Journal of Universal Computer Science, vol. 27, no. 7 (2021), 755-773
submitted: 14/5/2020, accepted: 26/5/2021, appeared: 28/7/2021 CC BY-ND 4.0

Model-Driven Engineering for End-Users in the Loop in Smart Ambient Systems

Sylvie Trouilhet
(IRIT, University of Toulouse, UPS, France
https://orcid.org/0000-0003-4330-5034, Sylvie.Trouilhet@irit.fr)

Jean-Paul Arcangeli
(IRIT, University of Toulouse, UPS, France
https://orcid.org/0000-0002-0521-9082, Jean-Paul.Arcangeli@irit.fr)

Jean-Michel Bruel
(IRIT, University of Toulouse, UT2J, France
https://orcid.org/0000-0002-3653-0148, Jean-Michel.Bruel@irit.fr)

Maroun Koussaifi
(IRIT, University of Toulouse, UPS, France
Maroun.Koussaifi@irit.fr)

Abstract: At the heart of cyber-physical and ambient systems, the user should permanently benefit from applications adapted to the situation and her/his needs. To do this, she/he must be able to configure her/his software environment and be supported as much as possible in that task. To this end, an intelligent "engine" assembles software components that are present in the ambient environment at the time and makes unanticipated applications emerge. The problem is to put the user "in the loop", i.e., provide adapted and intelligible descriptions of the emerging applications, and present them so that the user can accept, modify or reject them. Besides, user feedback must be collected to feed the engine's learning process. Our approach relies on Model-Driven Engineering (MDE). However, differently from the regular use of MDE tools and techniques by engineers to develop software and generate code, our focus is on end-users. Models of component assemblies are represented and made editable for them. Based on a metamodel that supports modeling and description of component-based applications, a user interface provides multi-faceted representations of the emerging applications and captures user feedback. Our solution relies on several domain-specific languages and a transformation process, based on the established MDE tools (Gemoc studio, Eclipse Modeling Framework, EcoreTools, Sirius, Acceleo). It works in conjunction with the intelligent engine that builds the emerging applications and to which it provides learning data.

Keywords: Ambient intelligence, Automated software composition, User in the loop, Application description, Model-driven engineering
Categories: D.1.2, D.2.2, D.2.11, H.1.2, I.2.2
DOI: 10.3897/jucs.70515

1 Introduction

Applications of the Internet of Things, ambient and cyber-physical systems consist of fixed or mobile, connected devices. These devices host independently developed and managed software components that may be assembled to build distributed applications. Due to mobility and separate management, devices and software components may appear and disappear without foreseeing these dynamics. Hence, the environment is open, and its changes are out of control.

Humans are at the core of these dynamic systems where they may use applications at their disposal. Ambient intelligence aims to offer them a personalized environment adapted to the current situation and their needs, i.e., to provide them the right applications at the right time, with the least effort possible.

To this end, our team is exploring and designing an approach called opportunistic software composition, which aims to intelligently and automatically assemble software components at runtime to make emerge composite applications and so customize the user ambient environment: for example, an interaction component present in a smartphone (e.g., a Slider or a Speech Recognition component), a software Converter and a connected Lamp can opportunely be assembled and provide the user with an ambient lighting control service when entering a room.

This article details the way we reinforce the place of the user in the engineering loop. First, the emerging applications must be described in a useful and understandable way to inform the user efficiently. Then, she/he must be able to configure the ambient environment and be as much as possible assisted in that task. Last, user feedback must be collected to feed the intelligent assembly process. For that, our approach relies on Model-Driven Engineering (MDE). The initial idea was to be able to quickly transform data about the emergent assemblies into graphical representations of the applications. The additional benefit is that, once you get this model representation, you can: (i) provide multiple syntactic representations of the composite application (e.g., component-based, graphical, textual); (ii) generate complementary representations (e.g., dynamic view of the application or rule-based description); and (iii) formally capture the user manipulations of the model. Our solution consists of a set of languages and transformation processes. We propose an editor that allows the user to be aware of the emerging applications, to understand their function and use, and to modify them if desired. From her/his actions, without overloading her/him, feedback data are extracted. The originality of our approach is that the models are dedicated to end-users, not to designers and developers. It brings both new advantages and new challenges that we detail in this paper.

The remaining of this article is organized as follows. Sec. 2 briefly introduces model-driven engineering (MDE), component-based software engineering (CBSE), and opportunistic software composition. Sec. 3 analyzes the problem and the requirements. In Sec. 4, the motivations for using MDE are examined, then our contribution is detailed and illustrated using an application example. Our prototype implementation and validation are described in Sec. 5 where other application examples are presented. Sec. 6 analyzes the related work. At last, in Sec. 7, the contribution is summarized, and some future works are discussed.

2 Background

2.1 Model-Driven Engineering (MDE)

MDE is a software development methodology that focuses on creating and exploiting domain models related to a specific problem. Software developers (SDs) use them to create abstract descriptions of the software, facilitating the generation of implementation code [Combemale et al., 2016]. MDE is based on modeling languages, typically defined as metamodels, used to formalize application requirements, structures, and behaviors within a particular domain. Formal rules can be defined to verify that the instances comply with the metamodel. These rules are added to the metamodel to perform model-checking and detect and prevent many errors before code generation. Most of the time, the abstract syntax is defined using metamodeling by specifying the structure of the modeled system (e.g., building a class diagram that characterizes logical object structures as illustrated in Fig. 4). Metamodeling is also used to define the concrete syntax and customize the generated code.

Substancial benefits of MDE set in: (i) model transformation engines that are used to produce various types of artifacts, such as source code, deployment descriptors, or other models; and (ii) the possibility to define Domain-Specific Languages (DSLs). A DSL is a dedicated language devoted to expressing and solving problems in a specific domain [Baciková et al., 2013]. By defining the actions that can be done by SDs, it allows them to manipulate and edit particular models. Finally, to manage models, SDs use model editors, that may be graphical or textual depending on the DSL.

2.2 Component-Based Software Engineering (CBSE)

Software components are loosely coupled runtime entities that may provide services specified by interfaces and, in turn, may require other services [Sommerville, 2016]. Unlike objects, they bring the *required services* at the same level as the *provided* ones. Fig. 1 shows a Converter component represented in UML: provided services (here, the Convert service) are pictured by a bullet and required services (here, the Order service) by a socket. Components, whose implementation is hidden, are reusable building blocks. To build applications, they are *assembled* by binding required to provided services if they match. CBSE involves both developing components individually and building assemblies by using or reusing components. Usually, a middleware supports component deployment and integration.

