

Automated user-oriented description of emerging composite ambient applications

M. Koussaifi, S. Trouilhet, J.-P. Arcangeli, J.-M. Bruel
IRIT, University of Toulouse, France

Abstract—Ambient environments consist of components surrounding the user and offering services. Applications can here be composed opportunistically and automatically by an intelligent system that puts together available components. Thus, applications that are a priori unknown emerge from the environment. The problem is in the intelligible presentation to an average user of those emerging composite applications. Our approach consists in automatic generation of user-oriented application descriptions from unit descriptions of each component and service. For that, we propose a well-defined language for component description and a method for combining descriptions. A prototype has been developed and used to experiment the generation of different composite application descriptions. Based on these experiments, we assess the degree of fulfillment of the requirements we have identified for the problem.

I. INTRODUCTION

Applications of the Internet of Things, ambient and cyber-physical systems consist of fixed or mobile connected devices. Devices host independently developed and managed software components that provide services specified by interfaces and, in turn, may require other services [1]. Components are building blocks that can be assembled by binding required and provided services to build composite applications.

Due to mobility and separate management, devices and software components may appear and disappear without this dynamics being foreseen. Hence, the environment is open and its changes are out of control. Humans are at the core of these dynamic systems and can use the applications at their disposal. Ambient intelligence aims at offering them a personalized environment adapted to the current situation, anticipating their needs and providing them the right applications at the right time with the least effort possible.

We are currently exploring and designing a solution in which components are dynamically and automatically assembled to build new composite applications and so customize the environment at runtime. Our approach is rather disruptive: unlike the traditional goal-directed top-down mode, applications are built on the fly in bottom-up mode from the components that are present and available at the time, without user needs being made explicit. That way, composite applications continuously emerge from the environment, taking advantage of opportunities as they arise: for example, a slider on a smartphone can opportunistically be composed with a connected lamp and provide

the user with a lightening service when entering a room. Here, contrary to the traditional SOA paradigm, the user does not specify a service or search for it in “pull mode”, but context-adapted applications are provided in “push mode”.

Automated composition is supported by an assembly engine in line with the autonomic computing principles and the MAPE-K model [2]: it senses the existing components and decides of the connections (it may bind a required service and a provided one if their interfaces are compatible) without using a pre-established plan. The heart of the engine is a distributed multi-agent system where agents, close to the software components and their services, cooperate and decide on the connections [3]. To make the right decisions and offer relevant applications, the engine (*i.e.*, the agents) learns at runtime by reinforcement. Thus, the engine assures proactivity and runtime adaptation in the context of openness, dynamics and unpredictability.

In such a context of automation based on artificial intelligence, we believe that, whatever the engine's decisions are, the deployment of emerging applications should remain under user control. So, she/he must first be informed of the new application. Then, depending on its interest, she/he must be able to accept it or not, possibly to modify it (provided that she/he has the required skills) and so to contribute to the customization of her/his environment. So, the user must be put “in the loop” [4]. In addition, the user's actions about the emerging application (acceptance, rejection, modification) are sources of feedback for the engine's learning. Based on them, the engine builds a model of the user's preferences and habits. Unknown a priori, this model is built at runtime and evolves dynamically.

Therefore, it is essential to assist the user in the appropriation of the emerging applications pushed by the engine. For that, applications must be presented to the user in a useful and understandable way. The goal of this paper is to propose a solution to provide the user with an intelligible description of emerging applications.

The paper is organized as follows. Section II describes in more details the problem and lists the main requirements. Section III analyzes the related work on service description and shows that the solutions are very limited in relation to our problem. Section IV presents and illustrates our approach to meet the requirements. Section V describes an experiment based on a prototype we have developed. It shows the feasibility of our approach and assesses whether it meets the requirements. Finally, a conclusion is given in Section VI as well as the perspectives of this work.

II. PROBLEM STATEMENT

In the absence of prior specification, emerging applications are unknown a priori and possibly surprising. They result from local interactions between distributed agents that constitute the engine. Composition relies on learned user preferences and a matching between required and provided services.

