J. Criado, L. Iribarne, N. Padilla.(2021)Heuristics-based mediation for building smart architectures at run-time.
Computer Standards & Interfaces. Elsevier, Volume 75, April 2021, 103501.ISSN: 0920-5489

Heuristics-based mediation for building smart architectures at run-time

Javier Criado [*], Luis Iribarne, Nicolás Padilla

Applied Computing Group, University of Almería, Spain

A R T I C L E I N F O

Keywords:
Dynamic software architectures
Heuristics
Trading
A* search
Model-based engineering

A B S T R A C T

Smart architectures are increasingly being used in current software development. Smart user interfaces, smart homes, or smart buildings are becoming common examples in the new era of smart cities. Software architectures usually related to these domains need to be adapted and reconfigured at run-time, for example, to provide new services, react to user interaction, or due to changes decided from the business logic of the application. Component-based techniques are a suitable way to carry out this kind of adaptation, as dynamic reconfiguration operations can be applied to the architecture. In this paper, we address run-time generation of component-based applications, taking the abstract definitions of their architecture as a reference, in addition to a set of available components. The process calculates the best configuration of components from the abstract definition by applying a trading approach based on an adapted A* algorithm. This algorithm uses heuristics based on syntactic and semantic information obtained from the component definitions. A case study related to mashup user interfaces formed by coarse-grained components is also explained. In short, the results show the usefulness of heuristics and suitable execution times for building the best configurations.

1. Introduction

Smart Cities is a trend-setting research domain that is changing software design, implementation and use. There are many initiatives in related topics, such as service management, data access, application run-time, configuration tools, and so forth, but many open issues still present opportunities for research [1]. One of them is related to their dynamics, since these types of systems tend to change, evolve, adapt to users, be scaled up, etc., and therefore requires flexible management of their structure and behavior [2].

Component-Based Software Engineering (CBSE) has long been considered an appropriate option for dealing with smart environments, for example, by applying component models to integrate heterogeneous applications, or using component interface standards to describe each piece of a system. Today, this technology has regained strength [3] for solving certain problems that arise in emerging smart environments such as mashups [4,5], smartphones [6], smart homes [7,8], smart buildings [9], or smart cities [2]. Furthermore, component-based development is being applied in certain recent solutions related to the Internet of Things (IoT) [10,11] and the Web of Things (WoT) [12,13].

CBSE provides mechanisms for constructing applications by joining and connecting pieces. Some component-based software systems need to be able to dynamically manage elements of the applications, for example, to change structure, adapt behavior or modify functionality. In such cases, the components are used for constructing software applications at run-time, as well as during design and construction. Components are used this way when applications are generated and adapted at run-time. As such, the most appropriate elements are selected from a set of available components at the time of adaptation. These mechanisms are used in software architectures that can be reconfigured by changing their structure and behavior, which are known in the literature as architectures of smart environments or **smart architectures** [1,2,14] regardless of whether the scope of application is smart cities, IoT systems or other such domains that require adaptation features.

The selection of components involved in this type of adaptation requires accessible repositories which can be inspected and queried to calculate the best possible configuration. Furthermore, these repositories might not remain static over time, since their components may be modified, either because existing components are deleted or due to the insertion of new resources (*i.e.*, new components). Therefore, the selection process does not always generate the same solution from the same input, as the result depends on the components existing in those repositories (among other factors).

Component repositories can be stored locally (managed internally by an organization), or if intended for public use can be shared by different organizations. Management of the repositories (queries, insertions,

* Corresponding author.
E-mail addresses: javi.criado@ual.es (J. Criado), luis.iribarne@ual.es (L. Iribarne), npadilla@ual.es (N. Padilla).

https://doi.org/10.1016/j.csi.2020.103501

deletions, etc.) may therefore come from different sources. This is the usual scenario in systems with applications constructed using components developed by third parties, such as Commercial Off-The-Shelf (COTS) components [15]. These repositories make up the existing component market that software is built from.

These component repositories can be managed like a service directory, which can be accessed by certain entities for offering services, and by other entities to make use of the available services. **Trading** techniques [16] are useful for facilitating execution of service export and import operations. Traditional trading approaches enable *services* to be discovered from *service type* definitions to solve an input specification, for example, by evaluating similarity [17]. Thus, trading mechanisms can be used to build component-based architectures from a reference architectural definition [18]. However, traditional mechanisms only focus on syntactic comparison of functional interfaces, and do not perform a **semantic** analysis of the interfaces (or other component parts).

Modern trading and **service discovery** [19] approaches are not limited to solving the functional requirements and capabilities of a target architecture, because in some cases, functional and non-functional information must be combined to calculate the best component configuration [20,21]. Moreover, from an industrial point of view, current and mature technologies related to distributed systems (such as Eureka, confd, etcd, Consul or Zookeeper [22]) support the implementation of service discovery operations that can be enriched with semantic information and improved with custom solutions [23,24]. As a result of the above, a trading approach to component management should take both component definition syntactics and semantics into account.

CBSE has been integrated into another kind of approaches apart from service-oriented applications [25], such as Mode-Based Engineering (MBE) [26]. This paradigm facilitates and supports the design, construction, deployment and reuse of software architectures, even more so in heterogeneous systems [27]. The use of Domain Specific Languages (DSLs) and other modeling techniques can be applied to formally describe the structure and behavior of the architectures or to generate the corresponding source code [28]. Furthermore, not only component descriptions and services can be achieved by using such models, but also the guidelines for designing different areas related to them, such as discovery or registration [23].

This paper presents STAS (Semantic Trading for smart Architectural Scenarios), a new version of a trading service based on semantic information for selecting components, evaluating their combination, and then constructing the software architectures with the best configurations. It therefore extends the traditional trading services applied in distributed processing by using CBSE to encapsulate the coarse-grained components that form part of the architecture, while benefiting from MBE to carry out the formal definitions and abstractions of both components and architectures. Furthermore, it executes a search algorithm to calculate the best component configuration, thus checking how well an architectural definition and each combination of components considered a possible solution match. Thealgorithm uses heuristics based on syntactic and semantic information described in eachdefinition.

The following research questions are addressed:

- (RQ1) What representation is appropriate for describing the components and architectures managed in our approach?
- (RQ2) Can this approach be applied in different domains or scenarios of smart architectures?
- (RQ3) How can a traditional trading service be extended to calculate the best configurations of components from a reference target architecture?
- (RQ4) What is the most important syntactic and semantic information when searching for architectural configurations that must be included in a component definition and still be considered manageable?
- (RQ5) How can the performance of this search be addressed to get suitable results at run-time?

The STAS trading service proposed is part of a methodology for adapting software architectures at run-time [5,29], and there is background research related to the definitions of components and architectures used as part of the fundamentals of our approach (RQ1). From these fundamentals, we can extract the prerequisites a scenario should have to be targeted by our methodology (RQ2). For RQ3, we studied whether a trading service could be adapted to build configurations of components at run-time. Then, we proposed new modules to generate these configurations by taking into account different conditions, such as the compliance of the architecture or the heuristics related to the components' properties. The new trading service and its modules are based on our previous research work [30]. To address RQ4, we developed an adapted A* version of the search algorithm used to calculate the configurations. This takes into account a set of properties related to syntactic and semantic information to score each possible solution. An A* algorithm ensures that the optimal solution is found, which takes the form of a simple path in a graph without evaluating the entire state space [31,32]. As such, our adapted algorithm creates a graph with nodes representing a combination of components during the calculation of the best possible configuration at run-time (RQ5). Research questions RQ4 and RQ5 are also addressed in a case study related to a specific scenario of smart architectures (mashup UIs) in which the best configurations are calculated from an input architecture and a set of available components.

The remainder of this article is organized as follows. Section 2 presents the context and fundamentals of our approach. Section 3 describes the semantic trading process developed for generating architectures at run-time. Section 4 explains the heuristics-based generation of configurations. Next, Section 5 gives the most relevant aspects about the implementation of the proposed semantic trader. Section 6 evaluates our approach in a component-based user interface scenario. Section 7 discusses the contributions with regard to the research questions. We review the related work in Section 8 and, finally, Section 9 outlines the conclusions and summarizes future work.

2. Real industrial context of smart architectural scenarios

Smart architectures are the structures (components, relations and properties) of a special type of component-based software that adapts its behavior depending on changes in the context. As mentioned above, reconfiguration of architectures enables this adaptation since component replacement and reconnection involve changes in the resulting software. However, component management and the calculation of new architectural configurations are not such simple tasks. Trading services can be used to manage component publication and search, thus allowing multiple implementations of real components related to the same component type. This enables to have multiple possible architectural solutions starting from a common initial architecture and from a change produced in the context information.

The semantic trading process proposed in this article is part of a methodology for run-time adaptation of software architectures in two main steps (see Fig. 1). First, *transformation*, which adapts the abstract representation of the architectures that define the software structure in terms of coarse-grained components [5], *i.e.*, it adapts the models which contain the architectural definitions by inserting new components, deleting elements or modifying connections in the architecture. Second, *semantic trading* (STAS) resolves the architectural solutions with references to real components from the architectural definitions obtained in the previous phase. This second process is the focus of this article, but it requires some background knowledge for it to be properly understood.

The software architecture shown in Fig. 1 is a short example which takes a closer look at this methodology. Suppose a user interacting with

J. Criado, L. Iribarne, N. Padilla.(2021)Heuristics-based mediation for building smart architectures at run-time. Computer Standards & Interfaces. Elsevier, Volume 75, April 2021, 103501.ISSN: 0920-5489

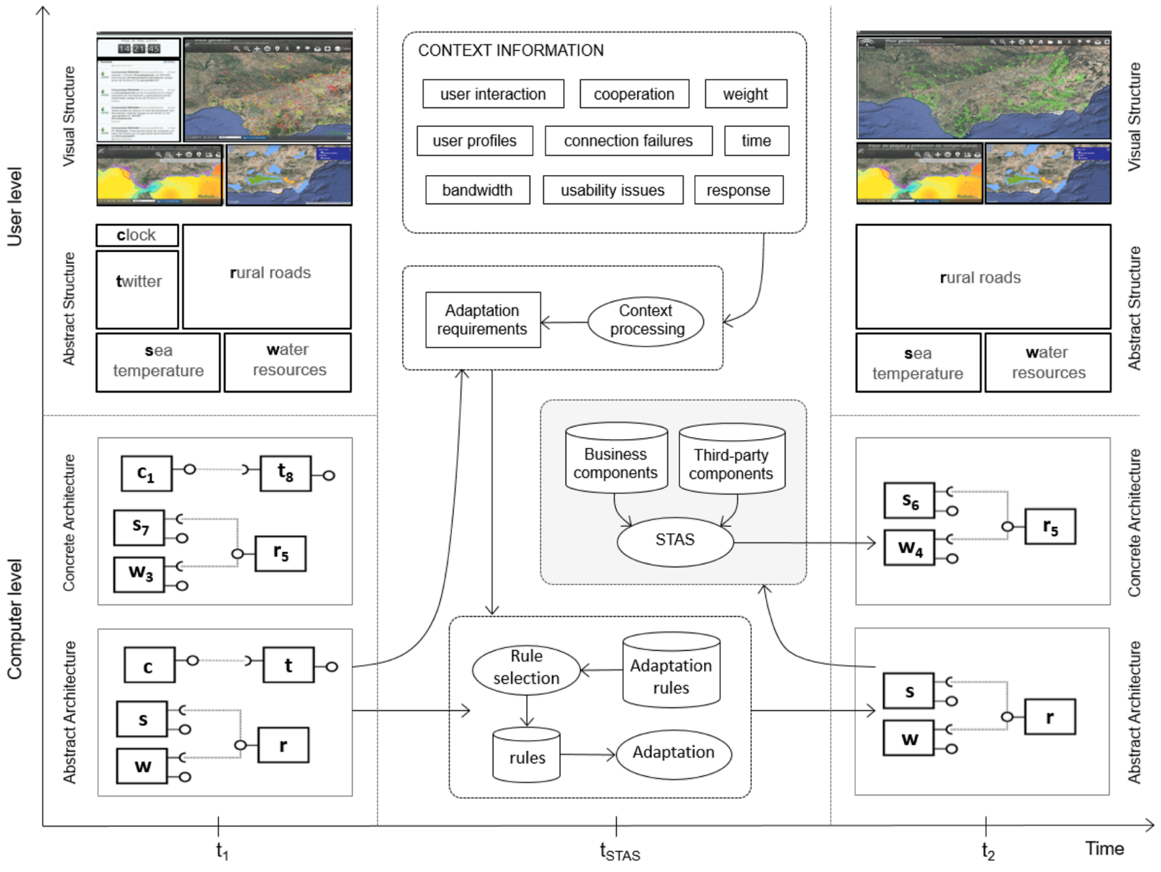


Fig. 1. Methodology for adapting software architectures at run-time.

an application composed of the following widgets: three maps, a clock and a social network. Then, due to a certain change in the context (*e.g.*, the user triggers an event to remove the information not related to the maps), the transformation process removes two components (clock and twitter). Next, the STAS selects the best configuration of real components that best resolves the abstract level architecture. As a result, c_1 and t_8 are removed, map s_6 replaces s_7, w_4 replaces w_3, and r_5 is not modified. Finally, the adapted web application is deployed from the architecture built. Although Fig. 1 shows an overall view of the methodology, this paper does not address transformation or deployment and visualization of final applications. Nevertheless, this section describes (1) the principles that must be taken into account for our methodology to be applied, (2) the models managed by the whole approach, and (3) transformation, since it is the step before semantic trading.

2.1. Prerequisites of the methodology

The methodology developed for adapting component-based software systems is intended to be applied to different domains. Accordingly, the following conditions must be taken into account for constructing architectures that can be targeted by our proposal:

(a) Components must be described by a **specification** which includes **functional** and **extra-functional** information. Both parts are used in calculating the architectural configurations.
(b) Component specifications must be stored in **repositories** and retrieved at run-time. Third-party developers can use these repositories to publish new components, as in COTS approaches. Consequently, architectural solutions depend on the architectural definitions and components available in the repositories.
(c) Platforms targeted for the deployment of the architectures must allow **dynamic changes**. The architectures must therefore be precompiled or interpreted so that their parts can be reconfigured at run-time.

Some of the possible application domains of this approach are smart home environments [33], smart TV applications [34], smart cities [35], communication networks [36], fine-grained user interfaces [37], or mashup user interfaces [38]. Even though the methodology was developed following a generalist approach, it is now being tested in the domains of smart home applications [39] and mashup UIs [5].

In the domain of smart homes, an installation made up of different devices can be developed as an architecture of components that communicate with each other. Fig. 2 shows an example of a smart home application we developed [39], which includes a TV, speakers, and a window with magnetic sensors. In addition, a smart watch measures the heart rate of one of the residents. The right-hand side of Fig. 2 shows the architecture of the application at a given moment. In this configuration, one software component shows the user's heart rate on the smart watch. The TV is managed by an isolated component and an actuator receives the signal from the sensor, sounding an alarm on the speakers that the window is being opened. This is only an example of the possible configurations, since the TV or the smart watch could also be connected to the actuator to show alarm events, and so on.

A mashup UI is a particular type of Graphical User Interface (GUI) operating in a web environment which is built by assembling coarse-grained components. Fig. 3 shows an example of a mashup UI from our previous research work as described in [5]. This client application is part of a Geographic Information System (GIS) and offers maps, histograms, pie charts, legends for information layers and social-network components, among others. The system was developed as part of the

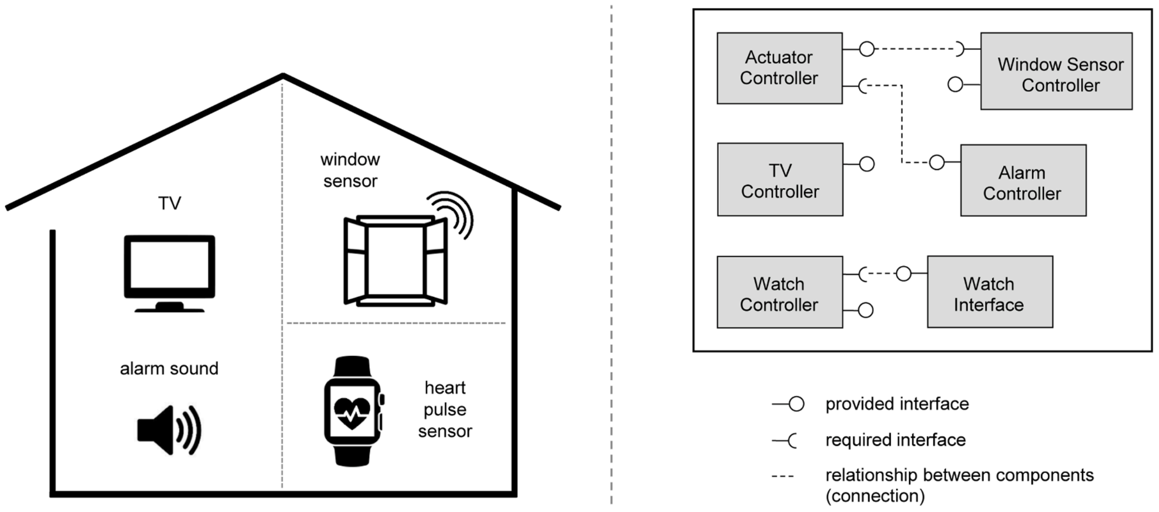
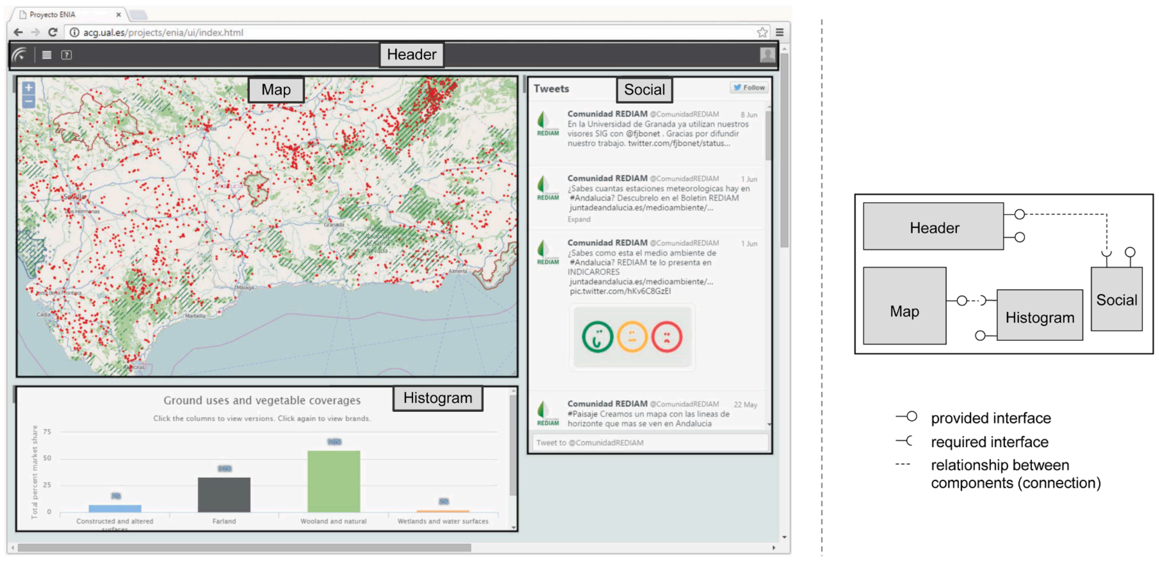


Fig. 2. Example of a smart home application [39].