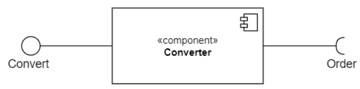

Figure 1: UML representation of a Converter component

2.3 Opportunistic Software Composition

Opportunistic software composition is a disruptive approach for dynamic and automatic construction of component-based applications: unlike the traditional goal-directed top-down mode, applications are built on the fly in a bottom-up way from the components that are present and available at the time, without the user needs have to be made explicit [Triboulot et al., 2015]. That way, applications that may be unanticipated emerge from the environment, taking advantage of opportunities as they arise. Let us imagine Paul who lives in a smart home with software components to control elements such as switches, lamps or kitchen appliances. Paul is often impressed by the ability of his environment to intelligently assist him in his activities. Yesterday, for example, his professional laptop he had only brought for teleworking served as his intercom when the postman rang.

Opportunistic software composition is supported by an intelligent middleware, called Opportunistic Composition Engine (OCE), in line with the autonomic computing principles: OCE senses the existing components, plans component assemblies (i.e., builds models), and pushes them to the user; then, assembly plans may be realized under user control. In the absence of prior explicit guidelines, OCE automatically learns the user's preferences according to the situation to later make relevant decisions and maximize user satisfaction. Learning is achieved online by reinforcement [Sutton and Barto, 2018] from feedback of the user who is put in the loop. This way, the engine assures proactivity and runtime adaptation in a context of openness, dynamics, and unpredictability. How OCE works is detailed in [Younes et al., 2020]; it is out of scope of this article.

3 Problem statement and requirements

In the absence of prior specification, emerging applications are unknown *a priori* and possibly surprising. While their functionalities may be implicit, they can also be difficult to understand. For example, Paul uses a slider on his smartphone to turn on the lamp in his living room. The first time he was offered this application, he did not immediately grasp its purpose and use.

Yet, the user must be aware of the emerging applications, arbitrate on them depending on she/he could benefit from, and provide feedback to the intelligent middleware. For that, the user must be put "in the loop".

The fundamental question is, therefore, how to realize the user in the loop and meet the corresponding requirements? The rest of this section analyzes and sets out these requirements. The next section describes our solution based on MDE.

3.1 Presentation

OCE assembles software components that are present in the ambient environment at the time and makes applications emerge. As applications may be unknown by the user, she/he must be informed of both the function of an emerging application and how to use it.

• **[R1.1] Functional description.** The function of the application must be presented to the user. For example, "The application allows someone to light up the lamp".

• **[R1.2] Usage.** The instructions on how to use the application must be presented to the user. For example, "Press the switch to turn ON/OFF the light".

3.2 Understandability

Depending on the user skills, presentation and assistance to her/him may be more or less efficient. However, a sound understanding by the user of the presented applications is critical, both for their acceptance and use and for the quality of the user feedback. Here, we target average users that are not necessarily familiar with programming and CBSE. For instance, the user may be the inhabitant of a smart house (such as Paul) or a public transport traveler in a smart city. Consider a simple assembly consisting of a switch and a lamp. In that case, we would ideally like to tell Paul something like "If you click on the switch, the lamp will turn ON/OFF". Another question is related to the complexity of the assembly in the number of components and services.

• **[R2.1] Intelligibility.** The application description must be understandable by an average user without programming skills.

• **[R2.2] Presentation scalability.** The description should remain intelligible and useful even when the application has about ten or more components.

3.3 Automated description

The problem lies in the construction of the description of an application, i.e., its computation from the components of the assembly that implements the application, their services and bindings, without human support.
• **[R3] Automation and composability.** Descriptions must be automatically built by combining unit descriptions of components.

3.4 User input and guidance

User control on their environment is of the highest importance [Bach and Scapin, 2003]. Thus, whatever OCE decisions and the pushed applications are, application deployment must remain under user control. Besides, users should be able to customize the ambient environment by themselves, depending on their needs and preferences, and be guided in this task.
• **[R4.1] Input.** The user must be able to *accept* the emerging application **[R4.1.1]**, then it is deployed, or *reject* it **[R4.1.2]**, then it is canceled. According to her/his skills, the user should be able to *edit and modify* an application model **[R4.1.3]**, i.e., create, remove, or change bindings between services.
• **[R4.2] Guidance.** When editing, the user must be guided, and the correctness of her/his actions must be guaranteed.

3.5 User feedback

Relevance of the applications pushed by OCE depends on the knowledge about the user. This knowledge is built by OCE at runtime and evolves dynamically. To learn from and for the user, OCE needs her/his feedback about the applications. But the user must not be overburdened or disturbed. User acceptance, rejection, and modification are the sources of feedback that are expected by OCE about its decisions. Note that a sound understanding by the user about the proposed application ([R2.1]) is mandatory to provide relevant feedback.
• **[R5.1] Feedback generation.** OCE must get feedback generated from the user's reactions to the emerging applications.
• **[R5.2] Discretion.** The provision of feedback should not overburden the user.

4 Putting the user in the loop with MDE

4.1 Overview

To put the user in the loop and meet the corresponding requirements, we have designed the Interactive Control Environment (ICE) based on MDE, which works together with the Opportunistic Composition Engine (OCE).

Fig. 2 shows a screenshot of the ICE user interface. It is a graphical editor that mainly consists of three panels. Panel 1 displays applications as UML component diagrams. Available components that do not participate in applications are displayed too. Through

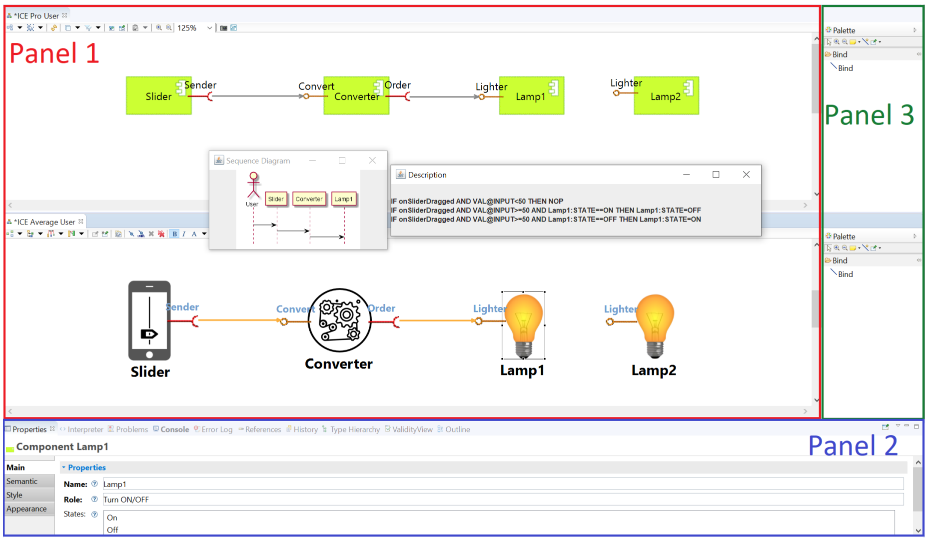

Figure 2: The Interactive Control Environment (ICE)

this panel, the user can accept or reject the assembly. This is an answer to [R4.1.1] and [R4.1.2] requirements. Besides, users that are unfamiliar with component diagrams can request other views of the application: here, a UML sequence diagram and a textual description make more explicit the function of the application and how to use it. This answers to [R1.1] and [R1.2]. However, depending on the user's skills, other outputs are possible in specific languages adapted to the user as shown in the icon-based diagram. This answers to [R2.1]. As the different outputs are computed automatically without user involvement, [R3] is met too.