The user must be aware of the emerging application, its function and how to use it, to consider if she/he could benefit from. Therefore, applications must be presented in an intelligible way. Here, we target average users that are not familiar to programming and computer science. For instance, the user may be the inhabitant of a smart house or a public transport traveller in a smart city.

Consider a simple assembly consisting of a switch and a lamp. In that case, we would ideally like to tell the user something like “if you click on the switch, the lamp will turn ON/OFF”. Therefore, the problem lies in the construction of the understandable description of an application defined by an assembly of software components, and to compute the description from the participating components, their services and bindings.

A. Previous work

In [4], we have proposed an architecture that puts the user in the loop (see Fig. 1). An editor presents the emerging application, allows the user to accept it, then it is deployed, or to reject it, then it is cancelled, or to modify it (that is add, remove or change bindings between services). Acceptance, rejection, and modifications are notified back to the engine for learning.

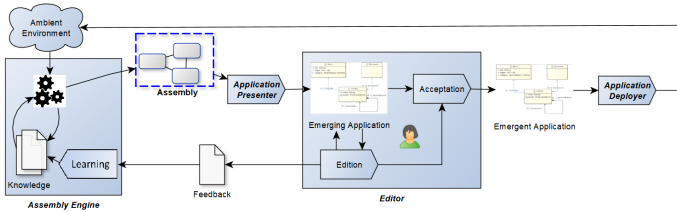


Fig. 1. Overall solution architecture

For that, the editor exposes a structural description of the application (see Fig. 2) that is an editable graph of software components that are connected through their respective services, as well as other available components that may be useful. This is achieved using model transformation techniques that transform the output of the engine (set of components, services and bindings) into a model (conforming to a metamodel we have defined for this purpose) that is presented for the user.

In the state of our work, the solution is limited to the presentation of an editable structure of the application. On one hand, this allows the user to build of a tailor-made ambient environment. On the other hand, this requires to understand component-based architecture, at least the meaning of an assembly of components. Therefore, structural presentation is not understandable by an average user,

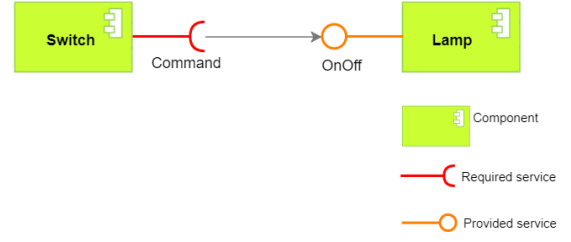


Fig. 2. Structural presentation via the editor

who needs to know the function rendered by the application (in other words its semantics), and how to use it.

The next section lists the requirements we have identified for an efficient presentation of an emerging application.

B. Requirements

1) **Semantics:** The function (*i.e.*, the semantics) of the application must be exhibited. For example, “the application allows to light up the lamp”.

2) **Usage:** The instructions on how to use the application must be exhibited. For example, “press the switch to turn ON/OFF the light”.

3) **Intelligibility:** The description must be understandable by an average user, without programming skills.

4) **Presentation scalability:** The description should remain useful and intelligible even when the application has about ten or more components.

5) **Automated processing:** The description must be automatically built by combining unit descriptions of components without human support.

6) **Expressiveness:** The description language must be expressive enough for software engineers with standard skills to make descriptions of the components they provide.

III. RELATED WORK

Fundamentally, software components and services are developed and documented for composition and reuse. In practice, they are built from scratch or from existing ones. In service-oriented engineering, service discovery and selection are fundamental operations. Selection generally aims to choose a service on a qualitative basis among several ones that have been previously discovered. Discovery and selection are performed either manually or more or less automatically, at design or runtime, from service descriptions.

In this section, we examine the related work in the field of service description. We first study the questions related to the target and objective of the description, then the questions of content and tools of description.

A. For whom and why describing a service?

Designers use service descriptions as documentation. They likewise describe the intent and use of the services they develop. When engineers specify business processes to be realized through the composition of existing services,

they describe (composite) services too. For example, composition of Web services are first explicitly specified by the service requester, then processed more or less automatically [5]. Thus, in that top-down mode, the demanded composite service is specified a priori, so no more description is necessary.