Proyecto ENIA
acg.ual.es/projects/enia/ui/index.html
Header
Map
Social
Histogram
Ground uses and vegetable coverages
Click the columns to view versions. Click again to view brands.
Constructed and altered surfaces
Farland
Wooland and natural
Wetlands and water surfaces
Total percent market share
Header
Map
Histogram
Social
provided interface
required interface
relationship between components (connection)

Tweets

Comunidad REDIAM @ComunidadREDIAM 8 Jun
En la Universidad de Granada ya utilizan nuestros visores SIG con @fjbonet . Gracias por difundir nuestro trabajo. twitter.com/fjbonet/status...

Comunidad REDIAM @ComunidadREDIAM 1 Jun
¿Sabes cuantas estaciones meteorologicas hay en #Andalucia? Descubrelo en el Boletin REDIAM juntadeandalucia.es/medioambiente/...
Expand

Comunidad REDIAM @ComunidadREDIAM 1 Jun
¿Sabes como esta el medio ambiente de #Andalucia? REDIAM te lo presenta en INDICARORES juntadeandalucia.es/medioambiente/... pic.twitter.com/hKv6C8GzEl

Comunidad REDIAM @ComunidadREDIAM 22 May
#Paisaje Creamos un mapa con las lineas de horizonte que mas se ven en Andalucia

Tweet to @ComunidadREDIAM

Fig. 3. Example of a mashup user interface [5].

ENIA (Environmental Information Agent) research project and the corresponding user interface is available online[1].

One of the main reasons for choosing this application domain is the presence of component dependencies. For example, the *Histogram* and *Map* components in Fig. 3 are dependent on each other. The first component shows some information related to the second one, and as a result, the histogram cannot operate properly without the map. Similarly, the *Social* component is connected to the Header component because it depends on it for information about the user. Components may also be isolated and have no relationship with any other component.

2.2. Models involved

From the MBE perspective, our semantic trading process uses four types of models to define abstract architectures, concrete architectures, abstract components and concrete components. *Abstract* and *concrete* concepts are related to the Cameleon reference framework [40]. In this framework, UIs have four possible representation levels: task and concept, abstract, concrete and final. The *task and concept* level is related to the Computation Independent Model (CIM) of the Model-Driven Architecture (MDA), the *abstract* level corresponds to the Platform Independent Model (PIM) and the *concrete* level correlates with the Platform Specific Model (PSM). The *final* level represents the real software which is executed or interpreted.

We developed our methodology based on the levels mentioned above, but extending it to any software architecture complying with the prerequisites. Moreover, the concrete architectures are not built by executing a model-to-model or a model-to-text transformation, but by STAS actions and algorithms. The abstract level therefore identifies what should be present in the architecture, whereas the concrete level defines real objects forming part of an architectural solution (for an in-depth description of these levels see [5] and [29]). Both types of architectures (abstract and concrete) are described using the Domain-Specific Language (DSL) shown in the metamodel in Fig. 4. Metamodeling or the construction of a UML profile are the two classical approaches for defining a domain-specific modeling language.

Our approach uses a new language instead of a UML profile with annotations for non-functional properties. This is because we pursued

[1] ENIA Mashup UI – http://acg.ual.es/enia/ui

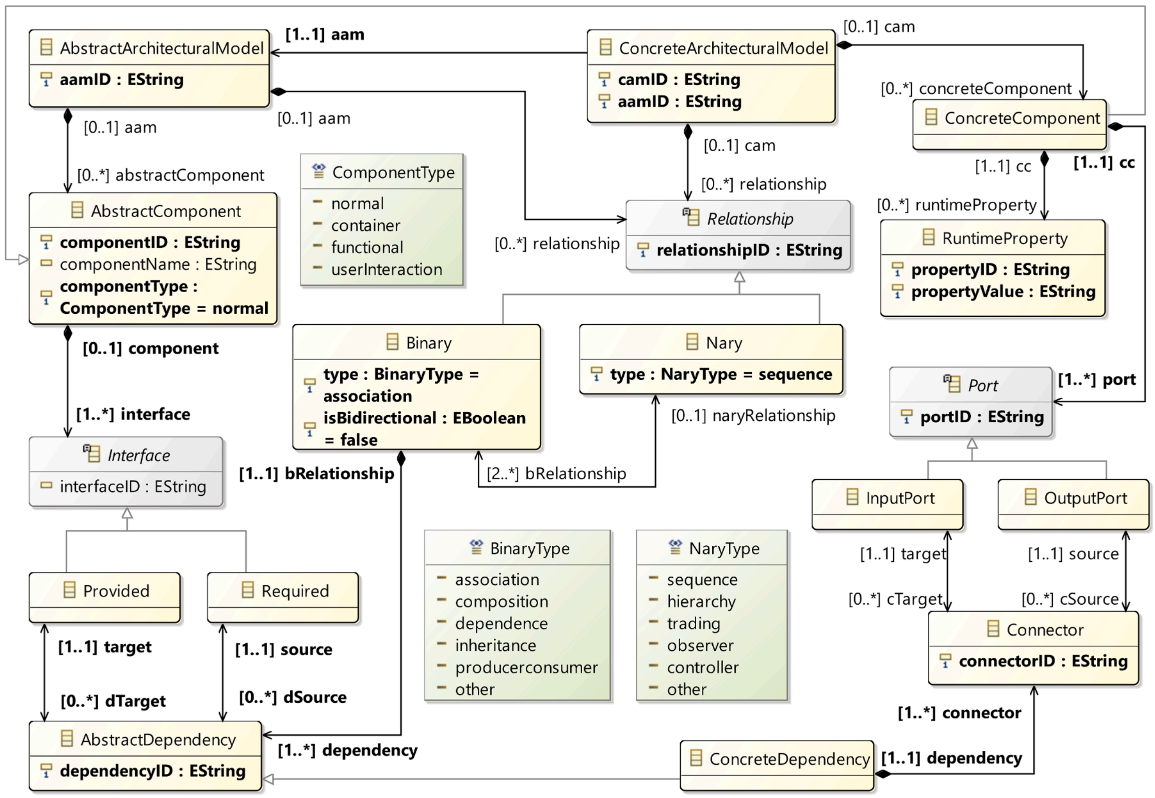
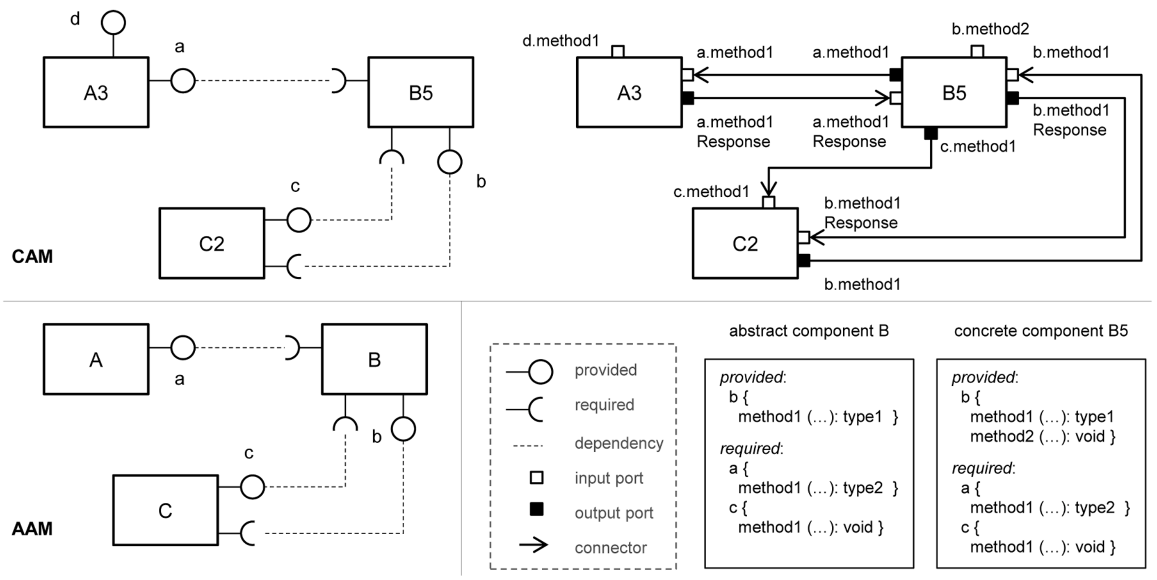


Fig. 4. DSL for describing architectures in abstract and concrete levels.

the definition of a closed abstract syntax with a specific set of classes, properties and relations between them [41,42]. Other Architecture Description Languages (ADLs) could be also used in our approach (such as AADL, ACME, Darwin, etc.), but due to our application of MBE techniques for adaptation and reconfiguration purposes, we required an MOF-based description language and we chose to build a new language instead of adapting an existing one to our requirements [43,44].

Abstract Architectural Models (AAM) are used for describing the components present in the architectures and how they relate to each other through the dependencies between functional interfaces. Provided interfaces define the services offered by a component, whereas required interfaces represent the services that a component needs (*i.e.*, functionality not belonging to the component and which must be accessed) to operate properly. Concrete Architectural Models (CAM) include the above information (derived from the inheritance relationship), but also describe the relationships between components in terms of ports and connectors.

Ports and connectors are defined with regard to communication between components. The mechanism for invoking the operations described in a provided interface is as follows. Each operation is accessible through one input port. Operations which return information are also defined with an output port. In required interfaces, the



```
provided:
 b {
  method1 (…): type1  }
required:
 a {
  method1 (…): type2  }
 c {
  method1 (…): void }
```

```
provided:
 b {
  method1 (…): type1
  method2 (…): void }
required:
 a {
  method1 (…): type2  }
 c {
  method1 (…): void }
```

Fig. 5. Differences between an abstract architectural model (AAM) and a concrete architectural model (CAM).

representation of the operations is the opposite. Required interfaces have an output port to invoke the corresponding operation in a component's provided interface. In addition, operations which return data are also defined with an input port. Fig. 5 shows an example of an abstract architectural model and a feasible concrete architectural model related to the abstract definition. The correlation (and the differences) between the two models can therefore be seen.

The functionality of component *B*, defined in the AAM, is resolved by component *B5* in the CAM. More specifically, the provided functional interface labeled *b* in the AAM corresponds to functional interface *b* in the CAM. Nevertheless, the provided interface in the CAM includes one more operation than the abstract definition. From the perspective of ports and connectors, the operation labeled *method1* involves the presence of corresponding input and output ports, since the operation returns information. The operation labeled *method2* is defined using only one input port because no data is returned.

The metamodel shown in Fig. 6 specifies the DSL used to describe the components contained in abstract and concrete architectural models. On one hand, an *abstract component* specification defines the set of features that a software component must include (*i.e.*, similar to a component type). On the other, a *concrete component* specification describes the characteristics of a real software component (*i.e.*, already implemented). As described above, abstract and concrete component concepts are matched to the 'service type' and 'service' concepts, used in distributed processing. The proposed metamodel was inspired by existing models for describing web services [45] and COTS [18].

In our approach, both abstract and concrete component definitions are divided into four parts: functional, extra-functional, packaging and marketing. The **functional** part describes functional features of provided and required interfaces. Each functional interface is defined using the Web Services Description Language (WSDL) standard syntax [45]. This language was chosen because it enables correspondence between the *portType* concept in WSDL 1.1 (*interface* in WSDL 2.0) and the functional interfaces of the abstract and concrete components to be established. Furthermore, operations, as well as input and output types, can be defined in the syntax of this language. Thus, each interface is composed of a set of operations, and each operation consists of an input, and optionally, an output. Both the input and output can be made up of a set of elements, which are described by a name, a type and also multiplicity. With this WSDL-based structure, interfaces can be defined in this language format. In such cases, a translation from this code to the corresponding model fragment is necessary by extraction or text-to-model transformation [46].

The **extra-functional** part of the specification describes the properties which are not defined in the functional part. That means this block contains any property other than the operations provided or required, including non-functional attributes and quality of service (QoS) features. Each property defines an attribute and the corresponding value an abstract component must have, or a concrete component implements. Component dependencies are also specified in this block. Each dependency determines which of the required interfaces are mandatory and must be resolved in an architecture to work properly. In our approach, required interfaces may or may not be mandatory. A non-mandatory required interface may remain unsolved, and the component will work. As such, required interfaces not included as dependencies participate in terms of complementary component

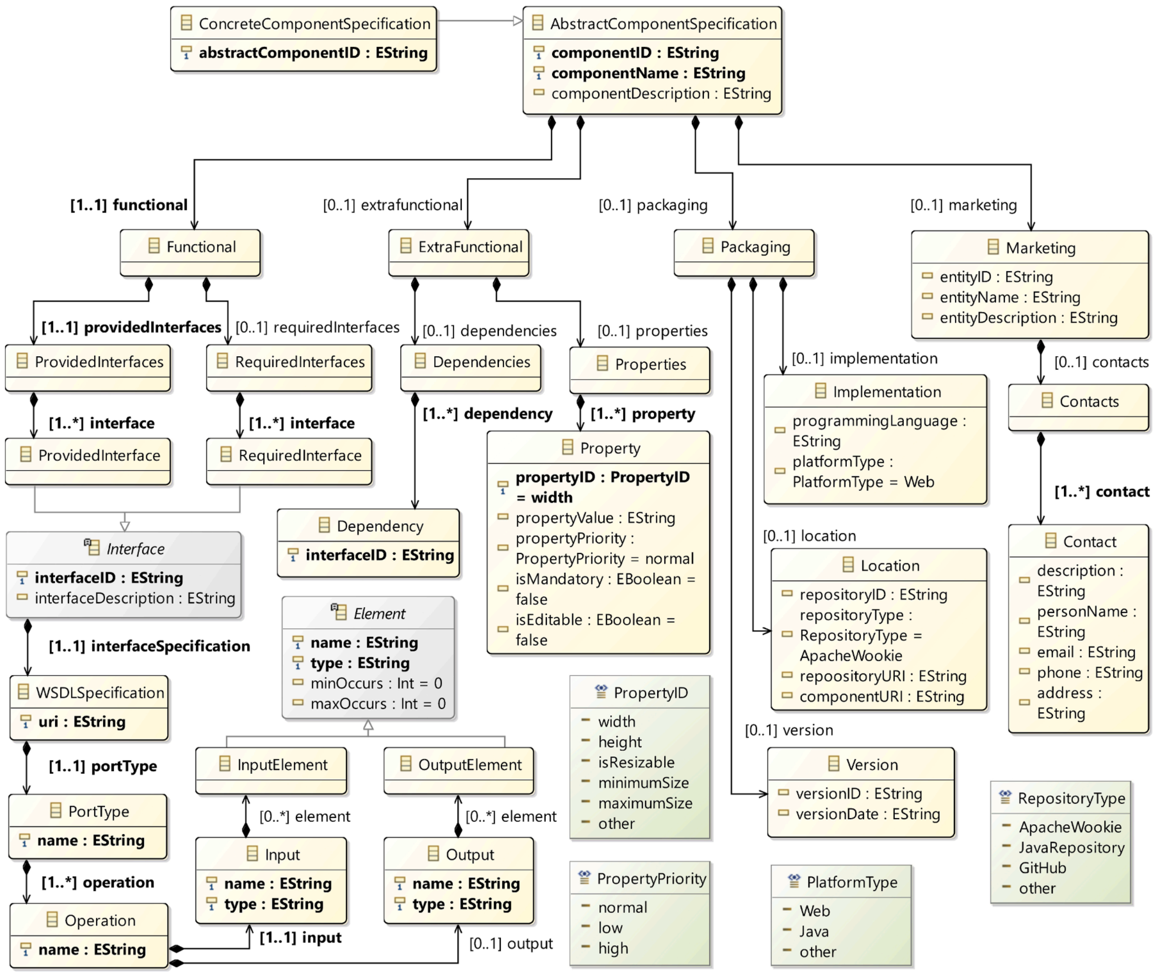


Fig. 6. DSL for describing components on abstract and concrete levels..