Moreover, ICE supports application edition by the user. Panel 2 displays the properties of a component or a service. Panel 3 provides a *bind* tool for creating bindings between services and so components. The user can also change or delete bindings. This answers to [R4.1.3] and [R4.2] since the editor authorizes only correct user's actions. Last, to meet [R5.1], the user's activities on the assembly are captured by ICE, then transformed into feedback and sent to OCE for learning. Note that in such a way, the user is not asked for explicit feedback, so [R5.2] is satisfied.

ICE is part of the overall architecture (see Fig. 3) we have designed to put the user in the loop: ICE automatically presents emerging application descriptions and captures feedback; besides, the user can edit the assemblies and finally accept them.

4.2 Motivations for using MDE

At this point, several observations can be made:

1. Basically, OCE designs and produces models. These are models of emerging applications in the form of assemblies of components, all conforming to the same metamodel. Such a model consists of a set of components and a set of bindings between their services. It is prescriptive as it specifies the application to be created but also descriptive of the application to be accepted and deployed [Ludewig, 2003].

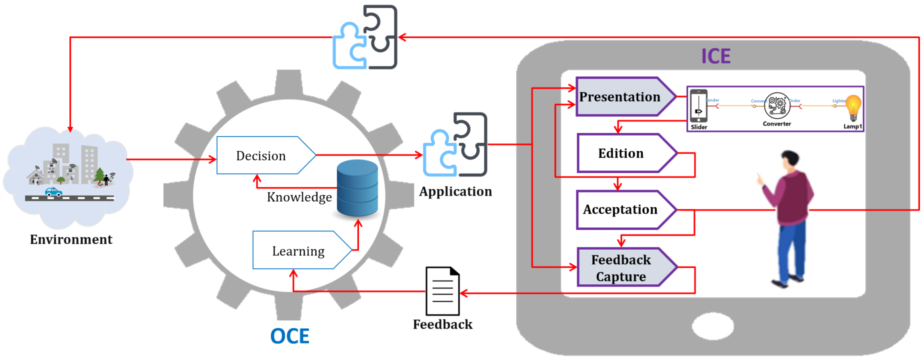

Figure 3: Overall architecture with the user in the loop

2. The models provided by OCE can be transformed and presented in multiple ways: in different user-friendly descriptions depending on her/his understanding skills, and as component diagrams to support component (re)composition by end-users who have programming skills.

3. As a programmer, the end-user must be supported with editing features and control rules that prevent unauthorized bindings. Since the models are subject to user's modifications, they can also be qualified as explorative [Ludewig, 2003].

Therefore, we have chosen MDE and the advantages it brings (see Sec. 2.1) to support ICE, i.e., to transform the models designed by OCE into user-oriented models, and conversely transform user-oriented models into both data for OCE learning and deployment scripts. Using MDE for this purpose is unusual: differently from the common use of MDE tools and techniques by engineers to develop software and generate code, we focus on end-users for whom prefabricated assembly models are transformed and represented. To develop our solution, we have used the GEMOC facilities [The GEMOC Initiative, 2021], for which our team is an active contributor. ICE consists of an Eclipse Modeling Framework (EMF) [Eclipse Foundation, Inc., 2021a] project, where EcoreTools [Eclipse Foundation, Inc., 2013] is used to define our metamodel and attach OCL rules to it. We have used Sirius [Eclipse Foundation, Inc., 2021b] to define the DSLs and create the graphical editor, and Acceleo [Eclipse Foundation, Inc., 2020] to support model transformations.

In the following, we expose the different elements of our solution.

4.3 The assembly metamodel

Several metamodels of components and services exist in the literature [Tigli et al., 2009, Janisch, 2010], but they both are too complicated for our needs and do not meet our description requirements. Therefore, we have defined our own (see Fig. 4), in line with what has been introduced in Sec. 2.2. It consists of an *Environment* which contains *Components*. To be composable, a component has at least one service (a *RequiredService* or a *ProvidedService*). A service has one property, *boundTo*, that defines the bindings to other services once the assembly has been built. Also, OCL invariants describe restrictions on the models, e.g., two required services (or two provided services) cannot be bound together. OCL rules assure that applications are well-formed when emerging or after user input, before their description and deployment.

The *Profile* attribute defined in the *Service* class is used to add controls on the edition process. It ensures that: (i) only compatible services are bound together; and (ii) that the maximum number of bindings of a service is respected.

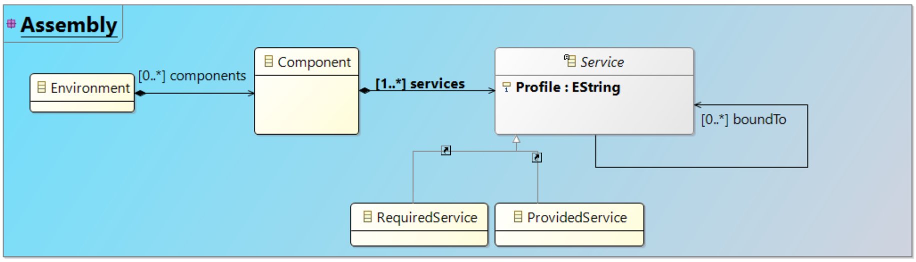

Figure 4: Assembly metamodel

The assembly model is part of a more complex metamodel (see Sec. 4.7.2) we have designed to support applications in whole and their transformations into user-oriented descriptions.

4.4 Transformation of OCE outputs to ICE models

To be presented, an OCE output must first be transformed into an internal manipulable and editable form (transformation is a consequent requirement of [R1.1] and [R1.2]). By model-to-model transformation, the model provided by OCE is transformed into an ICE XML-based model, both being compliant to the assembly metamodel. Then, the ICE model can be injected into the editor, to be then displayed and possibly modified. This ICE model is also the basis for the processes listed below.