In [6], authors propose a user-centric service composition platform that assists end-users without skills in service-oriented engineering. End-users first enter their goals using a few keywords. Then an editor presents the available services and suggests possible and modifiable processes.

Most of the existing solutions use service description to support automated service discovery, selection and composition. Services are described to be processed by a program. Description allows service location and use, as is the case for WSDL [7] in the field of Web Services.

Semantic description of Web services first targets interoperability [8]. Relying on semantics also has a positive impact on the quality of the composition [9], in particular when Quality of Service (QoS) attributes are considered [10], [11]. In [12], authors propose semantic enhancement of software components with their properties and functionality to support matching between candidate components.

B. How to describe a service?

Service description can take more or less advanced forms depending on the requirements for discovery, selection and composition. In [13], authors overview and classify service description approaches used in automated service composition research.

In a basic way, descriptions may be limited to a syntactic level. For example, in object-oriented middleware like Java RMI [14], remote objects that provide services are registered and located only through a name. Services (resources in general terms) may be described more precisely using keywords in order to be retrieved by their characteristics rather than a simple identifier.

The semantic description of a service can be functional. It can take the form of a signature with inputs and outputs, possibly completed by preconditions and effects [15]. Authors of [11] explain that signature is not enough because different functions may have the same signature on the one hand, and that two services rendering the same function may differ fundamentally in their performances on the other hand. Therefore, service description may include extrafunctional properties that is QoS-related properties.

In [16], authors have proposed different techniques to create descriptions of services using the DAML-S language that was proposed to bridge the gap between the Web services infrastructure and the Semantic Web [17]. According to [13], OWL-S that succeeded to DAML-S has become a standard for industrial service composition. OWL-S [18] is an ontology for describing Semantic Web services that enables their automated discovery, composition and use. Ontology-driven description of services proved to be efficient for selection and composition [6].

C. Analysis

There are many solutions for functional and extrafunctional service description. Most of them focus on service discovery, selection, and top-down composition in order to build a complex service from unit ones. In our bottom-up approach, as the complex service to be built is unknown, there exists no solution which aims at combining descriptions. In most cases, service description supports automation, for example when based on ontologies. But in that case, descriptions are little or not at all intelligible by average human users. In addition, when extrafunctional properties are considered, they mainly concern the quality of services, but not their usage. In conclusion, to the best of our knowledge, there is no work that meets our requirements, mainly those concerning usage, intelligibility, and automated processing, in the context of bottom-up and goal-free application construction. Nevertheless, a functional description of services using signatures with preconditions and effects [15] may help in extracting the semantic information about the components' behavior and their interactions, that should be useful for the entire application description.

IV. PROPOSITION

This section presents our approach to describe components and their services, and to compute user-oriented descriptions of emerging composite ambient applications (assemblies of software components). Descriptions mainly consist of rules that explain the components and the applications. Composite application descriptions are generated from the unit descriptions of the components that are given at component design time, and the bindings between services that are supplied by the engine. Generation is achieved by combining the descriptions together, precisely the rules that belong to each unit description. At the end, the combination process aims at building a rule or a set of rules that describes the application, that can be then transformed into a text readable by the user. Our contributions are: (i) a language for the description of components' services and (ii) a combination method.

In the following, our proposition is explained in details.

A. Component and Service Description

A component description (CD) is a tuple that expresses how the component and its services work and interact with other connected components.

$$CD = \langle \text{ComponentName}, \text{Role}, \text{States}, \text{ProvidedServices}, \text{RequiredServices} \rangle$$

ComponentName and *Role* are strings: the name of the component and a free text (e.g., *ComponentName* = "Switch", *Role* = "Send a signal when pressed"). As components may have an internal state, such as a lamp that is ON/OFF, *States* is the (possibly empty) set of possible states (e.g., *States* = {"ON", "OFF"}). Last, the component's

required and provided service descriptions are gathered in the *ProvidedServices* and *RequiredServices* sets.