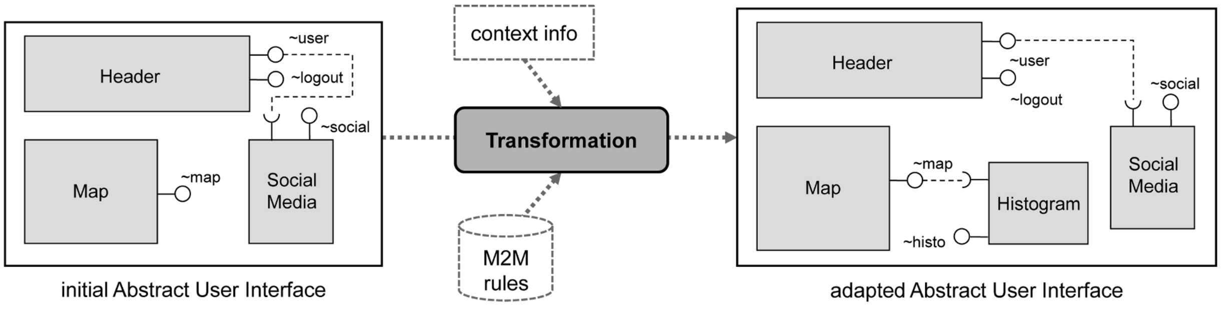


Fig. 7. Transformation example.

functionality.

In addition to the functional and extra-functional information, **packaging** and **marketing** data are valuable for characterizing components. Although such information is non-functional and could be included in the extra-functional block, it is described in two separate parts. Packaging defines certain information about the implementation and location of a component. Regarding implementation, a component can describe the programming language and the target platform type, and with regard to location, a specification can define information about the repository which stores the component. In addition, information about the URI and version can also be defined. The latter is only valid for concrete components, since an abstract component cannot specify this kind of data. Marketing includes information about the entity developing the component and organization contact information. An example of a concrete component instance is available on the project website[2].

There are some differences in the multiplicity and mandatory nature of some parts and attributes regarding the specifications of these two types of components. For example, the instantiation of the packaging and marketing blocks is mandatory in the case of concrete components, whereas they are optional in abstract components. These kinds of syntactic constraints are checked and validated using the Object Constraint Language (OCL) [47].

2.3. *Transformation step*

As mentioned, a transformation step related to the abstract level of the architectures, *i.e.*, the AAM, is necessary before the STAS can be applied. This process consists of a model-to-model (M2M) transformation [46] to refactor an AAM at a specific point in time depending on three inputs: (a) the initial model, (b) the context information, and (c) a repository of M2M transformation rules. Fig. 7 shows an example of this transformation in an initial UI containing a map, a social media widget and a header component. Due to a change in the context (*e.g.*, user interaction, proactive configuration, a system decision determined by the business logic, etc.), transformation adds a new component, namely a histogram showing geospatial information related to the map.

The M2M transformation is the first stage performed by the methodology. This is a flexible and smart process, and therefore, the model transformation in each adaptation step is not fixed or pre-set. Each transformation is dynamically built up from a repository of rules, thus providing a powerful mechanism for modifying and improving the adaptation logic. More details about the transformation stage are described in [5,48] and additional resources related to the fundamentals of our approach are available on the project website[3]. Summarizing, the transformation process is as follows. A set of adaptation rules (see M2M rules in Fig. 7) is used to change the structure of the component architecture and their connections. Depending on the context information and the initial software architecture, this process selects a subset of transformation rules in the ATL language [49] and a new M2M transformation is built for adapting the architecture (*i.e.*, the architectural model). The transformation provides an adapted architecture, but its representation is still on an abstract level. The second stage of the methodology regenerates these abstract architectures. The best configurations of concrete components for this are calculated in the STAS trading process.

3. Semantic trading at run-time

This section explains our semantic trading approach (the second step in the background methodology). First, the concept of trading for managing architectures is introduced. Following this, the use of the semantic information for trading is defined. Finally, the search algorithm for building component configurations at run-time is described.

3.1. *Trading in software architectures*

Mediation, or trading, is a well-known mechanism for searching repositories of services and locating the most appropriate operations for a specific input contract [50,51]. There are some proposals in the literature addressing the use of such mechanisms for managing software components (in particular, third-party or COTS components) and assisting in the construction of software architectures at design time [18,52,53]. However, it must be possible for the trading process to manage the architectures in our methodology to be executed at run-time. This is because component configurations must be calculated dynamically when changes occur in the execution context. Accordingly, the proposal should deal with some time-related aspects. The main features of this process are described below.

Proper handling of the specifications describing each of the elements involved in the process is an essential part of component and architecture management. As such, the object responsible for mediation (*i.e.*, the *trader*) should be able to: (a) find components based on input parameters, (b) add/modify/delete component specifications, and (c) provide a mechanism for configuring the execution policies. These three main sets of features are related to the implementation of the functionality specified by the (a) *Lookup*, (b) *Register* and (c) *Admin* interfaces in the reference model of distributed processing [16].

A trader implementing the three interfaces above is known as a *standalone* trading service. Our approach uses a trader of this type, but it extends the functionality of traditional approaches. The purpose of existing trading functions is to manage repositories of services and service types with regard to export and import operations. Consequently, producer objects publish their services in the trader and consumer objects query the trader to obtain information about available services. Similarly, our methodology uses a trading service to manage components and component types. Component types correspond to abstract components, whereas components are equivalent to concrete ones.

The proposed trading service provides a variety of features for

[2] Examples of components – http://acg.ual.es/projects/cosmart/stas/models/

[3] STAS approach (CoSmart project) – http://acg.ual.es/projects/cosmart/stas/

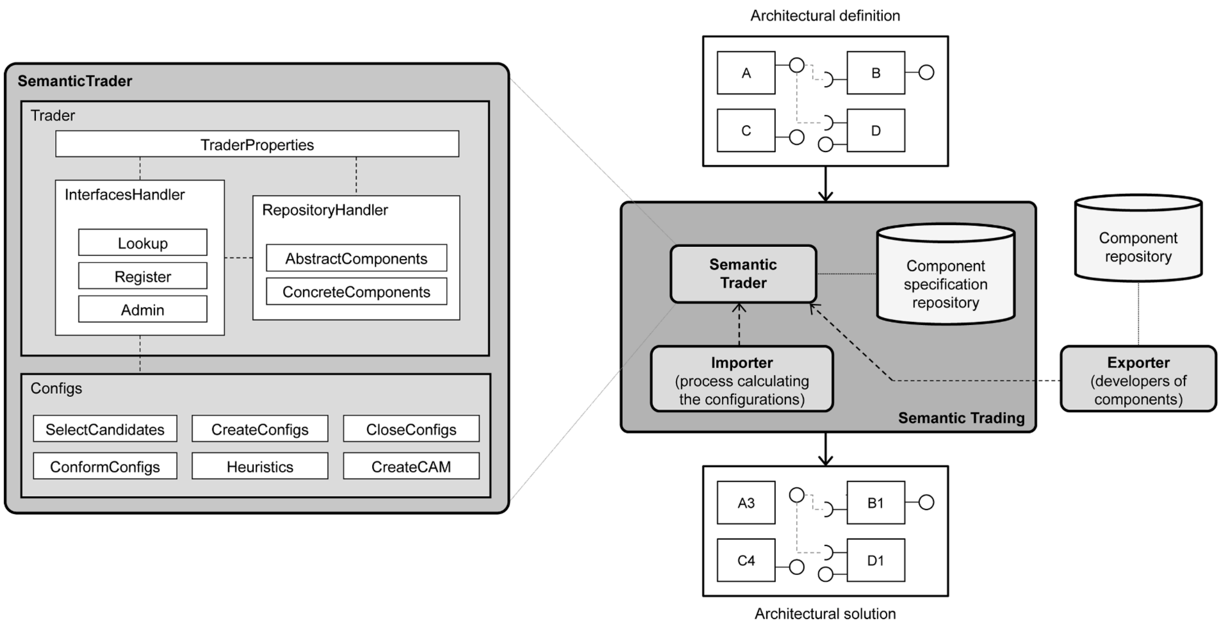


Fig. 8. Proposed trading service and its use in regenerating architectures.

handling the abstract and concrete component repositories (*RepositoryHandler*). In addition, it implements the functionality needed to manage the three interfaces of the standalone service (*InterfaceHandler*). Moreover, the service proposed extends this functionality by incorporating a module called *Configs*, which has the following features. First, candidates are selected by semantic matching of functional interfaces. Second, configurations are generated using a heuristic to find the best combination of components based on established criteria. Thus, a *Semantic Trader* for regenerating concrete architectures at run-time is obtained. The left hand side of Fig. 8 shows the proposed trader, which has the following functionalities:

(i) Obtain the set of component specifications meeting certain DSL values (*Lookup*).
(ii) Manage component specifications with create, update and delete operations (*Register*).
(iii) Calculate configurations of concrete architectures from abstract architectures at run-time (*Configs*).
(iv) Configure the number of component specifications returned by the operations of the Lookup module (*Admin*).
(v) Define the maximum time permissible for building the concrete architecture that is chosen as the best solution for an abstract definition (*Admin*).
(vi) Modify the configuration of matching performed when constructing the concrete architectures (*Admin*).

Once the trader object is designated, the objects with *exporter* and *importer* roles must be identified (see the right side of Fig. 8). Exporters are objects adding, modifying or removing specifications of components in the system. As a result, this role is played by any organization using the methodology, because it needs to manage the components in the repositories. In addition, third-party organizations involved in component development also act as exporters. However, objects that need to query and retrieve the components available in the repositories operate as importers. In this methodology, semantic trading (also called regeneration) itself operates as an importer. In particular, operations in the *Configs* module are importer objects, since they obtain the information needed to generate the component configurations and build the architectural solutions.

3.2. Semantic trading

As mentioned above, component functional interfaces (both abstract and concrete) are described in WSDL which, in general, is one of several possibilities for defining interfaces and, in particular, can be used as an Interface Description Language (IDL) to document COTS components, in particular. Thus, existing component models for describing COTS components make use of syntactic and semantic features to define their functional interfaces [55]. Syntactic information is therefore related to the way an interface is described, *i.e.*, the attributes and operations that comprise its definition. One possible solution for semantic information is to define each individual operating behavior.

This behavior is traditionally represented in formalities, such as algebraic equations, pre/post conditions and invariants, to provide detailed information about when an operation should be executed or what the state of the component is after its execution. Furthermore, such information is usually expressed in languages that can be analyzed automatically by software. However, the computation times necessary for management of potential interface matching together with incorrect combinations (mismatching) [56] suitable for the analysis of configurations at design time are inadmissible for building configurations and generating architectures at run-time.

Our semantic trading approach is based on the assumption that the possible types which can be used to describe operation inputs and outputs are limited. Accordingly, we created a *namespace* that groups all the possible types, identified as `trader:typeName`. These types are described using an XML schema , and referenced from the definition of interfaces, contained in a WSDL file or in the corresponding fragment of the model containing the specification of the component (see WSDLSpecification in Fig. 6). These types are equivalent to complex data types that provide information about: (a) the name, the type and the multiplicity of the elements in the complex data type, and (b) the operations using this data type and whether it is used as input or output.

A type definition can be illustrated with the following example. An operation called `loadLayer` is part of the `manageLayers` interface on the component map, as shown in Fig. 9 (chosen from the mashup UI in Fig. 3). This operation enables the map to load geospatial information so it can be displayed as a visual layer. As input, this operation requires a parameter with the location of the service providing the geospatial data, and another with the identifier of the layer to be selected from the service. A complex type, `trader:loadLayer`, composed of two

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions name="Example" targetNamespace="http://ws.cos.acg.ual.es/"
3    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
4    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5    xmlns:tns="http://ws.cos.acg.ual.es/"
6    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
7    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
8    <wsdl:types>
9      <xs:schema elementFormDefault="unqualified" targetNamespace="http://ws.cos.acg.ual.es/"
10       version="1.0" xmlns:tns="http://ws.cos.acg.ual.es/"
11       xmlns:trader="http://ws.cos.acg.ual.es/trader"
12       xmlns:xs="http://www.w3.org/2001/XMLSchema">
13     <xs:element name="loadLayer" type="trader:loadLayer"/>
14     <xs:element name="loadLayerResponse" type="trader:loadLayerResponse"/>
15     <xs:complexType name="loadLayer">
16       <xs:sequence>
17         <xs:element minOccurs="1" maxOccurs="1" name="layerID" type="xs:string"/>
18         <xs:element minOccurs="1" maxOccurs="1" name="serviceOGC" type="xs:anyURI"/>
19       </xs:sequence>
20     </xs:complexType>
21     <xs:complexType name="loadLayerResponse">
22       <xs:sequence>
23         <xs:element minOccurs="1" maxOccurs="1" name="return" type="xs:boolean"/>
24       </xs:sequence>
25     </xs:complexType>
26     </xs:schema>
27   </wsdl:types>
28   <wsdl:message name="loadLayer">
29     <wsdl:part element="tns:loadLayer" name="parameters"> </wsdl:part>
30   </wsdl:message>
31   <wsdl:message name="loadLayerResponse">
32     <wsdl:part element="tns:loadLayerResponse" name="parameters">   </wsdl:part>
33   </wsdl:message>
34   <wsdl:portType name="manageLayers">
35     <wsdl:operation name="loadLayer">
36       <wsdl:input message="tns:loadLayer" name="loadLayer"> </wsdl:input>
37       <wsdl:output message="tns:loadLayerResponse" name="loadLayerResponse"> </wsdl:output>
38     </wsdl:operation>
39   </wsdl:portType>
40 </wsdl:definitions>
```

Fig. 9. WSDL specification of an example interface.

elements is therefore defined: (1) `layerID`, which is an attribute of the *String* type; and (2) `serviceOGC`, which is an attribute of the *anyURI* type (see lines 15–20 in Fig. 9). As output, it returns a *boolean* value indicating whether the operation has been executed successfully. Accordingly, the complex type `trader:loadLayerResponse`, composed of this boolean element is also defined (lines 21–25).

Definition of new complex types makes it possible to identify all the operations that match a description, as shown in the example above. As such, interfaces can be matched by using the semantic information related to operation input and output. An example of two equivalent interfaces is shown in Fig. 10. The names of interfaces and operations do not match, but they can be paired by operation type. In both interfaces, there is a layer loading operation and another one for removing them, and their semantic description is the same as indicated by the input and output types.

The set of types defined conforms to a vocabulary of types used for managing synonyms during regeneration. This represents a limitation on the use of any type of data as input or output of the operations, since developers must know the types that exist before attempting to build the components. Nevertheless, this is a suitable solution for comparing components at run-time, taking semantic information about their functional interfaces into account. Note that this matching does not perform any calculations related to the operation execution order. This order is inherent in the business logic that implements each component, and the choreography of the operations is assumed to be carried out correctly. In addition to the description of interfaces, semantic information is present in the evaluation of components and architectures used in the search algorithm heuristics, as seenbelow.

3.3. Generating configurations

As mentioned above, one of the main functionalities of the semantic trader is construction of concrete architectures from abstract definitions. The *Configs* module calculates the best possible configurations of the concrete components available in the repository and verifies that they meet the requirements established in the abstract architecture. The module developed for generating the configurations includes an adapted A* algorithm and therefore limits the search space, always ensuring the best possible solution within the configurations analyzed. This module performs the following steps:

(a) **Select candidates**. This operation is used to select (from the repository of concrete components) candidates to be included in the architecture. It checks which concrete components have at least one provided interface in common (*i.e.*, that can be matched) with the set of provided interfaces in the abstract architecture. Interface matching is carried using the semantic operation information, so two operations are considered equivalent if they have the same input and output types. The required interfaces and the rest of the information in the component specifications are not taken into account in this selection.

(b) **Calculate configurations**. In this operation, the different configurations that could be a solution of the abstract architecture are calculated from all the possible combinations of the candidates. Since this is a highly complex operation which cannot be performed at run-time, some limitation mechanisms must be included in the algorithms in this step. A configuration is considered a possible solution if it contains all the provided interfaces in the abstract architecture, and may also include any additional interfaces.

(c) **Closure of configurations**. This operation is performed from the configurations considered *possible solutions*. The goal is to filter out incomplete solutions. A configuration is considered incomplete when at least one of the components requires a component in addition to those defined in the abstract architecture. In such cases, the configuration is *not closed*.

(d) **Compliance of configurations**. As in the case of closed configurations, this operation is applied to configurations that are considered *possible solutions*. This process discards those configurations which do not comply with the structure established by the abstract architecture, in other words, that have decompositions or group elements that differ from the abstract definition.

Interface A

```
interface manageLayers {
  trader:loadLayerResponse loadLayer(trader:loadLayer);
  void removeLayer(trader:removeLayer);
};
```

Interface B