4.5 Assembly representation via domain-specific visual languages

ICE models represent component assemblies that software engineers commonly draw as UML component diagrams. To do that and provide users with UML-like representations of the emerging assembly, we have defined a Domain-Specific Visual Language (DSVL) that complies with the UML components notation.

At the top of Panel 1 in Fig. 2, there is the editable representation of the entire assembly that achieves the ambient lighting application. For that, a visual representation of each element of the metamodel has been defined. Also, pre- and post- conditions have been incorporated in the DSVL to add control on the edition process, allowing the user to be guided and supervised when editing applications (e.g., the user cannot connect two required services). They have been implemented thanks to the support of the GEMOC infrastructure and hence automatically enforced by the generated modeler. These conditions are verified on the fly at edition time, so before the OCL rules, to prevent validation errors. This is part of the answer to [R4.2].

Nevertheless, the average user is unfamiliar with component diagrams. In [Abrahão et al., 2017], S. Abrahão *et. al.* claim that a box in which "Cat" is written –maybe "Lamp1" in our example– may be understood as a "cat" for a software engineer –as a lamp for us–, but it is still a "box" for most people! To address this problem, the above solution can easily be adapted to customize the presentation to the user by replacing the UML representation of a component by an icon that is more explicit for the general public. An icon-based representation of the ambient lighting application is displayed in Fig. 2 at the bottom of Panel 1. Such a more intelligible description allows the average user to understand the application better. It contributes to meet the intelligibility requirement [R2.1]. In this way, other DSLs can be proposed that fit users' skills. The following sections propose complementary answers to meet [R2.1].

4.6 Transformation of ICE model into a sequence diagram

To better inform the user, we explicit how the control passes from a component to another. For that, the ICE model with its bindings between services is transformed into a control model. This model represents how control passes through the components. It is then presented as a UML sequence diagram that shows which service the user directly controls, and how the services interact.

To display the sequence diagram, we coupled ICE with PlantUML editor [PlantUML, 2021], which provides graphical representations of UML sequence diagrams from a text describing the sequence. First, a model-to-text transformation builds a PlantUML-compliant textual model of the application control. Then PlantUML editor is called by ICE to create an image of the sequence diagram (see Fig. 2, Panel 1).

Other types of diagrams could be produced, e.g., UML communication diagrams, to provide complementary views of the application. With ICE, depending on her/his preferences, the user can select the type of diagram to generate and display.

4.7 Generation of rule-based application description

Software components transform inputs into inner effects or outputs. Inputs take place when a provided service is required with its parameters or when an internal action or event occurs, e.g., the user moves a slider or pushes a button, or a sensor takes a measurement. Inner effects are changes of a component inner value or state, e.g., a lamp lights up. Outputs are demands of external services. These transformations can be expressed by rules describing how inputs are transformed into effects and outputs. Expressing them is in charge of the component providers. Then, when a component becomes part of an assembly, its rules are automatically combined with the rules of the components it is connected to. For a given assembly, the combinations produces a set of rules that describes how the entire application works.

4.7.1 Description using rules

We have defined the rule-based description language in [Koussaifi et al., 2019]. In accordance with Ghiani *et al.* [Ghiani et al., 2017], rules conform to the trigger-action style because of "its compact and intuitive structure". As illustrated in Fig. 5, rules are attached to the services. For instance, for the Slider component, when the user sets a value by moving the slider, the Sender service is called with this value. In that assembly, the Convert service of the Converter component provides the Sender service that is required by Slider. When needed, depending on the parameter value (more or less than 50), the required Order service may be triggered. The principle is the same for Lamp1, but the

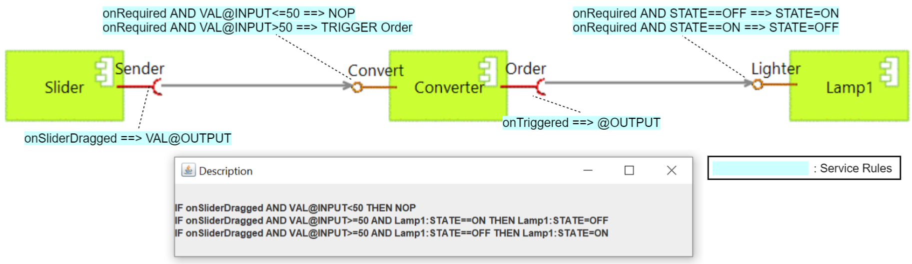

Figure 5: Description rules of the ambient lighting application

effect of the Lighter service call depends on the value of the STATE internal variable that is changed accordingly.

The pop-up window (in the "Description" frame of Fig. 5) shows a pretty-print of the automatically combined result, i.e., the rule-based description of the ambient lighting application that is the function and how to use it. When two services are bound together, their respective rules are automatically combined in pairs to reduce each pair to a single rule. The latter is then combined with the rules of the next component, and so on. The combination scheme is inspired by the *cut rule* in mathematical logic: from the rules $[\Gamma \Rightarrow A, \Delta]$ and $[\Gamma', A \Rightarrow \Delta']$, the rule $[\Gamma, \Gamma' \Rightarrow \Delta, \Delta']$ is inferred. The combination is conditioned by a matching between predefined keywords, e.g., *X@OUTPUT* matches *Y@INPUT*, *X* and *Y* being variable names. In our example, combining the Sender rule with the two Convert rules produces two rules. One indicates a lack of effect (NOP). The other, which indicates the request for the Order service, is combined with the rule that describes Order. It is combined with those of the Lamp1 Lighter service, producing the two rules that describe the practical effect on Lamp1 if the slider has been set over 50.

Note that the ambient lighting application conforms to a particular architectural style that is called "Pipe & Filter". However, our solution deals with other composition styles with components that require several services, maybe in sequence, in parallel, or conditionally, and also get and use a result from their requests (see Sec. 5.2 for some examples).

4.7.2 The full metamodel

To achieve the description, the assembly metamodel has been extended to address the description issues. The entire metamodel is shown in Fig. 6.