A service, whether provided or required, is also described by a tuple (SD).

$$SD = \langle \text{ServiceName}, \text{IOAction}, \text{Launcher}, \text{ServiceDescription}, \text{BoundTo}, \text{Rules} \rangle$$

ServiceName is a string (e.g., *ServiceName* = "Command").

IOAction represents how the service interacts with other services. It may have the following forms: *VAL@OUTPUT* or *TRIGGER ServiceName*. The first form refers to the emission of a message on the output interface of a required service. *VAL* covers all possible data types that the service handles (as the services have previously been composed by the engine, the type matching problem has been already resolved; thus types are useless for our descriptions), and may even be omitted. The second form refers to the transfer of control between services inside a component. Furthermore, *IOAction* can be empty for a provided service handling only the evolution of the component's state, without any output (e.g., the *OnOff* provided service of the lamp that changes the state to ON/OFF).

Launcher is a key defining what activates the service. It covers two cases. The first refers to an external interaction coming from another component (*onRequired*) or to an internal one coming from the component itself (*onTriggered*). The second case refers to an interaction coming from the user and can have multiple values such as *onButtonPressed*, *onSliderDragged*, *onCheckBoxChecked*...

ServiceDescription describes the service in a free text (e.g., *ServiceDescription* = "Turn ON/OFF the lamp"). This attribute is used by our combination algorithm to generate a textual form of the description (see section IV-B).

BoundTo is a set of services within their component (C-S) described as follows.

$$C - S = \text{ComponentName.ServiceName}$$

For example, $C - S = \text{Switch.Command}$.

Rules is a set of logic rules of the form "*Condition* \Rightarrow *Consequence*" that describes the service behavior. It is written as follows in BNF notation [19] where $\langle cp \rangle$ is a comparator, $S \in \text{States}$ and V is a value. Note that *IOAction* and *Launcher* have been put out of the rules for expressiveness and separation of concerns purpose.

$$\begin{aligned} & \text{Launcher} \\ & [\wedge (\text{STATE} < cp > S)] \\ & [\wedge (\text{VAL@INPUT} < cp > V)] \\ & \Rightarrow \\ & \text{IOAction} \mid \\ & \text{STATE} = S \mid \\ & \text{IOAction} \wedge (\text{STATE} = S) \mid \\ & \text{NOP} \end{aligned}$$

The common case is "*Launcher* \Rightarrow *IOAction*". Premises concerning the component's state and the inputted value are optional. It is usually the case for services that have no condition to check, other than their *Launcher*, before triggering their action. For example, for a switch to issue a command, it is only necessary that the button is pressed by the user.

In the general case, the condition part of a rule may contain a test on the component's state or on an inputted value. For the consequence part of the rule, several forms are possible. In particular, it may contain a state changing operation. Furthermore, *NOP* is a special key used if the service does not carry out any operation.

Here are examples of service descriptions of the *Command* service required by a switch and the *OnOff* service provided by a lamp, that have been connected by the assembly engine. Component and service descriptions have initially been written by the designer. They are completed by the engine by filling the *BoundTo* attribute according to the emerging assembly.

$$\begin{aligned} & \langle (\text{ServiceName}) \text{Command}, \\ & (\text{IOAction}) @\text{OUTPUT}, \\ & (\text{Launcher}) \text{onButtonPressed}, \\ & (\text{ServiceDescription}) \text{Send a signal}, \\ & (\text{BoundTo}) \{\text{Lamp.OnOff}\}, \\ & (\text{Rules}) \{\text{Launcher} \Rightarrow \text{IOAction}\} \rangle \end{aligned}$$

$$\begin{aligned} & \langle (\text{ServiceName}) \text{OnOff}, \\ & (\text{IOAction}), \\ & (\text{Launcher}) \text{onRequired}, \\ & (\text{ServiceDescription}) \text{Turn ON/OFF the lamp}, \\ & (\text{BoundTo}) \{\text{Switch.Command}\}, \\ & (\text{Rules}) \{\text{Launcher} \wedge (\text{STATE} == \text{OFF}) \\ & \Rightarrow \text{STATE} = \text{ON}, \\ & \text{Launcher} \wedge (\text{STATE} == \text{ON}) \\ & \Rightarrow \text{STATE} = \text{OFF}\} \rangle \end{aligned}$$

The next section presents the method for combining descriptions.