```
interface layerOperations {
  trader:loadLayerResponse loadOGCService(trader:loadLayer);
  void delInfoLayer(trader:removeLayer);
};
```

Fig. 10. Equivalent interfaces from a semantic perspective.

(e) **Apply the heuristics.** This operation scores each component configuration. Depending on the purpose, this score can be calculated from a configuration of components which may or may not be a solution of the abstract architecture. The targeted configurations therefore include those combinations of components consisting of partial solutions. This step relies on the guarantee of being able to determine a set of suitable score metrics related to the different software architecture domains [54].

(f) **Build the concrete architecture.** Once the configuration of components that best matches the abstract architecture has been selected, the *Configs* module constructs the concrete architectural model. This model is used to deploy the final software application.

In the following definitions, components and architectures are simplified to emphasize the functional part. To this end, a formal definition of the main operators used in the proposal is provided below for Component, Closure, Replacement and Compliance.

Definition 1. (*Component*) A component C can be described as $C = (\mathscr{R}, \overline{\mathscr{R}})$, where $\mathscr{R}$ is a set of provided interfaces and $\overline{\mathscr{R}}$ is a set of required interfaces. Therefore, $C.\mathscr{R}$ represents the component's provided interfaces.

In a similar way, the expression $AAM.\mathscr{R}$ represents the provided interfaces of an abstract architecture, whereas $AAM[i].\mathscr{R}$ identifies the provided interfaces of the i-th component. The closure of configurations will discard these configurations having one or more required interfaces which are not provided by any other component.

Definition 2. (*Closure*) A configuration is closed if it fulfills the expression $\cup\ C_i.\overline{\mathscr{R}} \subseteq \cup\ C_i.\mathscr{R}$, *i.e.*, the union of all the required interfaces is included in the union of all the provided interfaces. It is important to note that the subset operator ($\mathscr{R}_1 \subseteq \mathscr{R}_2$) means that for all interfaces from $\mathscr{R}_1$ ($R_1^i \in \mathscr{R}_1$) exists an interface in $\mathscr{R}_2$ ($R_2^j \in \mathscr{R}_2$) and R_1^i can be replaced by R_2^j.

An interface can be replaced by another one if both offer the same services. In this sense, if an architecture is closed, all the required interfaces are provided by a certain component of it. The compliance of configurations will discard these configurations with a different structure with regard to the abstract definition.

Definition 3. (*Replacement*) A component $C_1 = (\mathscr{R}_1, \overline{\mathscr{R}}_1)$ can be replaced by another component $C_2 = (\mathscr{R}_2, \overline{\mathscr{R}}_2)$, denoted by $C_2 \leqslant C_1$, if $(C_1.\mathscr{R}_1 \subseteq C_2.\mathscr{R}_2) \wedge (C_2.\overline{\mathscr{R}}_2 \subseteq C_1.\overline{\mathscr{R}}_1)$.

Definition 4. (*Compliance*) Let's suppose an abstract architecture defined as $AAM = \{A_1, A_2, \ldots A_n\}$ and a configuration of a possible concrete architecture defined as $CAM = \{C_1, C_2, \ldots C_m\}$, we established that CAM is compliant with AAM if $\forall i \in \{1.m\}$, $\forall j \in \{1.n\} \bullet C_i.\mathscr{R} \cap A_j.\mathscr{R} \neq \varnothing \Rightarrow (C_i \leqslant A_j) \wedge (A_j \leqslant C_i)$.

This means that component C_2 provides all the services (it can in fact provide more) offered by component C_1, and it requires the same services (and if need be less services) from other components. The constraint of compliance has been established to obtain a configuration with components that conform to the abstract specifications as well as match the structure defined in the *AAM*.

To make it easier to understand closure and compliance, Fig. 11 shows an example of an abstract architecture and four possible solutions. The four configurations of concrete components are not all suitable for building the concrete architecture as a process output. The configurations that would result in CAM_1 and CAM_3 being acceptable, whereas CAM_2 and CAM_4 would not be. The CAM_2 *MapOther* component has a required interface in addition to the abstract definition, meaning this configuration is not closed. The *Header* component in the abstract architecture is resolved by two components of CAM_4 (*UserLogOut* and *UserInfo*) and causes structural mismatching that breaks with compliance. It should be mentioned that a required interface is mandatory if it is described as a dependency in the concrete component (see DSL for describing components in Fig. 6).

Regarding application of the heuristics, note that the adapted A* algorithm only scores the configurations that are calculated during the exploration of the search tree. Thus, the possible solutions are checked to analyze if they are closed and compliant.

4. Heuristics-based generation of configurations

This section is focused on the explanation of the search process that is performed during the construction of the configurations. First, the algorithm applied in the process is defined. Then, the metrics and calculations related to the heuristics used in this algorithm are described.

4.1. Adapted A* search algorithm

The algorithm implemented to generate the configurations of the *Configs* module is carried out by adapting the A* search algorithm [31]. In this type of algorithm, a graph represents the search space and its nodes identify the states to be advanced to in the search. The goal is to find the least-cost path to the target node from a starting node. This is calculated with an evaluation function $f(x) = g(x) + h'(x)$. Function $g(x)$ represents a known distance (pre-calculated) between the starting node and the current node. Furthermore, $h'(x)$ identifies the estimated value of an admissible heuristic concerning the distance $h(x)$ from the current node to the target node. To be admissible, the heuristic should not overestimate the real value of the distance calculated.

This type of algorithm always finds a solution if there is one. In addition, the search process does not typically need to explore all the nodes in the graph to find this solution. The explored search space, and therefore, the complexity of the algorithm, depends on the quality of the heuristics. In the worst case, the order is exponential, whereas the order of the best case (where the estimated heuristic is close to optimal) is linear. This is the main reason for choosing this type of algorithm, since a greedy alternative for building configurations always results in an

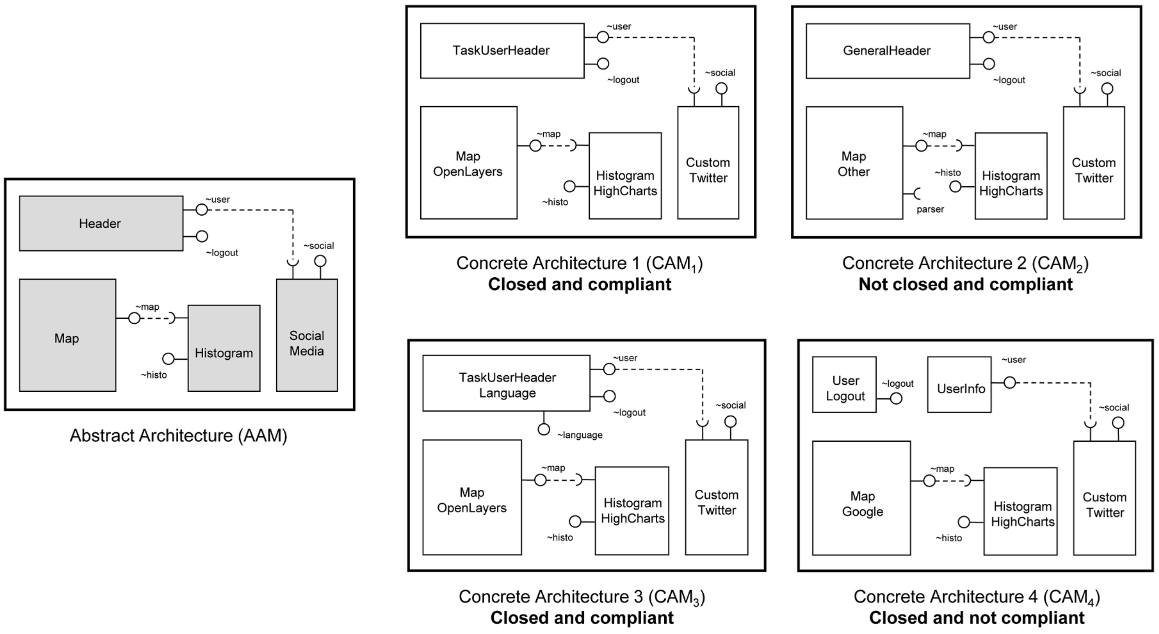


Fig. 11. Example of closure and compliance in concrete architectures.

exponential execution order, because all the nodes in the search space must be evaluated.

Another reason for choosing this type of algorithm is the run-time calculation of configurations. The exploration path always moves toward a solution closer to the target node than the previous state. As such, a reference to the last 'best solution' can be kept and made use of if regeneration is forced to finish the search (for example, due to time constraints or other restrictions). Nevertheless, if the selection of the best possible configuration in the search space must be ensured, the 'greedy mode' of the adapted A* algorithm would be executed by establishing the value of $g(x)$ as equal to 0.0.

In this proposal, each graph node represents a configuration of concrete components, so that one node is adjacent to another if its configuration differs by one component. Thus, every iteration of the A* search algorithm is executed until a configuration that meets the architectural definition is found. Before executing the algorithm, candidate components are grouped by the information in the functional part to limit the search (Fig. 12). The clustering of candidate components is executed before the search algorithm starts, with the aim of improving the performance of our approach. The groups of candidates are inspected at different execution stages of the algorithm when new nodes of the exploration tree are created. If the clustering is not performed at the beginning, the algorithm will explore those branches with groups of candidates already selected faster. Each group is related to the operation of a component in the abstract architecture (architectural definition) and contains those concrete components which have at least one operation (belonging to a provided interface) in common. Accordingly, graph nodes do not contain more than one component of the same group.

A generic A* search algorithm traverses a graph from a starting node and uses two main data structures, a set with the nodes to be evaluated (open nodes) and another set for knowledge about the nodes evaluated (closed nodes). In each step, the algorithm extracts a node from the open set and checks whether the new node can be considered as the goal. If the node is not the goal, it is included in the set of closed nodes, and its neighbors are added to the set of open nodes (if not already included). Each node added must be described by a score, which is obtained from the abovementioned evaluation function $f(x)$ (representing the distance to the goal node). The node extracted from the open set in each iteration is therefore the one with the lowest score. Fig. 13 shows the pseudocode of the algorithm, an adapted A* search.

Nodes represent configurations of concrete components and the goal node is a configuration where $f(x)$ equals 0 and there can be as many goal nodes as configurations which constitute a possible solution of the architecture. The algorithm begins from a starting node (*source*) which is adjacent to every node created from the components of a group of candidates (Fig. 12). Those are the initial nodes on the graph, and the others are created dynamically when a new node is explored (line 28 of the algorithm). Furthermore, new neighbors are only created if the resulting configurations do not exceed the size of the abstract definition

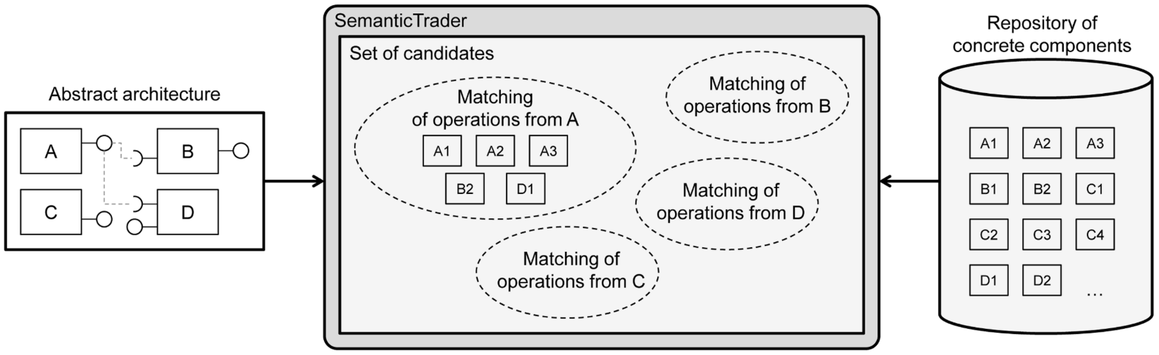

Abstract architecture
A
B
C
D
SemanticTrader
Set of candidates
Matching of operations from A
A1
A2
A3
B2
D1
Matching of operations from B
Matching of operations from D
Matching of operations from C
Repository of concrete components
A1
A2
A3
B1
B2
C1
C2
C3
C4
D1
D2
...

Fig. 12. Clustering candidates from matching of operations.

```
1: function CONFIGURATIONS($source, AAM$)
2:     $openSet \leftarrow \{source\}$                 // Set of nodes to be evaluated
3:     $pQueue \leftarrow \{source\}$                  // Priority queue to manage the open set
4:     $closeSet \leftarrow \emptyset$                 // Set of evaluated nodes
5:     $discardedConfigs \leftarrow 0$                 // Counter of discarded configurations
6:     $notDesiredCC \leftarrow \emptyset$             // Set of non-desired candidates
7:     $firstSolution \leftarrow false$                // Flag to detect when the first solution is found
8:     $bestNode \leftarrow \varnothing$               // Best node of the graph
9:     while $openSet \neq \emptyset$ do
10:        $currentNode \leftarrow pQueue.poll()$
11:        $openSet.remove(currentNode)$
12:        if $currentNode.getH() < bestNode.getH()$ then
13:            $bestNode \leftarrow currentNode$
14:        end if
15:        if $currentNode.getH() == 0$ then
16:            if $firstSolution == false$ then $firstSolution \leftarrow true$
17:            end if
18:            $bestNode \leftarrow currentNode$
19:            if $evaluateCAM(currentNode, AAM) == true$ then
20:                return $bestNode$
21:            else $discardedConfigs \leftarrow discardedConfigs + 1$
22:            end if
23:        else
24:            $closeSet.put(currentNode)$
25:            if $checkCAMSize(currentNode) == true$ then
26:                if $contains(notDesiredCC, currentNode) == false$ then
27:                    $neighbors = \emptyset$      // Set of neighbors to the current node
28:                    $neighbors \leftarrow createNewAdjacentNodes(currentNode)$
29:                    for each $neighbor$ in $neighbors$ do
30:                        if $contains(closeSet, neighbor) == false$ then
31:                            if $contains(openSet, neighbor) == false$ then
32:                                $h \leftarrow heuristics(neighbor, AAM)$
33:                                $newNode \leftarrow createNode(neighbor, h)$
34:                                if $h == 0$ then
35:                                    if $firstSolution == false$ then $firstSolution \leftarrow true$
36:                                    end if
37:                                    $bestNode \leftarrow currentNode$
38:                                    if $evaluateCAM(currentNode, AAM) == true$ then
39:                                        return $bestNode$
40:                                    else $discardedConfigs \leftarrow discardedConfigs + 1$
41:                                    end if
42:                                else
43:                                    $openSet.put(newNode)$
44:                                    $pQueue.add(newNode)$
45:                                end if
46:                            end if
47:                        end if
48:                    end for
49:                end if
50:            end if
51:        end if
52:    end while
53:    return $bestNode$
54: end function
```

Fig. 13. Adapted A* algorithm for generating the configurations in STAS.

(line 25). This limits the algorithm's search space and reduces the number of nodes for which the evaluation function $f(x)$ is calculated.

As mentioned, $f(x)$ is the reference for managing the open node priority queue. The nodes explored in each iteration are selected from this queue (line 10). Focusing on both addends of $f(x)$ (*i.e.*, $g(x)$ and $h'(x)$), we can see that the default value of $g(x)$ is 1.0, since one node differs from another in the incorporation of one concrete component. However, after evaluating the algorithm, some situations (*e.g.*, when the number of candidate components is very large), in which $g(x)$ equals 0.0, enable the architectural solution to be found faster. In such cases, the adapted A* algorithm is equivalent to a greedy search. This variation means that the implementation of the algorithm does not ensure an optimal solution (*i.e.*, the one closest to the starting node). Other operations are therefore responsible for checking that the algorithm does not add extra components to those defined in the abstract architecture. Moreover, $g(x)$ can be configured by the *Admin* interface of the *Semantic Trader*.

The value of $h'(x)$ is the distance between the configuration of concrete components (associated with the current node) and the input *AAM*. This distance is calculated from the semantic information in the functional interfaces and must be 0.0 (lines 15 and 34) to consider a configuration a possible architectural solution. This decision ensures (at least) resolution of a valid configuration of the functional part in less time than if all the parts of the components were to be evaluated. Nevertheless, when a configuration fulfilling the functional part is found (lines 15 and 34), a full evaluation of the configuration is performed by calculating the distance from the *AAM* using all the component parts (including extra-functional information). This evaluation also checks that the configuration is (a) closed and (b) compliant with the abstract architecture. If different constraints are required to consider a configuration as being valid to resolve an abstract architecture, our approach should be modified for this evaluation function. For example, considering the scenario shown in Fig. 11 in which CAM_1 and CAM_3 are not available, perhaps the best output to be obtained is an extended CAM_2 by including a new component that provides the additional required interface (*i.e.*, parser). As a consequence, it is possible to modify the closure and compliance considerations in order to change the constraints for valid architectures.

4.2. Metrics for semantic trading

The metrics applied in the adapted A* algorithm must be defined in more detail. The method employed for calculating the $h'(x)$ score is called *heuristics* (line 32 of the algorithm shown in Fig. 13). For this calculation, a macro-component containing all the information related to the functional specifications of the abstract architecture is created. Similarly, another macro-component gathering all the functional information of the concrete components that form part of the evaluated configuration is built. In both cases, there is a joining of the provided interfaces and the required interfaces which are mandatory (it entails a joining of the operations specified in each interface). The new specifications of macro-components are compared to calculate the matching (*i.e.*, the distance) between them.

In the matching example of Fig. 14 the abstract architecture defines four provided interfaces (a, b, c and d) and two required interfaces (in this case, the same interface twice). The configuration being compared provides the interfaces a, b and d from the concrete components $A3$, $B1$ and $D1$, respectively; and it also meets the required interfaces. However, the interface c is not provided and hence, the configuration mismatches the abstract architecture. As a consequence, the matching between them is partial and the value calculated for the heuristics function (that represents the distance between the configuration and the estimated solution) is greater that zero.

For a better understanding of the matching calculation for the *heuristics* function, Fig. 15 represents a schematic representation of its behavior (top right). From the input parameters (*i.e.*, the abstract architectural model *AAM* and the configuration which is evaluated), the concrete components are joined together in a macro-component, as mentioned before. After that, the matching between the functional parts of both components is calculated.

This matching operation makes a distinction between the provided (*MPI*) and required (*MRI*) interfaces. Both values are described by a real number between 0 (no elements in common) and 1 (total matching) and are calculated from the division of the number of matched operations by the total of operations in the abstract definition. The bottom of Fig. 15 summarizes the behavior for matching the provided interfaces. First, the operations of the abstract architecture are obtained. Second, we check if any operation in the evaluated configuration matches each operation in the *AAM*, thus increasing the value of *matchedProvidedOp*. Two operations match if their semantic descriptions are the same (*i.e.*, if their parameters and the returned value are equivalent), as explained in Subsection 3.2.

The value of *MPI* factor is calculated from the division between *matchedProvidedOp* and *acProvidedOp*, where the divider represents the total number of provided operations in *AAM*. In addition to the matching values, the heuristics function stores certain information about which operations (and which interfaces they belong to) of a configuration resolve the operations of the abstract definition. This information is useful for optimizing performance when the concrete architecture is built (the last step in generating configurations). Furthermore, other complementary information is calculated to enable the analysis of the matching: (a) the type of intersection, and (b) the identification of which

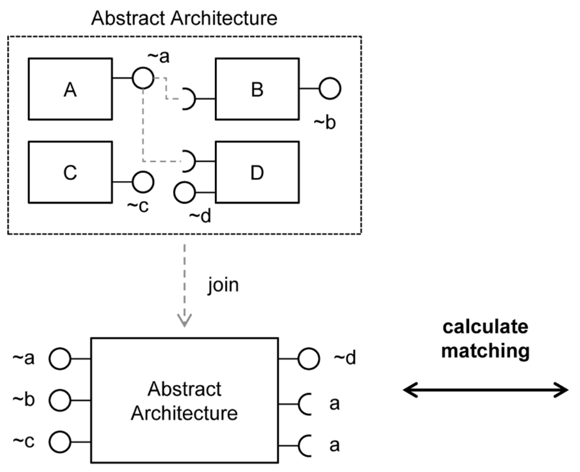
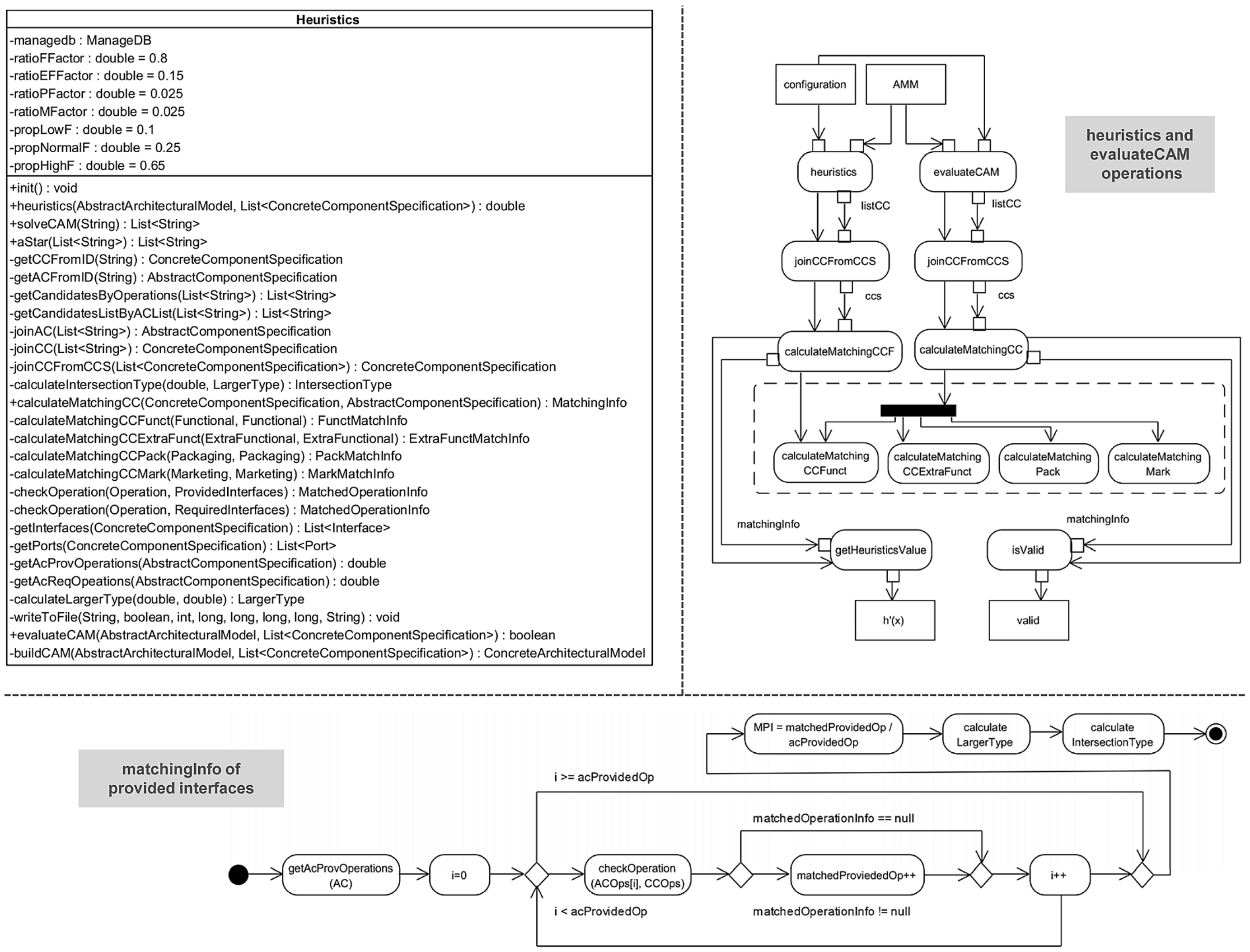


Fig. 14. Matching between an abstract architecture and a configuration.