Every component and service has one description. Therefore, the *Component* and *Service* classes of the *Assembly* package are composed of their respective description class of the *Description* package. *ComponentDescription* consists of three attributes. *Name* and *Role* are strings, the latter being a free text describing the component, e.g., *Name* = "Slider", *Role* = "Send the specified value when the slider is moved". As components may have an internal state, such as a lamp that is ON or OFF, *States* defines the (maybe empty) set of the possible states, e.g., *States* = {"ON","OFF"}. *ServiceDescription* consists of four attributes. *Name* is a string, e.g., "Sender". *Launcher* is a key defining what activates the service. It may refer to an external interaction coming from another component, e.g., *onRequired*, or to an internal one coming from the component itself, e.g., *onTriggered*. For HCI components, it may also refer to user's interaction types and take different values, e.g., *onSliderDragged* for a Slider. *IOAction* represents how the service acts

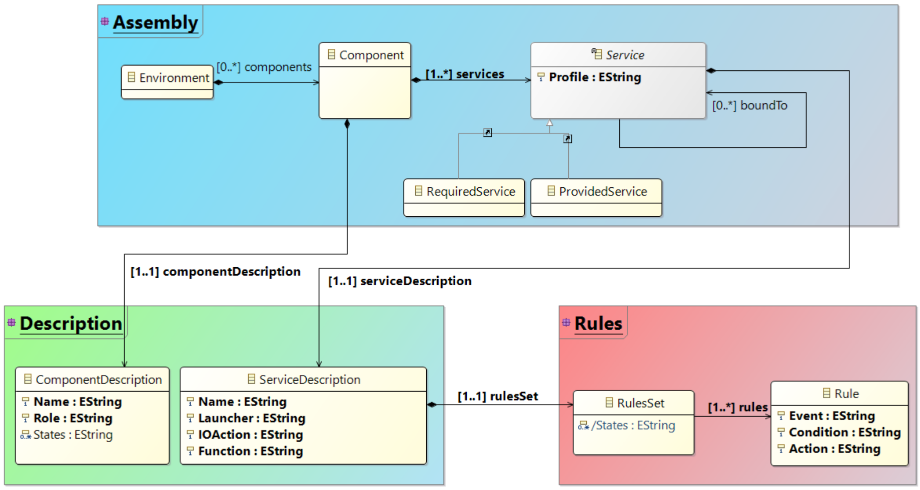

Figure 6: The extended metamodel

on other services, e.g., *VAL@OUTPUT* means that *VAL* is given as parameter of a request. *IOAction* can be empty for a provided service handling only an inner effect on the component, e.g., the Lighter service that only changes Lamp1 state. Last, *Function* describes the service as a free text, e.g., "Turn ON/OFF".

In addition, to generate application descriptions, a service description contains a non-empty *ruleSet*. A *Rule* conforms to the ECA model [Berndtsson and Mellin, 2009]: *Event* represents what triggers the service; *Condition* is a logical expression e.g., "VAL@INPUT<=50" or "STATE==OFF" (STATE is a keyword whose value belongs to *States*; *States* value comes from the owner component description); if *Condition* is true, the *Action* which represents the rendering of the service, e.g., "@OUTPUT" or "STATE=ON", is carried out. Note that *Event* and *Action* can directly refer to *Launcher* and *IOAction*, which are set in the service description.

4.8 Model comparison for feedback generation

Once the user has accepted, modified or rejected the edited assembly, feedback is generated to be given back to OCE. This meets [R5.1] and [R5.2].

The feedback is computed from the initial ICE model (the emerging application model) and the final ICE model (the application model after user inputs). The two XML-based versions of these models are compared so that a list of similarities and differences (in term of bindings between the services and components) is extracted. More precisely, it lists the bindings that have been modified, deleted, and added, and the components that have been removed or added to the assembly. This list is then used to give to OCE a positive (respectively negative) reinforcement signal for the bindings that have been accepted (respectively rejected) by the user [Younes et al., 2020]. Thanks to this signal, OCE will propose more pertinent applications in the future.

4.9 MDE for machine learning

Basically, machine learning [Mitchell, 1997] aims at building knowledge, i.e., models or patterns of reality. These models are automatically and iteratively constructed in the training phase from training data. The goal is to improve later, in the exploitation phase, the learner's behavior.

In Sec. 4.8, we have shown how model comparison supports feedback generation for OCE learning. Here we go beyond the requirements listed in Sec. 3 and take advantage of MDE facilities to design a complementary learning mode.

The idea is to provide OCE with ready to use assembly "plans". The problem is to inject these plans into OCE, which was not designed to manipulate such artifacts. For that, the principle is to reify assembly plans when accepted by the user, i.e., generate special components called connectors. There is no business logic in a connector. It implements the Mediator design pattern [Gamma et al., 1994], and as such, centralizes the interaction logic between the components: it routes service requests from the caller component to the callee and the results in the opposite direction. For that, it gathers all the provided and required services of all the components involved in the assembly. In the connector-based equivalent assembly, the direct links between the components are replaced by links between each component and the connector. Fig. 7 shows the connector-based assembly of the ambient lighting application: it is equivalent to that of Fig. 2 but interaction between the components is centralized.

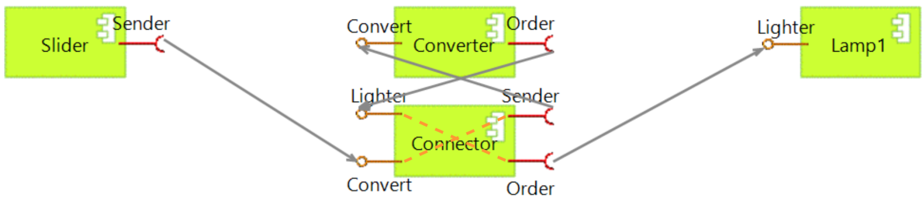

Figure 7: Connector-based assembly of the ambient lighting application

First, using the EMF model factory, a model of the connector is generated from the provided and required services involved in the whole assembly. Then, a model-to-text transformation generates the connector implementation code (Java code in the current version) and a script for its deployment in the environment. After deployment, it may be sensed by OCE and treated as an ordinary software component.

5 Implementation and validation

A prototype version of ICE has been implemented and runs coupled with OCE. ICE validation has two objectives: first, to verify that it actually provides descriptions for different assembly topologies; then, to verify that it is operational in real-life situations, with software components actually deployed on several devices.

5.1 Implementation

ICE implementation relies on the Gemoc studio, Eclipse Modeling Framework, Ecore-Tools, Sirius, and Acceleo. A stable and usable version is available on GitHub[1]. ICE provides multi-view application representations. It allows to graphically present application structures by means of UML-like and icon-based DSVLs. In addition, how applications work can be described using PlantUML sequence diagrams or in the rule-based language. Besides being informed, the user can modify, accept or reject applications. Furthermore, ICE and the intelligent engine (OCE) have been integrated to achieve fully operational system. Associated with a simulated ambient environment, it works in a loop as shown in Fig. 3: applications built by OCE are given as input to ICE, which provides user-friendly editable descriptions, extracts user feedback and supplies it back to OCE. ICE is today used in our team to, among other purposes, experiment and validate the OCE solution for learning.