B. Combination of descriptions

Application descriptions are generated mainly from the rules that describe the services, and then from the remaining attributes (if used by the rules). If a service S1 is connected to a service S2 then the rules of S1 are combined with the rules of S2 to generate the rules that describe the composition.

The combination algorithm first finds matching keys available in each possible pair of rules which belong to S1 and S2 descriptions. For example: *VAL@OUTPUT* matches *onRequired* and *VAL@INPUT*; *TRIGGER Service-Name* matches *onTriggered*. Then, the algorithm infers the combined rules by transitivity. For example:

```
* R1: A  $\Rightarrow$  B  $\wedge$  C
* R2: C  $\Rightarrow$  D
R1 and R2 are combined into:
* R: A  $\Rightarrow$  B  $\wedge$  D
```

Note that there might be several combined rules at the same time, for example:

```
* R1: A  $\Rightarrow$  B
* R2: B  $\wedge$  B'  $\Rightarrow$  C
* R3: B  $\wedge$  B''  $\Rightarrow$  D
R1, R2 and R3 are combined into:
* R': A  $\wedge$  B'  $\Rightarrow$  C
* R'': A  $\wedge$  B''  $\Rightarrow$  D
```

Here is an example for the Switch-Lamp application:

```
* R': LauncherCommand  $\wedge$  (STATE == ON)  $\Rightarrow$  STATE = OFF
* R'': LauncherCommand  $\wedge$  (STATE == OFF)  $\Rightarrow$  STATE = ON
```

The combined rules are transformed into a textual form to be presented to the user. An attribute is linked to its component and the label is replaced by its content (*Launcher_{Command}* becomes *OnButtonPressed of Switch*). The *ServiceDescription* content is used to make the elements of the rule more explicit (the *STATE* element and *Turn ON/OFF the lamp* are compared). Finally, the \Rightarrow is translated in a verbal form (*IMPLIES ... IF*). In addition, the algorithm is able to group rules that have the same launcher. The textual presentation of the Switch-Lamp application is:

```
onButtonPressed of Switch IMPLIES
  Turn OFF the lamp IF Lamp is ON
  Turn ON the lamp IF Lamp is OFF
```

Note that the generated textual description may sometimes not be grammatically correct. At this point, syntax improvement is left for future work.

The above example has the simplest topology, but the solution targets more complex ones: pipeline, star... Our description language can easily be extended. For example in order to cover the case of a component that requires several services sequentially, a sequence operator could be added to the description language and handled by the combination algorithm.

The next section shows the experimentation we have carried out and analyzes our solution in relation to the requirements.

V. EXPERIMENTATION AND ANALYSIS

A. Proof of concept

In order to demonstrate the feasibility of our approach, we have developed a prototype in Java, where each component and its services XML-like descriptions are stored in a separate file. It was tested on different composite applications built by our assembly engine.

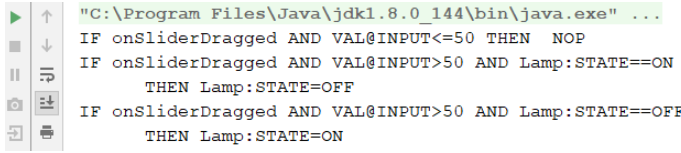
Here is an example with three components assembled in pipeline mode: a slider, a converter and a lamp. The slider acts as a switch. It requires the *ProcessVal* service. The converter provides the *Transform* service: it receives a value and, if greater than 50, transforms it into an order for the lamp through the *Order* required service. As in the previous section, the lamp provides the *OnOff* service. The descriptions of the services are given below (see Section IV-A for *OnOff*).

```
< (ServiceName) ProcessVal,
  (IOAction) VAL@OUTPUT,
  (Launcher) onSliderDragged,
  (ServiceDescription) Send a value  $\in$  [0,100],
  (BoundTo) {Converter.Transform},
  (Rules) {Launcher  $\Rightarrow$  IOAction} >
```

```
< (ServiceName) Transform,
  (IOAction) TRIGGER Order,
  (Launcher) onRequired,
  (ServiceDescription) Change value into signal,
  (BoundTo) {Slider.ProcessVal}
  (Rules) {Launcher  $\wedge$  (VAL@INPUT > 50)
     $\Rightarrow$  IOAction,
    Launcher  $\wedge$  (VAL@INPUT  $\leq$  50)  $\Rightarrow$  NOP} >
```