```
Heuristics
-managedb : ManageDB
-ratioFFactor : double = 0.8
-ratioEFFactor : double = 0.15
-ratioPFactor : double = 0.025
-ratioMFactor : double = 0.025
-propLowF : double = 0.1
-propNormalF : double = 0.25
-propHighF : double = 0.65
+init() : void
+heuristics(AbstractArchitecturalModel, List<ConcreteComponentSpecification>) : double
+solveCAM(String) : List<String>
+aStar(List<String>) : List<String>
-getCCFromID(String) : ConcreteComponentSpecification
-getACFromID(String) : AbstractComponentSpecification
-getCandidatesByOperations(List<String>) : List<String>
-getCandidatesListByACList(List<String>) : List<String>
-joinAC(List<String>) : AbstractComponentSpecification
-joinCC(List<String>) : ConcreteComponentSpecification
-joinCCFromCCS(List<ConcreteComponentSpecification>) : ConcreteComponentSpecification
-calculateIntersectionType(double, LargerType) : IntersectionType
+calculateMatchingCC(ConcreteComponentSpecification, AbstractComponentSpecification) : MatchingInfo
-calculateMatchingCCFunct(Functional, Functional) : FunctMatchInfo
-calculateMatchingCCExtraFunct(ExtraFunctional, ExtraFunctional) : ExtraFunctMatchInfo
-calculateMatchingCCPack(Packaging, Packaging) : PackMatchInfo
-calculateMatchingCCMark(Marketing, Marketing) : MarkMatchInfo
-checkOperation(Operation, ProvidedInterfaces) : MatchedOperationInfo
-checkOperation(Operation, RequiredInterfaces) : MatchedOperationInfo
-getInterfaces(ConcreteComponentSpecification) : List<Interface>
-getPorts(ConcreteComponentSpecification) : List<Port>
-getAcProvOperations(AbstractComponentSpecification) : double
-getAcReqOpeations(AbstractComponentSpecification) : double
-calculateLargerType(double, double) : LargerType
-writeToFile(String, boolean, int, long, long, long, long, String) : void
+evaluateCAM(AbstractArchitecturalModel, List<ConcreteComponentSpecification>) : boolean
-buildCAM(AbstractArchitecturalModel, List<ConcreteComponentSpecification>) : ConcreteArchitecturalModel
```

Fig. 15. Main class of the Heuristics microservice (top-left), *heuristics* and *evaluateCAM* operations (top-right), and matching of provided interfaces (bottom).

component has the larger part. This information is used to extend the information about the ratio of the matching but, for subsequent improvements, it can be used for other purposes, such as hiding unwanted functionality, building a wrapper for the component, or fixing the unprovided operations, among other possible examples.

Fig. 16 shows the possible combinations of the intersection and larger types. For example, when the evaluated part of the abstract component (*AC*) is larger than the corresponding part of the concrete component (*CC*) and the intersection is equal to the concrete component, the matching is lower than 1 because a certain part of the *AC*

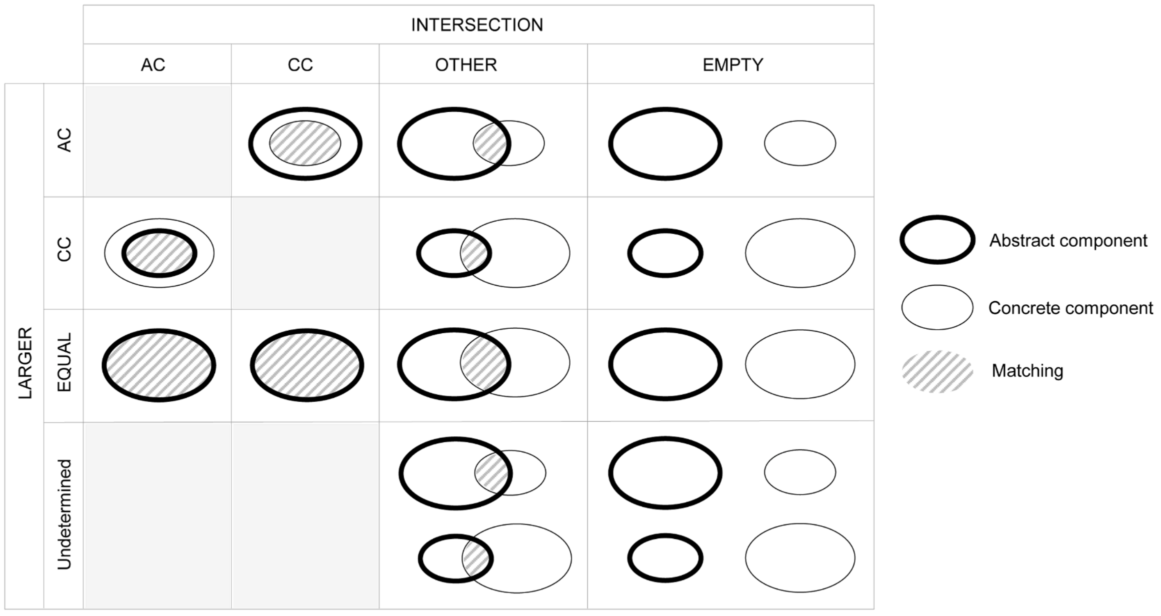


Fig. 16. Possible combinations of the intersection and larger types during the matching between abstract and concrete components.

specification is not fulfilled. Otherwise, when the evaluated part of the *AC* is smaller than the corresponding part of the *CC* and the intersection is equivalent to the *AC*, there is a total match although the additional parts of the concrete component must be taken into account in order to avoid undesired behaviors. The optimal combinations with total matching occur when both parts are equal and the intersection is equivalent to *AC* or *CC*. When the intersection is identified as *other*, the matching value is lower than 1, and when it is *empty* the value is equal to 0.

The following expressions summarize the calculation of the matching related to the functional information (*MF*) performed in the *heuristics* method, and obtained from the matching of provided and required interfaces:

$$MPI = \frac{matchedProvidedOp}{acProvidedOp} \qquad MRI = \frac{matchedRequiredOp}{acRequiredOp}$$

$$MF = \frac{MPI + MRI}{2}$$

As a consequence, the distance value (or the evaluation function) is calculated from the matching value:

$$h'(x) = 1 - MF$$

Architecture closure and compliance are calculated by the *evaluateCAM* method (lines 19 and 38 of the algorithm in Fig. 13), which is executed when the evaluation function $h'(x)$ is equal to 0. This operation performs a new calculation of the matching. In this case, all parts of the component specification are taken into account (Fig. 15):

(a) Functional information (*MF*). The *heuristics* method is executed to obtain the corresponding metrics.
(b) Extra-functional information (*MEF*). Properties and dependencies are compared:
  (b.1) Properties. First, the properties meeting the abstract definition are selected. Then, the matching value is calculated as a weighted sum of the three types of properties (*i.e.*, high, normal or low priority). High-priority properties have greater weighting than low priority.
  (b.2) Dependencies. The type of intersection between the concrete component and the corresponding abstract definition (with regard to dependencies) is considered. If there is no intersection and they are not empty, then there is a zero match. If there is intersection (see Fig. 16), depending on the type, there are three possibilities: (*i*) all concrete component dependencies (DCC) are within the set of abstract component dependencies (DAC), *i.e.*, DAC contains DCC; (*ii*) DCC contains DAC; and (*iii*) there is intersection but no set contains the other. The corresponding formulas that use the number of paired dependencies (*matchedDep*) are shown below:

$$(i)\quad m = \frac{matchedDep}{card(DAC)} \qquad (ii)\quad m = \frac{matchedDep}{card(DCC)}$$

$$(iii)\quad m = \frac{matchedDep}{card(DAC) + card(DCC)}$$

(c) Packaging information (*MP*). The score calculated shows whether the components are implemented with the same technologies and whether they are stored in the same repository.

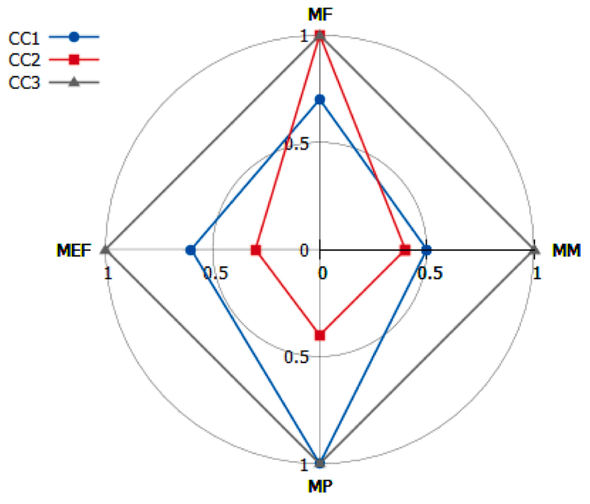


Fig. 17. Example of matching scores for semantic trading.

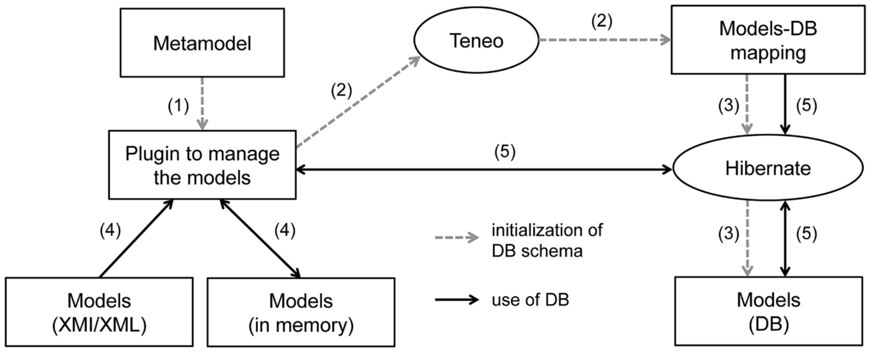


Fig. 18. Persistence of models of components and architectures.

(d) Marketing information (*MM*). The distance calculated shows whether the components were implemented by the same development team.

All the metrics representing the matching values of the *evaluateCAM* method are described by a real number between 0 (no elements in common) and 1 (total matching). For all the subparts, the type of intersection and the identification of which component has the larger set of elements are also calculated (Fig. 16). Fig. 17 shows an example of three scored components. In this case, the component that best matches the abstract definition is *CC*3. However, there are differences in *CC*1 and *CC*2 which must be discussed. *CC*1 matches the functional information better, but the other parts are complied better in *CC*2. Moreover, the total area derived from the scores of *CC*2 is greater than the area related to the scores of *CC*1. In conclusion, a component is considered to match the abstract specification better than another depending on the importance of each part. Therefore, the total value of the match is weighted using the following formula:

$$\begin{aligned} matching = {} & MF^{*}factorMF + MEF^{*}factorMEF + MP^{*}factorMP \\ & + MM^{*}factorMM \end{aligned}$$

The default values for the weighting factors are 0.8 (*factorMF*), 0.15 (*factorMEF*), 0.025 (*factorMP*) and 0.025 (*factorMM*). These values were set based on the construction of quality models [54] which determined that while the functional information is the most important part, the extra-functional information is less important, but has a considerable impact on the matching score. At the same time packaging and marketing make up the remaining weight. Nevertheless, these values can be modified by using the *Admin* interface of the *Semantic Trader* whenever the sum of the factors is equal to 1. Furthermore, this interface enables the configuration of the minimum distances that must be accomplished to consider a configuration as a solution. For example, the distance for the extra-functional part can be set within the interval [0.0,0.2]if the minimum *MEF* matching score is modified to 0.8.

5. Implementation of the semantic trader

Repositories of components and architectures must be managed by an efficient persistence mechanism. When the size and the number of models used in an application is small, the load in the memory may be adequate, but if the number of models grows, they need to be stored persistently. In this case, some model (*e.g.*, component specifications) export and import mechanisms must also be provided at run-time. As such, models of components and architectures are stored in a database.

The proposed DSLs and their instances (*i.e.*, models) were developed with the Eclipse Modeling Framework (EMF)[4]. In EMF, models are implemented as Java objects once they are loaded into the memory. Accordingly, they can be made persistent by applying Object-Relational Mapping (ORM) techniques, such as Java Persistence API (JPA). One of the most used JPAs implementations is Hibernate[5], which enables mapping between objects and a relational database to be defined by annotations in the classes, or a file describing such associations.

Mapping between the objects representing the models and the database can be done automatically with Teneo[6], which generates the description files from the DSL metamodels (see Fig. 18). First, plugin Java classes used to manage the models are obtained from the metamodel. Then, mapping between the classes (which manage the models) along with the database are generated using Teneo. Third, the database schema is initialized and the tables are created using Hibernate. In the following operations, when the database is accessed (to execute operations of query, insertion, update, deletion, etc.), the mapping and plugin classes are used to manage the models (steps 4 and 5 in Fig. 18).

The implementation of the Semantic trader was designed for remote invocation of all its operations. Its functionality was therefore developed using a microservice approach [58]. Such services are accessible both internally by the system and externally by users or other processes via the web. An example of external processes are those acting as exporters and importers of components. With regard to internal processes, some examples are the calculation of configurations, or the management of trader policies. Functionality is divided into five microservices: *Lookup, Register, Admin, Configs* and *Heuristics*. The first three web services mentioned implement trader operations in a standalone mode. The *Configs* service implements the operations related to the calculation of configurations, and the *Heuristics* service includes those methods used to evaluate and score components and configurations.

Fig. 19 shows the microservice-based architecture that implements our semantic trading proposal. The *Lookup* microservice is used by the *Register, Configs* and *Heuristics* microservices, because the first contains the operations for finding the components matching an input criterion, while the *Configs* module uses the *Heuristics* microservice for calculating the configuration scores. The *Lookup* microservice uses the *Admin* module because it must read trader policies, such as the maximum number of specifications returned by a query. In addition, the *Configs* module also uses the *Admin* microservices, for example, to read match factors or the minimum distances (if any) required for valid configuration, as mentioned above.

Microservices are implemented as Spring Boot[7] applications exposing their operations and resources through RESTful web services [59]. Ribbon and Feign are used for internal communication for accessing and load balancing, and Eureka for registering and discovering services [60]. This architecture makes semantic trading resilient to

[4] Eclipse Modeling Framework (EMF) – https://www.eclipse.org/modeling/emf/

[5] Hibernate – http://hibernate.org

[6] Teneo/Hibernate – http://wiki.eclipse.org/Teneo/Hibernate

[7] Spring Boot – https://spring.io/projects/spring-boot

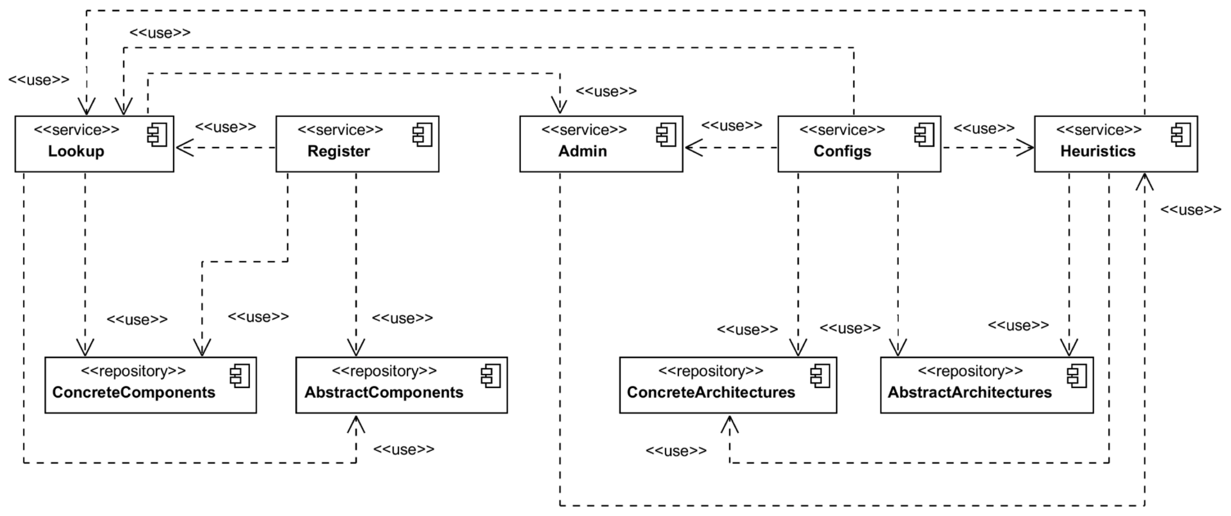

<<service>> Lookup
<<service>> Register
<<service>> Admin
<<service>> Configs
<<service>> Heuristics
<<repository>> ConcreteComponents
<<repository>> AbstractComponents
<<repository>> ConcreteArchitectures
<<repository>> AbstractArchitectures
<<use>>

Fig. 19. Proposed architecture for semantic trading.

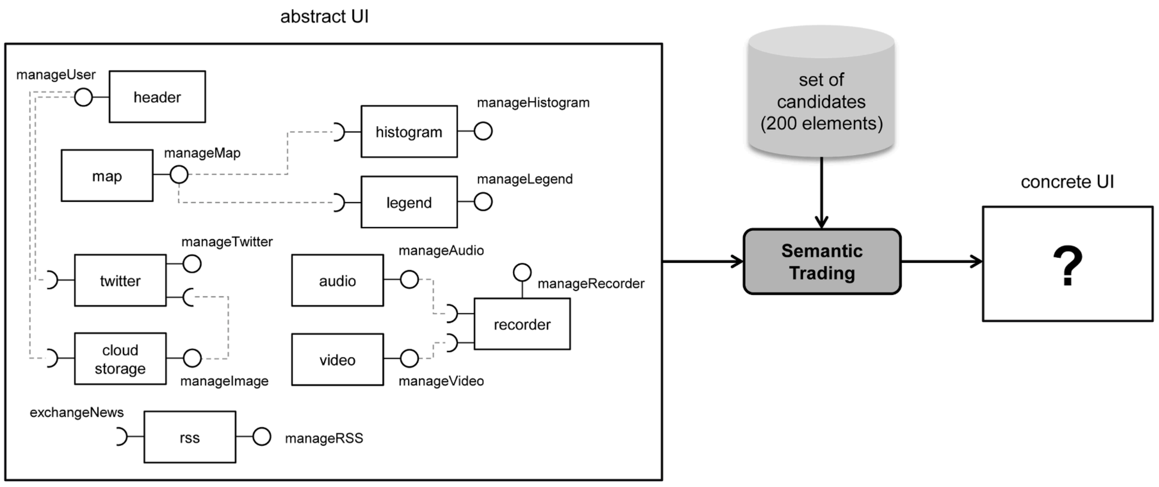


Fig. 20. Example of an abstract architecture managed by the STAS.

change in the execution environment. For example, if a high demand for the *Register* microservice is detected (during insertion and updating of component specifications), then that microservice can be replicated to cover all requests.

The implementation of the microservices is available in a GitHub repository[8]. The application of the STAS approach to other domains or architectures of smart environments is made possible by using the following guidelines. First, a new vocabulary of complex types for defining the inputs and outputs of the component operations must be developed. Second, the target scenario components must be modeled using the proposed languages. The two levels of representation (abstract and concrete) must be considered. Thus, each set of concrete components with functional interfaces in common would have a related abstract component specification. All the component specifications must be inserted into the repositories by using the *Register* microservice. Moreover, the *Admin* microservice can be used to modify the configurable parameters. Before starting the resolution of the concrete architecture, the abstract architecture used as input must be inserted into the database by using the *Configs* microservice. After that, the search algorithm can be executed to generate the resulting configuration and save the corresponding concrete architecture in the database. The *Configs* microservice uses the *Heuristics* and *Lookup* microservice to operate.

6. Generating software architectures using the STAS approach: a case study for mashup UIs

As discussed above, the Semantic Trader solution proposed can generate software architectures, thus enabling their adaptation at runtime. Examples of application domains are dynamic and flexible architectures based on coarse-grained components, which are very useful in ecosystems related to smart city solutions, such as smart buildings, smart grids, smart homes or smart UIs. In this section, we apply our semantic trading approach to a particular domain of smart architectures, called mashup UIs. The scenario used for validation and evaluation experiments is related to the ENIA frontend, a mashup web-based user interface in an environmental GIS application (described in Section 2).

This case study regenerates concrete architectures from an abstract definition formed by the following types of components: map, histogram, legend, header, storage, audio, video, recorder, social network and RSS reader (see Fig. 20). Architectures made up of mashup UIs in this domain may have component dependencies. For example, the recorder component requires some audio and video operations, and the social network component requires some user information provided by the header, whereas the RSS reader is an isolated component. The

[8] GitHub repository containing the implementation of the STAS approach – https://github.com/acgtic211/costrader

J. Criado, L. Iribarne, N. Padilla.(2021)Heuristics-based mediation for building smart architectures at run-time. Computer Standards & Interfaces. Elsevier, Volume 75, April 2021, 103501.ISSN: 0920-5489

behavior of each component is not described, since the purpose here is to create a suitable scenario for validating our approach.

All the abstract components in the case study have only one provided interface, but this contains a set of provided operations. For example, the provided interface in the header is formed by two operations, *getUserInfo* and *login*. All of the operations are mandatory except for the RSS (which can operate without resolving the *exchangeNews* interface) and twitter (which may or may not be connected to the *manageImage* interface) components.

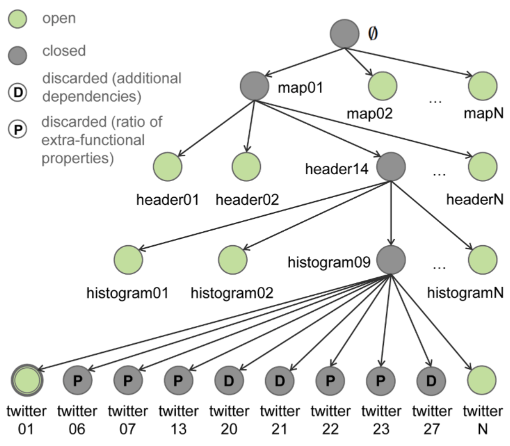
The set of concrete components used in the example scenario (200 elements) makes it possible to analyze all the matching possibilities. As such, there are components with additional provided and/or required interfaces, or components with the same functional interface, but containing additional operations. We have also defined extra-functional, packaging and marketing information in some components. We performed tests with the different combinations of initial abstract architectures that could be constructed with the components in Fig. 20, varying the size of the input architecture. Thus, when performance in resolving architectures with only one component is evaluated, the map is resolved. The component map, histogram and header are used when the architectures of three elements are evaluated, and the complete abstract architecture is used for experiments with ten elements.

6.1. Calculation of alternative configurations

In the tests conducted for this case study, 20 concrete specifications were defined for each of the component types shown in Fig. 20 (*i.e.*, for each abstract component). This represents a total of 200 candidate components (10*20) for architectures made up of the 10 types of components. Clustering limits the search space to a combinatorial number of $\binom{20}{10} = 184756$ configurations. This set of candidate components was inserted in a repository with another 10000 specifications of concrete components that were generated automatically (from the random combination of the previous 200 specifications). With these components, the regeneration process could be validated and evaluated for a significantly large scenario (1200 elements in total). Fig. 21 shows an example execution of semantic trading illustrating the behavior of the adapted A* search algorithm.

The abstract architecture (*i.e.*, the input architectural definition) is made up of four components (shown at the top of the table): a map, a histogram, a header, and a social network component. To execute this algorithm, the first step is to obtain the 90 candidate components (*CC*) extracted from the component repository (*CCR*) mentioned above. Note that abstract and concrete components are described schematically, listing only the provided (represented by the name of the interface, *e.g.*, *interfaceName*) and the required interfaces (represented by the name of the interface with a line above it, *e.g.*, $\overline{interfaceName}$). In this example, all required interfaces are mandatory.

Algorithm execution starts by evaluating the components located in one of the groups found in the candidate clusters (Iteration 1). As a result of this iteration, the node having the lowest evaluation function value (0.8 in this case) is the one formed by component `map01`. From this node, in Iteration 2, nodes are dynamically created adjacent to it. These new nodes are formed by the combination of component `map01` with those concrete components belonging to a different group of candidates used in the previous iteration. Accordingly, adjacent nodes such as [`map01,header01`], [`map01,header02`], etc., are created from [`map01`]. Following this, the search algorithm selects the graph node with the lowest score (found by the evaluation function). In this case, the



Abstract components
map = {$manageMap$}, histogram = {$manageHistogram, \overline{manageMap}$}, header = {$manageUser$}, twitter = {$manageTwitter, \overline{manageUser}$}
Candidates components (*CC*) – Total = 90
map01 = {$manageMap$}
map02 = {$manageMap, \overline{feedback}$}
map03 = {$manageMap, \overline{servicesOGC}$}
map04 = {$manageMap, manageHistogram$}
...
histogram09 = {$manageHistogram, \overline{manageMap}$}
histogram10 = {$manageHistogram$}
...
header01 = {$manageUser, language, \overline{logout}$}
header02 = {$manageUser, logout, \overline{weather}$}
...
twitter01 = {$manageTwitter$}
twitter02 = {$manageTwitter, \overline{manageImage}$}
...

Calculation of configurations

#	Configuration (Node)	*CC* added	$h'(x)$	*ev.CAM*
1	∅	map01	0.8	–
2	[map01]	header14	0.7	–
3	[map01,header14]	histogram09	0.19	–
4	[map01,header14,histogram09]	twitter21	0.0	(1)
5	[map01,header14,histogram09]	twitter22	0.0	(2)
6	[map01,header14,histogram09]	twitter23	0.0	(2)
7	[map01,header14,histogram09]	twitter06	0.0	(2)
8	[map01,header14,histogram09]	twitter13	0.0	(2)
9	[map01,header14,histogram09]	twitter07	0.0	(2)
10	[map01,header14,histogram09]	twitter27	0.0	(1)
...				
17	[map01,header14,histogram09]	twitter20	0.0	(1)
18	[map01,header14,histogram09]	twitter01	0.0	(*)

(1) → Discarded: the configuration has some additional dependencies
(2) → Discarded: the configuration does not match the ratio of extra-functional properties
(*) → Final solution

Fig. 21. Example of alternative configurations.

node selected is formed by components `map01` and `header14` which have a score of 0.7.

The algorithm continues this way until it finds a node with a heuristic function equal to 0.0, at which point it proceeds to evaluate the configuration corresponding to that node. In the example, configuration checking is done after Iteration 4. In some cases, the configurations are discarded because any given component has additional dependencies and hence it is a not closed configuration (*e.g.*, Iterations 4, 10 and 17). In other cases, the configurations analyzed are closed, but they do not conform to the architectural definition. For example, possible solutions analyzed in Iterations 5, 6, 7, 8 and 9, are discarded, because one of their components does not meet the minimum matching ratio for extra-functional properties. In Iteration 18, after excluding a total of 14 possible solutions, the algorithm finds a configuration evaluated as the final solution. Based on this configuration (consisting of components `map01`, `header14`, `histogram09` and `twitter01`), a concrete architectural model is constructed describing a UI which looks like the web application shown in Fig. 3. The left hand side of Fig. 21 shows the nodes evaluated by the algorithm.

6.2. Scoring a possible solution

The Heuristics microservice calculates the value of the evaluation function $h'(x)$ for each configuration of concrete components that need to be analyzed. In the 'greedy mode' of the adapted A* search ($g(x) = 0.0$), the configurations are complete (with regard to the abstract definition) and are possible solutions. In the case of the normal behavior of the adapted A* search ($g(x) = 1.0$), the configurations may also be partial solutions of the architecture. In both cases, the score of a configuration is calculated from the individual matching scores for each of the concrete components forming the configuration.

The XML code shown in Fig. 22 is the result returned by the operation that calculated the match of a concrete component. In this example, the elements compared are the abstract component `twitter` and the concrete component `twitter13`. The functional information is only partially matched, because the provided interface matching is 1.0, whereas the required interface matching is 0.0. As such, the ratio of the functional part is 0.5=(1.0+0.0)/2.

Regarding the extra-functional part, the operation returns matching scores for dependencies and properties. Since there is an equivalent dependency (because the identifier of the required interface matches), but the concrete component has an additional dependency, the matching value for dependencies is 0.5. In the case of properties, there is correspondence with one of them, whose priority is high (among the four properties). Thus, the matching value for properties is 0.52. As a result, the matching for the extra-functional part is 0.51=(0.5+0.52)/2.

For the extra-functional part, the operation returns matching scores for dependencies and properties. There is an equivalent dependency (because the identifier of the required interface matches), but the concrete component has an additional dependency. Consequently, the match value for dependencies is 0.5. One of the four properties, whose priority is high, matches the abstract component specification. Thus, the matching value for properties is 0.52, and as a result, the matching score for the extra-functional part is 0.51=(0.5+0.52)/2. The score for the packagingis 1.0, since the concrete component matches the location and implementation data described in the abstract component specification. There is no correlation with marketing information, and therefore, the score is 0.0. For these matching values, the total score is 0.5265=0.85*0.5+0.15*0.51+0.025*1.0+0.025*0.0.