5.2 Description of applications involving different topologies

We have experimented description of component-based applications arranged according to different architectural styles, i.e., different ways to associate components: “Pipe & Filter” like in the ambient lighting application, “Call & Return”, sequential requesting, parallel requesting, etc. For these different cases, on the basis of “test” application models, we have checked: (i) effectiveness and rightness of application structural description using different DSVLs (ii) user ability to edit and modify the structural representations (iii) operationality of transformations in UML sequence diagrams and rules-based descriptions (iv) correctness of user feedback extraction and delivery to OCE.

Intelligibility has been assessed with IT specialists, but so far, we have not asked average users for their opinion. Besides, scalability of the descriptions [R2.2] has yet to be tested.

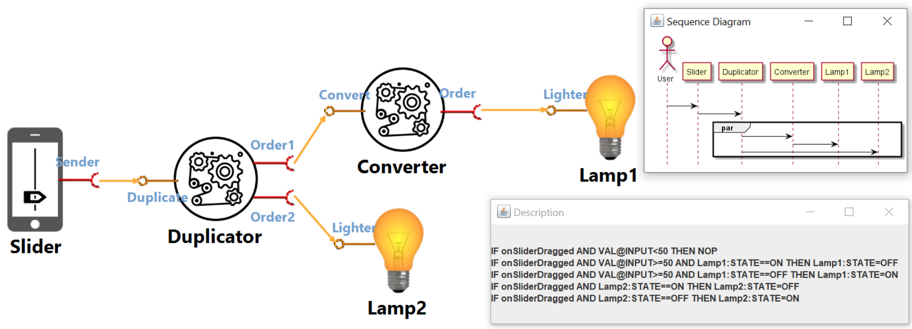

Figure 8: The parallel ambient lighting application

Fig. 8 shows an application whose architecture mixes “Pipe & Filter” and parallel requesting styles: here, the Duplicator component demands the Order1 and Order2 services in parallel. So, this application allows the user to turn on/off the two lamps at

[1] https://github.com/marounkoussaifi/ICE.git

the same time by moving the slider. As depicted in Fig. 8, in addition to the structural description, ICE provides a PlantUML sequence diagram with the “par” operator, which expresses the parallelism, and the rule-based description where parallelism is implicit (the default meaning is that all rules are triggered at the same time). In the application sketched in Fig. 9, the Sequencer component first requests Order1 and next Order2. To express the sequence, the NEXT operator is used; it separates sets of rules that are triggered successively.

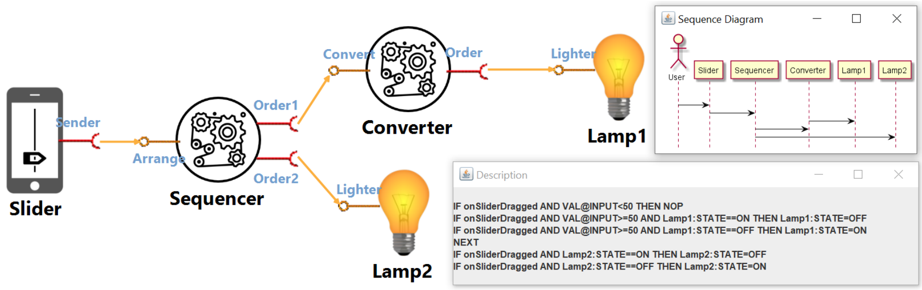

Figure 9: The sequential ambient lighting application

Although the forms of the descriptions are perfectible, especially the one based on rules, experimentation has proven the viability of our approach to realize the “user in the loop” and meet its requirements. MDE effectively supports user-oriented presentation and controlled edition of application models, as well as unobtrusive extraction of user feedback. Model transformation techniques provide multi-view representations of applications that emerge or that the user modifies or builds from scratch. MDE handles DSL definition and easy switching from a DSL to another, so as outputs can be customized for the user.

5.3 Realistic use case

We also have experimented a more realistic use case [Delcourt et al., 2021]: in his electric car, Paul benefits from an emerging application that guides him to the charging station closest to his current position. For that, actual software components have been developed such as GPS components, battery level sensors, a power station localizer, and an Android map. When deployed, components are discovered connected via the “Universal Plug and Play” (UPnP) protocol, then selected by OCE and arranged in a fully operational application running on a smartphone[2]. This experience has shown that ICE correctly presents the different applications that are built as the environment changes, and that it actually supplies feedback data to OCE.

[2] A demo is available: https://www.irit.fr/~Sylvie.Trouilhet/demo/outletSeeking.mp4

6 Related work

6.1 Description in service- and component-based engineering

The main purpose of software components and services are composition and reuse. Designers use their descriptions as documentation, which details their intent and use. When engineers specify business processes through service composition, they describe (composite) services, too, before they are processed more or less automatically [Sheng et al., 2014]. Thus, in the traditional top-down mode, demanded composite services are specified at the beginning, so there is no need to produce descriptions afterward, unlike in the case of opportunistic bottom-up composition.

Service description supports automated service discovery, selection and composition [Fanjiang et al., 2017]. In that case, descriptions are processed by a program. Descriptions allow service location and use, as is the case for WSDL [W3C, 2007] in the field of Web Services. Descriptions can take more or less sophisticated forms depending on their use. They may only be purely syntactic, e.g., in object-oriented middleware like Java RMI [Oracle, 1995]. Semantic descriptions may be limited to functional signatures with inputs and outputs, possibly extended with preconditions and effects [Klusch, 2008]. In addition, they may include the expression of extra-functional properties, i.e., QoS-related properties.

As solutions for service or component descriptions that are not user-oriented, they do not meet our presentation and understandability requirements. Moreover, we do not know any solution that satisfies the automation required to build application or service descriptions from the unit descriptions of their components.

6.2 User in-the-loop and end-user development

Even in self-adaptive systems, humans must be in the loop to cope with conflicts and improve adaptation strategies, and a trade-off should be made between autonomy and human involvement [Gil et al., 2016]. In [Evers et al., 2014], the user in-the-loop can set her/his preferences to configure and adapt existing component-based applications. User's preferences and profiles can be also be learned by semi-supervised reinforcement [Karami et al., 2016].