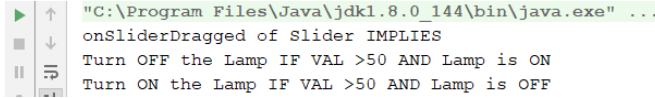
```
< (ServiceName) Order,
  (IOAction) @OUTPUT,
  (Launcher) onTriggered,
  (ServiceDescription) Send a signal,
  (BoundTo) {Lamp.OnOff},
  (Rules) {Launcher  $\Rightarrow$  IOAction} >
```

Fig. 3 shows the rules resulting from the combination algorithm. Then, the rules are transformed into a more intelligible textual version to describe the emerging application (Fig. 4).



```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
IF onSliderDragged AND VAL@INPUT<=50 THEN NOP
IF onSliderDragged AND VAL@INPUT>50 AND Lamp:STATE==ON
    THEN Lamp:STATE=OFF
IF onSliderDragged AND VAL@INPUT>50 AND Lamp:STATE==OFF
    THEN Lamp:STATE=ON
```

Fig. 3. Description's rules of the emerging application



```
"C:\Program Files\Java\jdk1.8.0_144\bin\java.exe" ...
onSliderDragged of Slider IMPLIES
Turn OFF the Lamp IF VAL >50 AND Lamp is ON
Turn ON the Lamp IF VAL >50 AND Lamp is OFF
```

Fig. 4. User-oriented description of the emerging application

B. Analysis

Rules describing the composite application are actually inferred. They provide the information about both the function of the application and how the user can interact with it. Thus, the rule-based description of the components and their services in an assembly makes possible to satisfy the main requirements we have defined (see Section II-B) concerning *semantics*, *usage*, and *automated processing*. By transforming rules into text, the understanding is made easier. Nevertheless, no real users have yet assessed the *intelligibility*. User assessment experiments could help us in improving the description language and the rule combination process. Concerning *expressiveness*, we consider that the rule-based language is expressive enough for software engineers. Moreover many elements of CD and SD could be automatically extracted from the code of components, in particular from the signatures of methods. *Presentation scalability* requirement has yet to be improved. We would like to allow the folding and unfolding of descriptions when a number of components are involved, and therefore offer a kind of “responsive” presentation of applications with different levels of abstraction.

VI. CONCLUSION

In this paper, we have exposed an approach that aims to answer the requirements to generate user-oriented intelligible descriptions of emerging assemblies of software components. We have presented the limitations of the current solutions and highlighted the benefits of our one. We have developed a proof of concept that shows that our approach can meet the requirements. Further experiments must now be carried out on more complex composite applications and topologies in order to consolidate our solution and enrich the description language. Real users should be involved in the experiments to improve and validate intelligibility and scalability of the presentation.

Our next step towards addressing the scalability issue will be to fully use the power of Model-Driven Engineering

(MDE) approaches and tools to support the automatic generation of combination algorithms from the description language definition itself. Our description language being a domain-specific language (DSL), and our input assembly being a model, MDE which has been proved useful in this particular case [20] will allow us to define transformation between assemblies and their descriptions.