```
<MatchingInfo>
  <ratio>0.5265</ratio>
  <FunctionalMatchInfo>
    <ratio>0.5</ratio>
    <intersection>OTHER</intersection>
    <larger>AC</larger>
    <provided>
      <ratio>1.0</ratio>
      <intersection>OTHER</intersection>
      <larger>EQUAL</larger>
      <acNProvidedIfaces>1</acNProvidedIfaces>
      <ccNProvidedIfaces>1</ccNProvidedIfaces>
      <matchedPI>1</matchedPI>
      <involvedPI>1</involvedPI>
      <acNProvidedOps>2</acNProvidedOps>
      <ccNProvidedOps>6</ccNProvidedOps>
      <matchedPOps>2</matchedPOps>
      <matchedInterface>
        <acInterfaceID>manageTwitter</acInterfaceID>
        <ccNumInterfaces>1</ccNumInterfaces>
        <matchedOperation>
          <acOperationID>filter</acOperationID>
          <ccOperationID>filter</ccOperationID>
          <ccInterfaceID>manageTwitter</ccInterfaceID>
          <inputType>trader:filterInput</inputType>
        </matchedOperation>
        <matchedOperation>
          <acOperationID>removeAvatar</acOperationID>
          <ccOperationID>removeAvatar</ccOperationID>
          <ccInterfaceID>manageTwitter</ccInterfaceID>
          <inputType>trader:removeAvatarInput</inputType>
        </matchedOperation>
      </matchedInterface>
    </provided>
    <required>
      <ratio>0.0</ratio>
      <intersection>EMPTY</intersection>
      <larger>AC</larger>
      <acNRequiredIfaces>2</acNRequiredIfaces>
      <ccNRequiredIfaces>1</ccNRequiredIfaces>
      <matchedRI>0</matchedRI>
      <involvedRI>0</involvedRI>
      <acNRequiredOps>2</acNRequiredOps>
      <ccNRequiredOps>1</acNRequiredOps>
      <matchedROps>0</matchedROps>
    </required>
  </FunctionalMatchInfo>
  ...
  <ExtraFunctionalMatchInfo>
    <ratio>0.51</ratio>
    <intersection>OTHER</intersection>
    <larger>UNDETERMINED</larger>
    <dependencies>
      <ratio>0.5</ratio>
      <intersection>OTHER</intersection>
      <larger>CC</larger>
      <acNumDependencies>1</acNumDependencies>
      <ccNumDependencies>2</ccNumDependencies>
      <matchedDependencies>1</matchedDependencies>
    </dependencies>
    <properties>
      <ratio>0.52</ratio>
      <intersection>OTHER</intersection>
      <larger>AC</larger>
      <acNProps>4</acNProps>
      <ccNProps>2</ccNProps>
      <matchedProps>1</matchedProps>
      <acNPropsLow>1</acNPropsLow>
      <matchedPropsLow>0</matchedPropssLow>
      <acNPropsNormal>2</acNPropsNormal>
      <matchedPropsNormal>0</matchedProsNormal>
      <acNPropsHigh>1</acNPropsHigh>
      <matchedPropsHigh>1</matchedPropsHigh>
    </properties>
  </ExtraFunctionalMatchInfo>
  <PackagingMatchInfo>
    <ratio>1.0</ratio>
    <intersection>OTHER</intersection>
    <larger>CC</larger>
    <implementation>
      <ratio>1.0</ratio>
      <intersection>EMPTY</intersection>
      <larger>CC</larger>
    </implementation>
    <location>
      <ratio>1.0</ratio>
      <intersection>AC</intersection>
      <larger>EQUAL</larger>
    </location>
  </PackagingMatchInfo>
  <MarketingMatchInfo>
    <ratio>0.0</ratio>
    <intersection>EMPTY</intersection>
    <larger>CC</larger>
  </MarketingMatchInfo>
</MatchingInfo>
```

Fig. 22. Example of matching score for a concrete component.

6.3. Performance

As part of the scenario for the case study presented, different tests have been carried out to evaluate the performance of our approach. With regard to the operation for obtaining the candidate components, we tested our repository containing 1200 elements (200 manually constructed candidates in addition to 1000 components randomly generated from the candidates). This operation has a significant impact on performance because it is the first step before calculating the possible configurations. Fig. 23 (non-dashed line) shows the execution times for obtaining the candidates in the case of abstract definitions that vary from 1 to 10 components. The execution time grows in a linear fashion with a high value gradient and is not suitable for a trading process at run-time. For this reason, we reduced the execution time of this operation by caching the repository of concrete components in the configuration file of the database access (see the bottom of Fig. 23). From this improvement, the execution time for obtaining the candidates from the database is reduced to a maximum of 264.2 milliseconds when there are 10 components in the abstract architecture.

The following experiments were related to the generation of concrete architectures from abstract definitions ranging in size from 1 to 10 components. For each architecture size, the semantic trading process was executed 100 times and then we calculated the average obtained for the following data: (*i*) total time to obtain the final architectural solution (Fig. 24(a)), (*ii*) time when the first functional solution is obtained (Fig. 24(b)), and (*iii*) number of configurations discarded during the process (Fig. 24(c)). These tests were performed using an Eclipse 2019-03 framework on a 3.70 GHz Intel(R) Core(TM) i7-8700K machine with 16 GB of main memory.

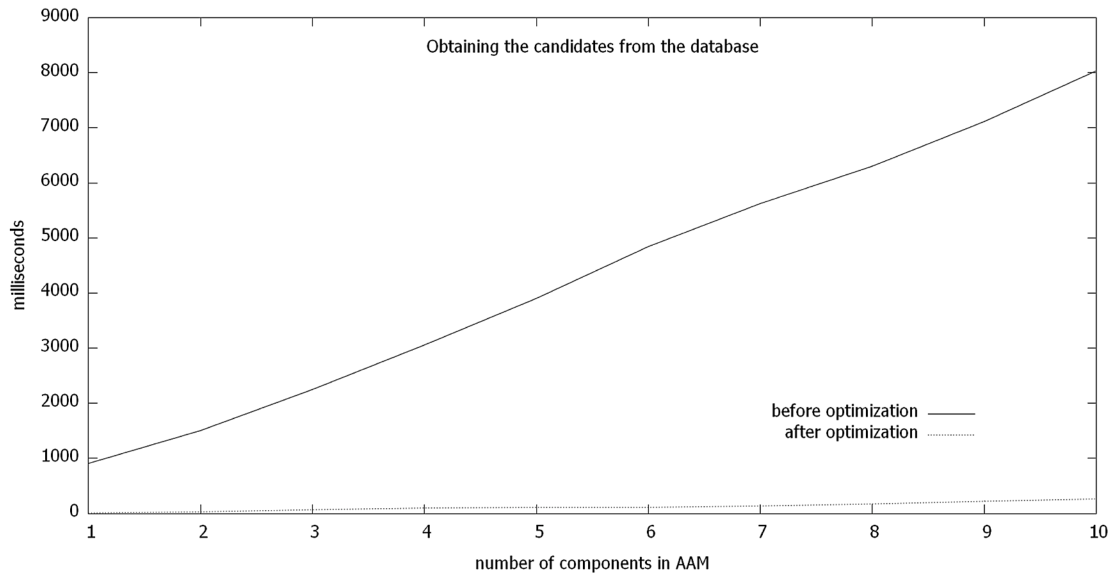
The results in Fig. 24(a) can be approximated by a linear function when the number of components of the architecture is increased. Two types of executions of the search algorithm are shown. The dashed line shows the results when the search has a greedy behavior ($g(x) = 0$), whereas the normal line shows the results for the A* search algorithm when the distance function of the heuristics is increased by one for each additional component in the architecture ($g(x) = 1$). It is observed that the results obtained for the A* search are always better than the execution times obtained from the greedy exploration.

The data in Fig. 24(b) offer a guarantee of how reliable the process is at obtaining at least one solution that has been matched in the functional part (there is a trend between linear and logarithmic when the size of the architecture is increased). Similarly to the results for the final solution, the A* search finds the first functional solution in a better time than the search with a greedy behavior. Between the first functional solution and the final solution, invalid configurations are discarded, reaching values of almost 30 discarded configurations for architectures of 10 components (see Fig. 24(c)). The number of discarded configurations is greater for the normal execution of the search algorithm from a certain number of components. This result is due to the higher number of possible solutions than the algorithm evaluates to decide if a valid configuration has been found; *i.e.*, while a greedy search is exploring paths and nodes to find a solution, the search based on an A* algorithm is validating and discarding potential solutions.

The configuration of the *Semantic Trader* for these experiments is the most restrictive, *i.e.*, the matching distance calculated in the algorithm must be 0.0 for all parts of the components in the architecture. This maximizes the total time to obtain a final architecture, in order to validate the process correctly. The highest times are around 0.3 seconds for interfaces with 10 components, which are acceptable times because mashup UIs do not usually consist of a large number of components at once (since these coarse-grained components encapsulate the functionality of a mini-application). Apart from mashup UIs, the obtained execution times are suitable for most of the systems that require reconfiguration of architectures at run-time. Nevertheless, these times may not be suitable for other environments that require a higher speed during the process of adaptation, such as robotic architectures running time-critical tasks.

7. Discussion

This section discusses the benefits of the approach by analyzing the achievement of the proposed research question. Additionally, the



num of comp. in AAM	before optim. (ms)	after optim. (ms)
1	910,98	12,0
2	1505,11	30,7
3	2250,00	70,1
4	3057,36	99,1
5	3902,55	108,2
6	4843,41	108,4
7	5625,52	137,2
8	6303,72	174,6
9	7114,00	223,0
10	8035,76	264,2