End-user programming proposes a set of techniques that enable end-users to create applications for personal use [Barricelli et al., 2019]. It is part of end-user development (EUD) that involves users in development at both design and operating time since "regular development cycles are too slow to meet users' fast changing requirements" [Paternò, 2013]. The AppsGate client-server system [Coutaz and Crowley, 2016] empowers end-users with human-machine interfaces to configure and control their smart home. Home inhabitants use visual and pseudo-natural languages to program their ambient environment and add or remove appliances on the fly. They are both assisted by the proposal of possible options and prevented from making errors. In conclusion, the authors state that "it should be possible to augment EUD with machine learning"; then, the behavior of the inferred services must be "understood by the user and adaptable using the EUD". In [Xiao et al., 2011], end-users are assisted in service composition by an editor that allows them to specify goals; here, keyword-based descriptions of available services and ontologies support generation of *ad hoc* processes that can be customized by end-users.

Thus, there is little or no intelligence (and no emergence) in existing EUD solutions, which do not offer flexibility and customization of descriptions unlike our MDE-based solution.

6.3 Contribution of MDE

Models allow humans to abstract from low-level features and get closer to their business domain. MDE promotes less code and an increased level of abstraction. Since its appearance, it has mainly been used by engineers to guide and support quality software construction.

Combined with CBSE, MDE should help to master the complexity and dynamics of modern software systems [Ciccozzi et al., 2019]. MDE can also support development, deployment and runtime adaptation. Based on UML models and transformations, MDE4IoT [Ciccozzi and Spalazzese, 2016] assists engineers when designing IoT applications, providing them with abstraction and separation between functional and operational concerns, and supports runtime evolution. In [Hussein et al., 2017], system functionality and adaptations are modeled as a state machine, then model-to-model and later model-to-text transformations support platform-specific code generation. MDE and Models@Runtime also support the design of feedback loops and their execution at runtime in self-adaptive software [Vogel, 2018].

Anyway, the role of MDE in the future of IoT and smart systems is still an open question [Bucchiarone et al., 2020]. As far as we know, existing MDE-based approaches do not target end-users as we do. However, end-users can participate in software modeling in cooperation with software professionals, provided they have development skills [Pérez et al., 2011]. In [Evers et al., 2014], a UML profile allows capturing variability in a human-readable way that is understandable by non-experts. Furthermore, MDE contribution to the design of user interfaces and their adaptation at runtime is analyzed in [Coutaz, 2010]. In [Abrahão et al., 2017], S. Abrahão *et. al.* introduce the concept of User eXperience (UX) for MDE: they analyze the challenges and list future work on MDE to meet UX requirements, including identify who the users are, customize tools for domain specifics, and adapt them for acceptability. Besides, some works explore the use of MDE to increase end-user involvement in the use of an application, for example, in application gamification [Bucchiarone et al., 2019].

7 Conclusion and open issues

Ambient systems, with their dynamics and unpredictability, do not allow software design to follow traditional development cycles. End-user involvement enables on-the-fly creation of software adapted to the situation and the user's preferences and skills. Nevertheless, this user must be supported and helped in the development task. Hence, an intelligent system builds and makes emerge relevant applications that the user has not explicitly asked for nor expected. This article describes an original approach and its framework for presenting emerging applications to the user in an intelligible and manipulable way. It shows how MDE techniques with the definition of dedicated languages help to provide the user with personalized views of applications, as well as the tools that allow them to be modified or even built from scratch. Thus, MDE supports both the controlled construction of applications and the production of descriptive material. Besides, by comparing and transforming models, feedback is captured and provided to the intelligent system to feed its learning process. Therefore, our original MDE-based approach puts the end-user "in the loop" by giving her/him direct access to the handling of internal application models.

Using MDE techniques and tools, we have implemented and experimented with a fully operational prototype version of ICE[3]. To strengthen our case, intelligibility and scalability of descriptions should be assessed, in particular with average users. Depending on them, other informative views could be provided. A possible lead we plan to explore is model animation [Combemale et al., 2008].

Our prototype framework has several limitations that we intend to remove. For example, to simplify rule combinations, we assume that descriptions are provided using a unique vocabulary. But components may be provided by different suppliers who do not share this vocabulary. For that, we are exploring the use of ontologies to align the terms used in the rules [Alary et al., 2020]. More experiment need now to be conducted both in terms of scalability and of stressing the framework with more complex combinations of components.

References

[Abrahão et al., 2017] Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., and Whittle, J. (2017). User experience for model-driven engineering: Challenges and future directions. In 20^{th} ACM/IEEE Int. Conf. on MDE Languages and Systems, pages 229–236. IEEE Computer Society.

[Alary et al., 2020] Alary, G., Hernandez, N., Arcangeli, J.-P., Trouilhet, S., and Bruel, J.-M. (2020). Comp-O: an OWL-S Extension for Composite Service Description. In 22^{nd} Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW), pages 171–182.

[Bach and Scapin, 2003] Bach, C. and Scapin, D. (2003). Adaptation of ergonomic criteria to human-virtual environments interactions. In Proc. of Interact'03, pages 880–883. IOS Press.

[Baciková et al., 2013] Baciková, M., Porubän, J., and Lakatos, D. (2013). Defining Domain Language of Graphical User Interfaces. In 2^{nd} Symposium on Languages, Applications and Technologies, volume 29 of OpenAccess Series in Informatics (OASIcs), pages 187–202.

[Barricelli et al., 2019] Barricelli, B. R., Cassano, F., Fogli, D., and Piccinno, A. (2019). End-user development, end-user programming and end-user software engineering: A systematic mapping study. Journal of Systems and Software, 149:101 – 137.

[Berndtsson and Mellin, 2009] Berndtsson, M. and Mellin, J. (2009). ECA Rules, pages 959–960. Springer. https://doi.org/10.1007/978-0-387-39940-9_504.

[Bucchiarone et al., 2020] Bucchiarone, A., Cabot, J., Paige, R., and Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling, 19(1):5—13.

[Bucchiarone et al., 2019] Bucchiarone, A., Cicchetti, A., and Marconi, A. (2019). Exploiting Multi-level Modelling for Designing and Deploying Gameful Systems. In 22^{nd} ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2019), pages 34–44. IEEE Computer Society.

[Ciccozzi et al., 2019] Ciccozzi, F., Carlson, J., Pelliccione, P., and Tivoli, M. (2019). Editorial to theme issue on model-driven engineering of component-based software systems. Software & Systems Modeling, 18(1):7–10.

[Ciccozzi and Spalazzese, 2016] Ciccozzi, F. and Spalazzese, R. (2016). MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In Int. Symp. on Intelligent and Distributed Computing, pages 67–76. Springer.

[3] A demo is available: https://www.irit.fr/~Sylvie.Trouilhet/demo/iceDemo2020.mp4

[Combemale et al., 2008] Combemale, B., Crégut, X., Giacometti, J.-P., Michel, P., and Pantel, M. (2008). Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In 4^{th} Euro. Congress on Embedded Real-Time Software (ERTS). SIA.