REFERENCES

- [1] I. Sommerville, “Component-based software engineering,” in *Software Engineering*, ch. 16, pp. 464–489, Pearson Education, 10 ed., 2016.
- [2] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, pp. 41–50, Jan. 2003.
- [3] W. Younes, S. Trouilhet, F. Adreit, and J.-P. Arcangeli, “Towards an intelligent user-oriented middleware for opportunistic composition of services in ambient spaces,” in *Proceedings of the 5th Workshop on Middleware and Applications for the Internet of Things, M4IoT’18*, (New York, NY, USA), pp. 25–30, ACM, 2018.
- [4] M. Koussaifi, S. Trouilhet, J.-P. Arcangeli, and J.-M. Bruel, “Ambient intelligence users in the loop: Towards a model-driven approach,” in *Software Technologies: Applications and Foundations* (M. Mazzara, I. Ober, and G. Salaün, eds.), pp. 558–572, Springer, 2018.
- [5] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, “Web services composition: A decade’s overview,” *Information Sciences*, vol. 280, pp. 218 – 238, 2014.
- [6] H. Xiao, Y. Zou, R. Tang, J. Ng, and L. Nigul, “Ontology-driven service composition for end-users,” *Service Oriented Computing and Applications*, vol. 5, p. 159, Mar 2011.
- [7] “Web Services Description Language.” <https://www.w3.org/TR/wsdl/>. Accessed: 2019-01-31.
- [8] H. Nacer and D. Aissani, “Semantic web services: Standards, applications, challenges and solutions,” *Journal of Network and Computer Applications*, vol. 44, pp. 134–151, Sept. 2014.
- [9] Y. Charif and N. Sabouret, “An overview of semantic web services composition approaches,” *Electronic Notes in Theoretical Computer Science*, vol. 146, no. 1, pp. 33 – 41, 2006. Proceedings of the First International Workshop on Context for Web Services (CWS 2005).
- [10] E. Chindenga, M. S. Scott, and C. Gurajena, “Semantics Based Service Orchestration in IoT,” in *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT’17*, (New York, NY, USA), pp. 7:1–7:7, ACM, 2017.
- [11] A. Hurault, F. Camillo, M. Daydé, R. Guivarch, M. Pantel, C. Puglisi, and H. Astsatryan, “Semantic description of services: issues and examples,” *Computer Science and Information Technologies, Yerevan (Arménia)*, 2009.
- [12] J. M. Gomez, S. Han, I. Toma, B. Sapkota, and A. Garcia-Crespo, “A Semantically-enhanced Component-based Architecture for Software Composition,” in *Int. Multi-Conf. on Computing in the Global Information Technology (ICCGI’06)*, pp. 43–47, Aug 2006.
- [13] Y. Fanjiang, Y. Syu, S. Ma, and J. Kuo, “An overview and classification of service description approaches in automated service composition research,” *IEEE Transaction on Services Computing*, vol. 10, pp. 176–189, March 2017.
- [14] “Java Remote Method Invocation (RMI).” <https://docs.oracle.com/javase/tutorial/rmi/index.html>. Accessed: 2019-01-31.
- [15] M. Klusch, “Semantic web service description,” in *CASCOM: Intelligent Service Coordination in the Semantic Web*, (Basel), pp. 31–57, Birkhäuser Basel, 2008.
- [16] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid, “Composing web services on the semantic web,” *The VLDB Journal*, vol. 12, pp. 333–351, Nov. 2003.
- [17] M. Paolucci and K. Sycara, “Autonomous semantic web services,” *IEEE Internet Computing*, vol. 7, pp. 34–41, Sept. 2003.
- [18] “OWL-S: Semantic Markup for Web Services.” <https://www.w3.org/Submission/OWL-S/>. Accessed: 2019-01-31.
- [19] “Backus–Naur Form (BNF).” <https://www.w3.org/Notation.html>. Accessed: 2019-01-31.
- [20] H. Bruneliere, R. Eramo, A. Gomez, V. Besnard, J.-M. Bruel, M. Gogolla, A. Kästner, and A. Rutle, “Model-Driven Engineering for Design-Runtime Interaction in Complex Systems: Scientific Challenges and Roadmap,” in *MDE@DeRun 2018 workshop*, vol. 11176 of LNCS, June 2018.