```
hibernateProperties.setProperty("hibernate.cache.region.factory_class", "org.hibernate.
    ↪ cache.ehcache.EhCacheRegionFactory");
hibernateProperties.setProperty("hibernate.cache.use_second_level_cache","true");
hibernateProperties.setProperty("teneo.mapping.default_cache_strategy", "READ_ONLY")
```

Fig. 23. Cache strategy to get the candidates by pre-fetching the concrete components..

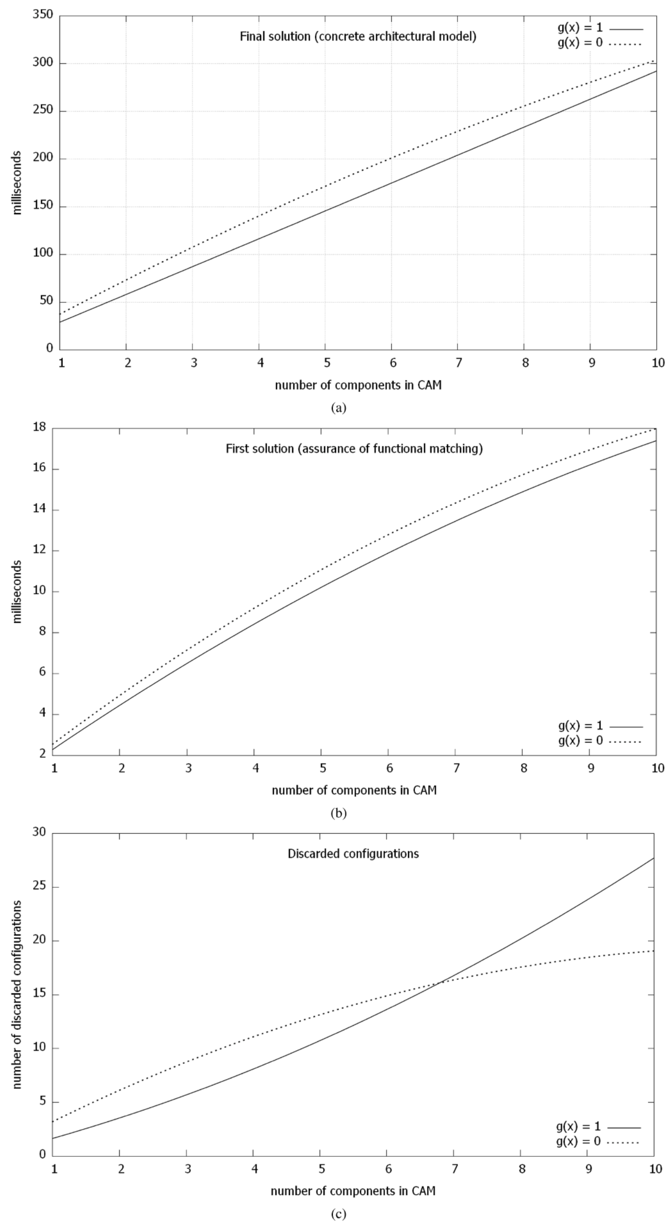


Fig. 24. Execution times of (a) calculating the final architectural solution, and (b) calculating the first functional solutions. Number of configurations discarded during the process of calculating the final solution (b)..

possible threats to validity and the drawbacks are described.

What representation is appropriate for describing the components and architectures managed in our approach?

Focusing on the specific domain of architectures of coarse-grained components that support dynamic changes at run-time, the created DSLs provide a specific set of concepts and relations adequate for resolving software architectures by applying a trading process. Since these languages are also required by the transformation process which is part of our adaptation methodology [5,48], we chose to use them in the process that is executed after it, *i.e.*, the presented semantic trading approach.

Consequently, the DSLs have been proved to be more than capable of describing components and architectures in our approach, but we cannot state with certainty that these languages are the most appropriate representations. In the current state of this research, there is a lack of comparison with other languages, such as UML profiles or other existing ADLs. The use of better-known languages may not affect the performance of the process, but it could be of benefit for improving understanding and reuse by the community.

Can this approach be applied in different domains or scenarios of smart architectures?

The approach can be useful for those types of architectures that meet certain specific requirements related to structure, functional definition, extra-functional properties, communication and dynamic changes. With regard to the last issue, we propose to apply our process to a subset of software architectures that need to be adapted or reconfigured at run-time, thus requiring a process of calculating new configurations that are built conforming to new input specifications, *i.e.*, reference abstract architectures.

From our point of view, this adaptation in general and the resolution of new configurations in particular involve a kind of intelligence, even if it is considered as a low-level type of smart behavior. During the explanation of the context of our approach, we illustrated its application to different domains, such as smart cities, smart homes, or user interfaces. Moreover, we validated and evaluated our approach with mashup UIs, a particular domain of smart architectures. As a consequence, we can state that our STAS approach can be applied to different smart architectures.

How can a traditional trading service be extended to calculate the best configurations of components?

Supported by the results of previous research work [18,30], we have described how a traditional trading service can be adapted to manage component specifications and to calculate component configurations. The proposed extension is focused on a new module in charge of calculating architectural solutions from a repository of component specifications and a target reference architecture. The calculation of configurations requires the development of additional functionality to select the candidates, calculate the distance from a configuration to the pursued solution, evaluate whether or not a configuration is valid for resolving the established abstract architecture, and build the concrete architecture when a configuration of components has been evaluated.

The main contribution of this paper is related to the heuristics-based generation of configurations. This process is supported by an adapted A* search algorithm because it ensures a solution will be found if one exists, without having to explore all the possible configurations to find this solution. Thus, we proposed an extension of a trading service which includes only one type of search algorithm. Although this algorithm can be configured to partially modify its behavior (for example to execute a greedy search), the generation of configurations can be improved in future versions by implementing new search algorithms or different heuristics alternatives.

What is the most important syntactic and semantic information when searching for architectural configurations that must be included in a component definition and still be considered manageable?

This research question is resolved by a general point of view in the literature, since the functional part of a component is the most important. Our approach uses complex types for ensuring the semantic matching of input parameters and the output value of operations. In addition to the functional interfaces, the extra-functional properties are essential for ensuring the fulfillment of non-functional attributes and QoS features. Due to the previous research work related to COTS components, we included additional parts in the component definition to describe some relevant packaging and marketing information. For example, a component included in the abstract architecture to be resolved, can be explicitly specified to have been developed by a particular contact of an entity or company.

With regard to the particular considerations that must be taken into account from the point of view of a specific domain, the search process and the calculation of the matching between architectures can be enriched by determining a set of suitable score metrics related to this domain, for example, building a quality model as in the case of our application domain of ENIA mashup UIs [54].

How can the performance of this search be addressed to get suitable results at run-time?

The search process developed in the STAS approach ensures the calculation of a configuration that fulfills the reference architecture (if the solution exists and can be constructed from the repository of components). Furthermore, the execution of the process must be finalized within a time that allows the resolution of the architecture at run-time. The performed evaluation shows a maximum execution time lower than 0.3 seconds which is a valid time for our purpose of adapting mashup user interfaces. For other domains running time-critical tasks, the execution times may not be suitable. In this regard, the selection of our approach as a solution for building smart architectures at run-time depends on the following trade-off analysis. If the maximum time allowed for the resolution of the architecture is greater than 300 milliseconds and the number of components is not greater than 10, our proposal is valid for these scenarios. However, if the maximum time allowed for the resolution is lower than 300 ms, the maximum number of components varies. For example, if the time required for the construction of the architectures is less than 200 ms, acceptable times can only be ensured for architectures with 6 components or less.

The obtained execution times are quite low, but it is noteworthy to remark the following two considerations: (a) once the concrete architectural model is constructed, the deployment of the corresponding components (thus rendered by the client or the server side) must be accomplished; and (b) the resolution process performed by the STAS process belongs to a methodology for adapting software architectures at run-time in which a transformation phase is executed before the STAS process. For this reason, we have optimized the STAS approach to minimize the execution times.

The results cannot assure that the obtained performance is the best one possible. For that purpose, we should implement different search algorithms or different heuristics alternatives. In our case, we can state that our approach ensures suitable results to resolve an architecture at run-time. Furthermore, we do not compare the current implementation of the STAS process to the previous trading approach presented in [30]. The reason is that the previous approach used different heuristics and a recursive algorithm with the behavior of an exhaustive search. Nevertheless, in our evaluation process, we included the execution times when our adapted A* search is configured to be executed in a greedy mode, thus demonstrating the better results of our approach.

8. Related work

The use of COTS components for building smart software architectures is one of the main elements of our proposal. In this construction, selection and evaluation processes are considered as key operations [61]. An example of work in which these processes are addressed is the Off-The-Shelf Option (OTSO) [62]. In such an approach, a hierarchical evaluation criteria analyzes the characteristics of the components based on other factors such as organizational infrastructure or the availability of libraries. In [63], the DesCOTS system proposes a methodology based on a quality model which divides up the characteristics of the components for their evaluation.

The study presented in [64] evaluated the components and ranked them in terms of performance and according to multiple criteria. In [65], the authors perform a management of dependencies between components using goal-oriented models as the basis for component selection. A proposal for selecting COTS components in large repositories is described in [66]. That approach made use of the 'integrator' concept instead of mediation or trading services. Unlike our proposal, the approaches mentioned above do not support component selection or calculation of configurations at run-time.

The trading service described in [18] forms the basis of the present research work. The paper presented a mediation process for managing COTS components and building configurations at design-time. Our approach is based on the model proposed for specifying COTS components, but our regeneration process is designed to build architectures at run-time. Apart from [18], there are other possible approaches for the characterization of COTS components, such as the proposals described in [67] and [68]. Our proposal characterizes and validates components and architectures using MBE techniques incorporating semantic information for their run-time analysis.

In [69], the authors describe a semi-automatic process for the identification and classification of components which is based on a taxonomy and some input semantic information. The work presented in [55] also points out taxonomies and ontologies as an option to provide semantic information in the process of identifying COTS components. In our case, establishing a vocabulary of types that can be used to describe properties of the components, makes possible the construction of a classification such as a taxonomy of types or an ontology of synonyms. More recent research work is also related to the use of ontologies for querying coarse-grained components [70]. ONTOCOTS is an ontology-based recommender system which uses the Analytic Hierarchy Process (AHP) to rank COTS components when a new query process is executed. However, such a process was intended to help developers find the components for software development, and hence is not suitable for adapting architectures at run-time.

Software component reuse mechanisms can be applied to web services, since both artifacts encapsulate its implementation and expose it through interfaces [71]. In this regard, approaches related to the selection of web services can be used to improve the selection of components, for example, by considering non-functional features [72]. Furthermore, discovery mechanisms related to web services can be applied to get software elements based on functional and non-functional requirements by using a keyword as an input [73]. In the web services domain, semantic information can be used to improve the selection and discovery operations mentioned [74]. Our proposal is inspired by such approaches that adapt certain web service mechanisms to COTS component trading, for instance, simplifying the discovery of candidates using a keyword to filter components with a pertinent specification.

With regard to the construction of architectural configurations, algorithms based on heuristic functions are a suitable option for the exploration and evaluation of possible solutions [75]. Nevertheless, these types of algorithms are general-purpose searching operations, usually applied to the calculation of paths and trajectories [57]. In our case, a generic A* search algorithm [31,32] was adapted for optimizing the process and incorporating specific operations which enable the evaluation and construction of architectural solutions at run-time.

Other approaches are specifically focused on the composition and adaptation of software architectures by applying other heuristics-based search processes different from the one proposed in this article. For example, an optimization is applied in [76] to select the system architecture from a finite set of candidate components that better fulfill the required attributes. Such a process uses a mixed approach of meta-heuristics search techniques relying on the SCA-ASM service-oriented component model [77]. In this sense, the application of heuristics in these types of processes requires the specification and matching of functional and non-functional aspects for the correct calculation of the scores for each possible configuration of components [78].

The correctness of these types of search and optimization processes based on heuristics (*e.g.*, Depth First Search, Best First Search, A*, etc.) and metaheuristics (*e.g.*, Tabu Search, Variable Neighborhood Search, Guided Local Search, etc.) is accepted as a valid solution in the literature for the construction of component-based systems which requires a combinatorial analysis of their configurations [79,80]. The main reason for adapting an A* search is that it always finds a solution if one exists and keeps the best solution at all times for the execution of the algorithm. The configuration of heuristics $h'(x)$ and distance $g(x)$ is used for adapting its behavior to the specific domain of our abstract and concrete architectures. Furthermore, A* can be adapted and executed like other path-finding algorithms by changing the heuristics it uses and how it evaluates each node. For example, as we have stated in this paper, the A* algorithm can be executed as a greedy search by setting $g(x)$ as equals to 0.0 [81].

9. Conclusions and future work

This paper presented STAS, an approach for resolving software architectures at run-time by applying semantic trading. This process is responsible for building architectures represented on a concrete level (*i. e.*, architectural solutions) from architectures described on an abstract level (*i.e.*, architectural definitions) and a set of available components. Such architectures on two levels of abstraction are an appropriate way to describe different domains and scenarios and to enable their management (research questions RQ1 and RQ2). The proposed trading service manages the specifications of components and architectures and also calculates the best configurations of candidates. This calculation extends a traditional trading service to enable the construction of a software architecture from a reference specification (RQ3).