[Combemale et al., 2016] Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., and Vojtisek, D. (2016). Engineering modeling languages: Turning domain knowledge into tools. Chapman & Hall.

[Coutaz, 2010] Coutaz, J. (2010). User Interface Plasticity: Model Driven Engineering to the Limit! In Int. Conf. on Engineering Interactive Computing Systems (EICS), pages 1–8. ACM.

[Coutaz and Crowley, 2016] Coutaz, J. and Crowley, J. (2016). A First-Person Experience with End-User Development for Smart Homes. IEEE Pervasive Computing, 15:26 – 39.

[Delcourt et al., 2021] Delcourt, K., Adreit, F., Arcangeli, J.-P., Hacid, K., Trouilhet, S., and Younes, W. (2021). Automatic and Intelligent Composition of Pervasive Applications. In 19^{th} IEEE Int. Conf. on Pervasive Computing and Communications (PerCom). IEEE Computer Society.

[Eclipse Foundation, Inc., 2013] Eclipse Foundation, Inc. (2013). EcoreTools - Graphical Modeling for Ecore. https://www.eclipse.org/ecoretools/. Accessed: 2021-04-16.

[Eclipse Foundation, Inc., 2020] Eclipse Foundation, Inc. (2020). Acceleo. https://www.eclipse.org/acceleo/. Accessed: 2021-04-16.

[Eclipse Foundation, Inc., 2021a] Eclipse Foundation, Inc. (2021a). Eclipse Modeling Framework (EMF). https://www.eclipse.org/modeling/emf/. Accessed: 2021-04-16.

[Eclipse Foundation, Inc., 2021b] Eclipse Foundation, Inc. (2021b). Sirius. https://www.eclipse.org/sirius/overview.html. Accessed: 2021-04-16.

[Evers et al., 2014] Evers, C., Kniewel, R., Geihs, K., and Schmidt, L. (2014). The user in the loop: Enabling user participation for self-adaptive applications. Future Generation Computer Systems, 34:110–123.

[Fanjiang et al., 2017] Fanjiang, Y., Syu, Y., Ma, S., and Kuo, J. (2017). An overview and classification of service description approaches in automated service composition research. IEEE Transactions on Services Computing, 10(2):176–189.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Pearson Education.

[Ghiani et al., 2017] Ghiani, G., Manca, M., Paternò, F., and Santoro, C. (2017). Personalization of Context-Dependent Applications Through Trigger-Action Rules. ACM Trans. on Computer-Human Interaction, 24(2).

[Gil et al., 2016] Gil, M., Pelechano, V., Fons, J., and Albert, M. (2016). Designing the Human in the Loop of Self-Adaptive Systems. In 10^{th} Int. Conf. on Ubiquitous Computing and Ambient Intelligence, pages 437–449. Springer.

[Hussein et al., 2017] Hussein, M., Li, S., and Radermacher, A. (2017). Model-Driven Development of Adaptive IoT Systems. In MODELS 2017 Sat. Event, volume 2019 of CEUR Workshop Proc., pages 17–23.

[Janisch, 2010] Janisch, S. (2010). Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication. PhD thesis, LMU Munich.

[Karami et al., 2016] Karami, A. B., Fleury, A., Boonaert, J., and Lecoeuche, S. (2016). User in the Loop: Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions. Information, 7(2).

[Klusch, 2008] Klusch, M. (2008). Semantic Web Service Description. In CASCOM: Intelligent Service Coordination in the Semantic Web, pages 31–57. Birkhäuser Basel.

[Koussaifi et al., 2019] Koussaifi, M., Trouilhet, S., Arcangeli, J.-P., and Bruel, J.-M. (2019). Automated user-oriented description of emerging composite ambient applications. In 31^{st} Int. Conf. on Software Engineering and Knowledge Engineering (SEKE), pages 473–478.

[Ludewig, 2003] Ludewig, J. (2003). Models in software engineering – an introduction. Software and Systems Modeling, 2(1):5–14.

[Mitchell, 1997] Mitchell, T. (1997). Machine Learning. McGraw-Hill.

[Oracle, 1995] Oracle (1995). Java Remote Method Invocation (RMI). https://docs.oracle.com/javase/tutorial/rmi/index.html. Accessed: 2019-01-31.

[Paternò, 2013] Paternò, F. (2013). End User Development: Survey of an Emerging Field for Empowering People. ISRN Software Engineering, 2013.

[PlantUML, 2021] PlantUML (2021). PlantUML in a nutshell. https://plantuml.com/en/. Accessed: 2021-04-16.

[Pérez et al., 2011] Pérez, F., Valderas, P., and Fons, J. (2011). Allowing end-users to participate within model-driven development approaches. In 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pages 187–190.

[Sheng et al., 2014] Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. (2014). Web services composition: A decade's overview. Information Sciences, 280:218 – 238.

[Sommerville, 2016] Sommerville, I. (2016). Component-based software engineering. In Software Engineering, chapter 16, pages 464–489. Pearson Education, 10^{th} edition.

[Sutton and Barto, 2018] Sutton, R. and Barto, A. (2018). Reinforcement Learning: An Introduction. MIT Press.

[The GEMOC Initiative, 2021] The GEMOC Initiative (2021). The GEMOC Initiative. http://gemoc.org/. Accessed: 2021-04-16.

[Tigli et al., 2009] Tigli, J.-Y., Lavirotte, S., Rey, G., Hourdin, V., and Riveill, M. (2009). Lightweight Service Oriented Architecture for Pervasive Computing. Int. J. of Computer Sci. Issues, 4(1).

[Triboulot et al., 2015] Triboulot, C., Trouilhet, S., Arcangeli, J.-P., and Robert, F. (2015). Opportunistic software composition: benefits and requirements. In Int. Conf. on Software Engineering and Applications (ICSOFT-EA), pages 426–431. INSTICC.

[Vogel, 2018] Vogel, T. (2018). Model-Driven Engineering of Self-Adaptive Software. PhD thesis, University of Potsdam, Germany.

[W3C, 2007] W3C (2007). Web Services Description Language (WSDL) Version 2.0. https://www.w3.org/TR/wsdl/. Accessed: 2020-04-09.

[Xiao et al., 2011] Xiao, H., Zou, Y., Tang, R., Ng, J., and Nigul, L. (2011). Ontology-driven service composition for end-users. Service Oriented Computing and Applications, 5(3):159.

[Younes et al., 2020] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2020). Agent-mediated application emergence through reinforcement learning from user feedback. In 29^{th} IEEE Int. Conf. on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). IEEE Computer Society.