitschang. Context-aware mashups for mobile devices. In *Proceedings of the 9th international conference on Web Information Systems Engineering*, WISE '08, pages 280–291, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85480-7. doi: 10.1007/978-3-540-85481-4_22.

[5] N. Cardozo and S. Clarke. Context slices: Lightweight discovery of behavioral adaptations. In *Proceedings of the Context-Oriented Programming Workshop*, COP'15, pages 2:1–2:6. ACM, July 2015. doi: http://dx.doi.org/10.1145/2786545.2786554.

[6] N. Cardozo, K. Mens, S. González, P.-Y. Orban, and W. De Meuter. Features on demand. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems*, number 18 in VaMoS'14, pages 18:1–18:8. ACM, January 2014. ISBN 978-1-4503-2556-1/14/01.

[7] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman. Message passing interfaces: The linda alternative to message-passing systems. *Parallel Computing*, 20(4):633 – 655, 1994.

[8] N. Chen, N. Cardozo, and S. Clarke. Self-organizing goal-driven services in mobile pervasive computing. *Transactions on Services Computing*, page to appear, 2016.

[9] P. Cobley, editor. *The Routledge Companion to Semiotics and Linguistics*. Routledge, 2005. ISBN 0–415–24313–0.

[10] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium*, pages 1–10, Oct. 2005.

[11] P. Cudré-Mauroux, K. Aberer, A. I. Abdelmoty, T. Catarci, E. Damiani, A. Illaramendi, M. Jarrar, R. Meersman, E. J. Neuhold, C. Parent, K.-U. Sattler, M. Scannapieco, S. Spaccapietra, P. Spyns, and G. D. Tré. Viewpoints on emergent semantics. *Journal on Data Semantics*, 5:1–27, 2006.

[12] A. Furno and E. Zimeo. Efficient cooperative discovery of service compositions in unstructured p2p networks. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP'13, pages 58–67, Feb 2013.

[13] M. Gharzouli and M. Boufaida. Pm4sws: A p2p model for semantic web services discovery and composition. *Journal of Advances in Information Technology*, 2(1):15–26, February 2011.

[14] C. Groba and S. Clarke. Opportunistic service composition in dynamic ad hoc environments. *IEEE Transactions on Services Computing*, 7(4):642–653, 2014.

[15] D. Jeon and W. Kim. Concept learning algorithm for semantic web based on the automatically searched refinement condition. In W. Kim, Y. Ding, and H.-G. Kim, editors, *Third Joint International Conference on Semantic Technology*, JIST'13, pages 414–428, Cham, November 2014. Springer International Publishing. ISBN 978-3-319-06826-8.

[16] S. Jiang, Y. Xue, and D. C. Schmidt. Minimum disruption service composition and recovery over mobile ad hoc networks. In *International Conference on Mobile and Ubiquitous Systems: Networking Services*, MobiQuitous'07, pages 1–8. IEEE Xplore, Aug 2007.

[17] S. K. Khaitan and J. D. McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, June 2015.

[18] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. *Informatik-Spektrum*, 33(2):107–121, 2010.

[19] S. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, pages 46–53, March 2001.

[20] A. Moustafa and M. Zhang. Multi-objective service composition using reinforcement learning. In S. Basu, C. Pautasso, L. Zhang, and X. Fu, editors, *Proceedings of the 11th International Conference on Service-Oriented Computing*, ICSOC'13, pages 298–312, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-45005-1.

[21] V. Nallur, A. Elgammal, and S. Clarke. Smart route planning using open data and participatory sensing. In E. Damiani, F. Frati, D. Riehle, and A. I. Wasserman, editors, *Open Source Systems: Adoption and Impact*, volume 451 of *IFIP Advances in Information and Communication Technology*, pages 91–100. Springer, 2015. ISBN 978-3-319-17836-3. doi: 10.1007/978-3-319-17837-0_9.

[22] C. E. Osgood, G. J. Suci, and P. H. tannenbaum. *The measurement of Meaning*. University of Illinois at Urbana-Champaign Press, 2 edition, 1967.

[23] R. Popescu, A. Staikopoulos, A. Brogi, P. Liu, and S. Clarke. A formalized, taxonomy-driven approach to cross-layer application adaptation. *Transactions on Autonomous and Adaptive Systems*, 7(1):7, 2012.

[24] D. Quercia, R. Schifanella, and L. M. Aiello. The shortest path to happiness: Recommending beautiful, quiet, and happy routes in the city. In *ACM Hypertext*, HT'14, pages 116–125, New York, NY, USA, September 2014. ACM. ISBN 978-1-4503-2954-5.

[25] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang. Middleware for pervasive computing: A survey. *Pervasive Mobile Computing*, 9(2):177–200, Apr. 2013.

[26] B. Schawrtz. *The Paradox of Choice*. Harper Perennial, 2004. ISBN 0-06-000568-8.

[27] S. Schuhmann, K. Herrmann, K. Rothermel, and Y. Boshmaf. Adaptive composition of distributed pervasive applications in heterogeneous environments. *ACM Trans. Auton. Adapt. Syst.*, 8(2):10:1–10:21, July 2013.

[28] D. Shah. Gossip algorithms. *Foundations in Networking*, 3(1): 1–125, 2009.

[29] B. A. Singletary and T. E. Starner. Symbiotic interfaces for wearable face recognition. In *Human Computer Interaction International Workshop on Wearable Computing*, HCII'01, New Orleans, LA, August 2001.

[30] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM'03. IEEE Computer Society, 2003.

[31] L. Thomas, J. Wilson, G.-C. Roman, and C. Gill. Achieving coordination through dynamic construction of open workflows. In J. M. Bacon and B. F. Cooper, editors, *Proceedings of the 10t International Middleware Conference*, pages 268–287, Berlin, Heidelberg, December 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10445-9.

[32] J. Vallejos, J. Huang, P. Costanza, W. De Meuter, and T. D'Hondt. A programming language approach for context-aware mash-ups. In *Third International Workshop on Web APIs and Services Mashups*, Mashups'09. ACM, October 2009.

[33] H. Wang, X. Zhou, X. Zhou, W. Liu, W. Li, and A. Bouguettaya. Adaptive service composition based on reinforcement learning. In P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, editors, *Proceedings of the 8th International Conference on Service-Oriented Computing*, ICSOC'10, pages 92–107, Berlin, Heidelberg, December 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17358-5.

[34] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College London, London, UK, May 1989.

[35] E. Wolff. *Microservices: Flexible Software Architectures*. CreateSpace Independent Publishing Platform, January 2016. ISBN 978-1523361250.

[36] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.

[37] H. Zhao and P. Doshi. Haley: A hierarchical framework for logical composition of web services. In *IEEE International Conference on Web Services*, ICWS'07, pages 312–319, July 2007.