The calculation of configurations is supported by an adapted A* search algorithm. The greedy mode of this algorithm could explore the entire search space by analyzing all the combinations of candidate components. It ensures the best configuration among all the possible solutions is always found but it results in worse execution times if there is a large amount of candidate components to be combined. For this reason, our adapted A* search algorithm includes a set of operations to find the optimal solution without evaluating the entire search space. The proposed search process based on the A* algorithm enables the generation of configurations in a suitable time to build software architectures at run-time. To do this, the search space is represented by a graph and each node describes a configuration of components. As such, the optimal solution is the simple path that starts from an initial component and ends in the configuration formed by the components that best fulfill the abstract definition of the architecture (RQ5). Furthermore, the trading service makes use of semantic and syntactic information about the components to (1) select the candidates, and (2) evaluate the possible configurations with a heuristic relying on this information (R4).

The application domains of the STAS approach are smart scenarios that can be developed and deployed as coarse-grained architectures. We focus on this kind of architecture because our proposal is valid in terms of building configurations of less than twenty elements, together with the fact that smart scenarios are a good target to execute dynamic reconfigurations on their architectures. We have validated and evaluated the approach with a case study applied to the domain of mashup UIs, as an example of these types of architectures.

Some research lines remain open as future work. We plan to improve the performance of the semantic trading process in general and of the search algorithm in particular. Examples of this could be parallelizing part of the execution or incorporating new techniques of heuristics and metaheuristics to calculate the distance between components and architectures. Furthermore, we plan to develop a mechanism to facilitate the management of component types, as well as input and output types of the operations. We should develop a set of metrics related to the composition of components for those scenarios in which the establishment of a specific hierarchy is needed. In addition, execution times should be improved for domains requiring a faster response time. We plan to apply our approach to the IoT scenarios that we are developing in the field of smart cities [39,82]. Finally, the evaluation results could include an analysis based on the opinion of users to improve the validation of the approach.

CRediT authorship contribution statement

Javier Criado: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Luis Iribarne:** Formal analysis, Resources, Writing - review & editing, Supervision. **Nicolás Padilla:** Validation, Data curation, Visualization.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been funded by the EU ERDF and the Spanish Ministry of Economy and Competitiveness (MINECO) under the research project CoSmart (ref. TIN2017-83964-R), and by the regional project (ref. CEIJ-C01.2) coordinated from UAL-UCA universities and funded by the CEI-MAR consortium.

References

[1] E.F.Z. Santana, A.P. Chaves, M.A. Gerosa, F. Kon, D.S. Milojicic, Software platforms for smart cities: concepts, requirements, challenges, and a unified reference architecture, ACM Comput. Surv. 50 (2017) 1–37, https://doi.org/10.1145/3124391.
[2] E. Palomar, X. Chen, Z. Liu, S. Maharjan, J. Bowen, Component-based modelling for scalable smart city systems interoperability: a case study on integrating energy demand response systems, Sensors 16 (2016) 1810, https://doi.org/10.3390/s16111810.
[3] K.K. Lau, From formal methods to software components: back to the future?. Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) Springer Verlag, 2017, pp. 10–14, https://doi.org/10.1007/978-3-319-57666-4_2.
[4] M. Krug, F. Wiedemann, M. Gaedke, Smartcomposition: A component-based approach for creating multi-screen mashups, Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) 8541 (2014) 236–253, https://doi.org/10.1007/978-3-319-08245-5_14.
[5] J. Criado, D. Rodríguez-Gracia, L. Iribarne, N. Padilla, Toward the adaptation of component-based architectures by model transformation: behind smart user interfaces, Softw. Pract. Exp 45 (2015) 1677–1718, https://doi.org/10.1002/spe.2306.
[6] J. Perchat, M. Desertot, S. Lecomte, Component based framework to create mobile cross-platform applications. Procedia Comput. Sci., Elsevier B.V., 2013, pp. 1004–1011, https://doi.org/10.1016/j.procs.2013.06.140.
[7] M.J. O'Grady, C. Muldoon, M. Dragone, R. Tynan, G.M.P. O'Hare, Towards evolutionary ambient assisted living systems, J. Ambient Intell. Humaniz. Comput. 1 (2010) 15–29, https://doi.org/10.1007/s12652-009-0003-5.
[8] A. Bejarano, B. Fernandez, M. Jimeno, A. Salazar, P. Wightman, Towards the evolution of smart home environments: asurvey, Int. J. Autom. Smart Technol. 6 (2016) 105–136, https://doi.org/10.5875/ausmt.v6i3.1039.
[9] E. Taktak, I.B. Rodriguez, Energy consumption adaptation approach for smart buildings. Proc. IEEE/ACS Int. Conf. Comput. Syst. Appl., AICCSA, IEEE Computer Society, 2018, pp. 1370–1377, https://doi.org/10.1109/AICCSA.2017.205.
[10] A. Agirre, J. Parra, A. Armentia, E. Estévez, M. Marcos, Qos aware middleware support for dynamically reconfigurable component based iot applications, Int. J. Distrib. Sens. Networks. 12 (2016) 2702789, https://doi.org/10.1155/2016/2702789.
[11] G.S. Ramachandran, N. Matthys, W. Daniels, W. Joosen, D. Hughes, Building dynamic and dependable component-based internet-of-things applications with dawn. Proc. - 2016 19th Int. ACM SIGSOFT Symp. Component-Based Softw. Eng. CBSE 2016, Institute of Electrical and Electronics Engineers Inc., 2016, pp. 97–106, https://doi.org/10.1109/CBSE.2016.18.
[12] A. Ruppen, J. Pasquier, S. Meyer, A. Rúedlinger, A component based approach for the web of things. Proc. 6th Int. Work. Web Things, Association for Computing Machinery, New York, NY, USA, 2015, https://doi.org/10.1145/2834791.2834792.
[13] L. Médini, M. Mrissa, E.M. Khalfi, M. Terdjimi, N.L. Sommer, P. Capdepuy, J. P. Jamont, M. Occello, L. Touseau, Building a web of things with avatars: a comprehensive approach for concern management in WoT applications. Manag. Web Things Link. Real World to Web, Elsevier Inc., 2017, pp. 151–180, https://doi.org/10.1016/B978-0-12-809764-9.00007-X.
[14] C. Herring, S. Kaplan, Component-based software systems for smart environments, IEEE Pers. Commun. 7 (2000) 60–61, https://doi.org/10.1109/98.878541.
[15] D. Carney, F. Long, What do you mean by COTS? finally, a useful answer, IEEE Softw. 17 (2000) 83–86, https://doi.org/10.1109/52.841700.
[16] ISO/IEC, ITU-T, ISO/IEC 13235-1:1998 - Information technology — Open Distributed Processing — Trading function: Specification — Part 1 (1998)

[Online], Available: https://www.iso.org/standard/21470.html (accessed August 13, 2020).
[17] P. Plebani, B. Pernici, URBE: Web service retrieval based on similarity evaluation, IEEE Trans. Knowl. Data Eng. 21 (2009) 1629–1642, https://doi.org/10.1109/TKDE.2009.35.
[18] L. Iribarne, A trading service for COTS components, Comput. J. 47 (2004) 342–357, https://doi.org/10.1093/comjnl/47.3.342.
[19] OMG, services directory specification version 1.0 (2014). [Online], Available: https://www.omg.org/spec/ServD/ (accessed August 13, 2020).
[20] K. Kritikos, D. Plexousakis, Towards combined functional and non-functional semantic service discovery. Lect. Notes Comput. Sci., Springer, Cham, 2016, pp. 102–117, https://doi.org/10.1007/978-3-319-44482-6_7.
[21] P. Rodriguez-Mier, C. Pedrinaci, M. Lama, M. Mucientes, An integrated semantic web service discovery and composition framework, IEEE Trans. Serv. Comput. 9 (2016) 537–550, https://doi.org/10.1109/TSC.2015.2402679.
[22] P. Jamshidi, C. Pahl, N.C. Mendonca, J. Lewis, S. Tilkov, Microservices: the journey so far and challenges ahead, IEEE Softw. 35 (2018) 24–35, https://doi.org/10.1109/MS.2018.2141039.
[23] S. Haselböck, R. Weinreich, G. Buchgeher, Decision guidance models for microservices: service discovery and fault tolerance. Proc. Fifth Eur. Conf. Eng. Comput. Syst., Association for Computing Machinery, New York, NY, USA, 2017, https://doi.org/10.1145/3123779.3123804.
[24] K. Jander, A. Pokahr, L. Braubach, J. Kalinowski, Service discovery in megascale distributed systems, Stud. Comput. Intell 737 (2017) 273–284, https://doi.org/10.1007/978-3-319-66379-1_24.
[25] S. Capelli, P. Scandurra, A framework for early design and prototyping of service-oriented applications with design patterns, Comput. Lang. Syst. Struct. 46 (2016) 140–166, https://doi.org/10.1016/j.cl.2016.07.001.
[26] T. Vale, I. Crnkovic, E.S. De Almeida, P.A.D.M.S. Neto, Y.C. Cavalcanti, S.R.D. L. Meira, Twenty-eight years of component-based software engineering, J. Syst. Softw. 111 (2016) 128–148, https://doi.org/10.1016/j.jss.2015.09.019.
[27] J. Fitzgerald, P.G. Larsen, J. Woodcock, Foundations for model-based engineering of systems of systems, in: M. Aiguier, F. Boulanger, D. Krob, C. Marchal (Eds.), Complex Syst. Des. Manag., Springer International Publishing, Cham, 2014, pp. 1–19, https://doi.org/10.1007/978-3-319-02812-5_1.
[28] C. Rieger, H. Kuchen, A process-oriented modeling approach for graphical development of mobile business apps, Comput. Lang. Syst. Struct. 53 (2018) 43–58, https://doi.org/10.1016/j.cl.2018.01.001.
[29] J. Vallecillos, J. Criado, N. Padilla, L. Iribarne, A cloud service for COTS component-based architectures, Comput. Stand. Interfaces. 48 (2016) 198–216, https://doi.org/10.1016/j.csi.2015.11.008.
[30] J. Criado, L. Iribarne, N. Padilla, Resolving platform specific models at runtime using an MDE-based trading approach. Lect. Notes Comput. Sci. 8186, Springer, Berlin, Heidelberg, 2013, pp. 274–283, https://doi.org/10.1007/978-3-642-41033-8_36.
[31] N.J. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw-Hill Pub. Co., 1971.
[32] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (1985) 505–536, https://doi.org/10.1145/3828.3830.
[33] C. Cetina, P. Giner, J. Fons, V. Pelechano, Autonomic computing through reuse of variability models at runtime: the case of smart homes, Computer (Long Beach Calif) 42 (2009) 37–43, https://doi.org/10.1109/MC.2009.309.
[34] N. Gui, V. De Florio, T. Holvoet, Transformer: an adaptation framework supporting contextual adaptation behavior composition, Softw. Pract. Exp. 43 (2013) 937–967, https://doi.org/10.1002/spe.2137.
[35] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, J.M. Jezequel, A dynamic component model for cyber physical systems. Proc. 15th ACM SIGSOFT Symp. Compon. Based Softw. Eng., Association for Computing Machinery, New York, NY, USA, 2012, pp. 135–144, https://doi.org/10.1145/2304736.2304759.
[36] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, Computer (Long Beach Calif) 37 (2004) 46–54, https://doi.org/10.1109/MC.2004.175.
[37] J. Grundy, J. Hosking, Developing adaptable user interfaces for component-based systems, Interact. Comput. 14 (2002) 175–194, https://doi.org/10.1016/S0953-5438(01)00049-2.
[38] F. Daniel, M. Matera, Mashups: Concepts, models and architectures, Springer-Verlag Berlin Heidelberg, 2014, https://doi.org/10.1007/978-3-642-55049-2.
[39] J. Criado, J. Asensio, N. Padilla, L. Iribarne, Integrating cyber-physical systems in a component-based approach for smart homes, Sensors 18 (2018) 2156, https://doi.org/10.3390/s18072156.
[40] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, J. Vanderdonckt, A unifying reference framework for multi-target user interfaces, Interact. Comput. 15 (2003) 289–308, https://doi.org/10.1016/S0953-5438(03)00010-9.
[41] B. Selic, A systematic approach to domain-specific language design using UML. Proc. - 10th IEEE Int. Symp. Object Component-Oriented Real-Time Distrib. Comput. ISORC 2007, 2007, pp. 2–9, https://doi.org/10.1109/ISORC.2007.10.
[42] A.D. Brucker, J. Doser, Metamodel-based UML notations for domain-specific languages, in: J.M. Favre, D. Gasevic, R. Lämmel, A. Winter (Eds.), 4th Int. Work. Softw. Lang. Eng., (ATEM 2007), Nashville, USA, 2007, pp. 1–15.
[43] A. Butting, R. Heim, O. Kautz, J.O. Ringert, B. Rumpe, A. Wortmann, A Classification of dynamic reconfiguration in component and connector architecture Description Languages. 4th Int. Work. Interplay Model. Component-Based Softw. Eng. ModComp, CEUR-WS, 2017, pp. 1–7.
[44] D.D. Ruscio, I. Malavolta, H. Muccini, P. Pelliccione, A. Pierantonio, Developing next generation ADLs through MDE techniques, Int. Conf. Softw. Eng. (2010) 85–94, https://doi.org/10.1145/1806799.1806816.
[45] S. Graham, G. Daniels, D. Davis, Y. Nakamura, S. Simeonov, P. Brittenham, P. Fremantle, D. Koenig, C. Zentner, Building web services with Java: making sense of XML. SOAP, WSDL, and UDDI, SAMS publishing, 2004.
[46] M. Brambilla, J. Cabot, M. Wimmer, Model-driven software engineering in practice, Morgan & Claypool Publishers LLC (2012), https://doi.org/10.2200/s00441ed1v01y201208swe001.
[47] J. Cabot, M. Gogolla, Object constraint language (OCL): a definitive guide. Lect. Notes Comput. Sci. 7320, Springer, Berlin, Heidelberg, 2012, pp. 58–90, https://doi.org/10.1007/978-3-642-30982-3_3.
[48] D. Rodríguez-Gracia, J. Criado, L. Iribarne, N. Padilla, C. Vicente-Chicote, Runtime adaptation of architectural models: an approach for adapting user interfaces. Lect. Notes Comput. Sci. 7602, Springer, Berlin, Heidelberg, 2012, pp. 16–30, https://doi.org/10.1007/978-3-642-33609-6_4.
[49] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A model transformation tool, Sci. Comput. Program. 72 (2008) 31–39, https://doi.org/10.1016/j.scico.2007.08.002.
[50] Y. Hoffner, A. Schade, Co-operation,contracts,contractual match-making and binding. Proc. - IEEE Int. Enterp. Distrib. Object Comput. Work. EDOCW, Institute of Electrical and Electronics Engineers Inc., 1998, pp. 75–86, https://doi.org/10.1109/EDOC.1998.723244.
[51] M. Merz, K. Mueller, W. Lamersdorf, Service trading and mediation in distributed computing systems. Proc. - Int. Conf. Distrib. Comput. Syst., IEEE, 1994, pp. 450–457, https://doi.org/10.1109/icdcs.1994.302452.
[52] L. Iribarne, J.M. Troya, A. Vallecillo, Selecting software components with multiple interfaces. Conf. Proc. EUROMICRO, 2002, pp. 26–32, https://doi.org/10.1109/EURMIC.2002.1046129.
[53] L. Chung, K. Cooper, Matching, ranking, and selecting components: aCOTS-aware requirements engineering and software architecting approach. Int. Work. Model. Process. Eval. COTS Components (MPEC 2004)" W7S Work. - 26th Int. Conf. Softw. Eng., 2004, pp. 41–44, https://doi.org/10.1049/ic:20040432.
[54] J. Criado, S. Martínez-Fernández, D. Ameller, L. Iribarne, N. Padilla, A. Jedlitschka, Quality-aware architectural model transformations in adaptive mashups user interfaces, Fundam. Informaticae 162 (2018) 283–309, https://doi.org/10.3233/FI-2018-1726.
[55] A. Cechich, A. Réquilé-Romanczuk, J. Aguirre, J.M. Luzuriaga, Trends on COTS component identification. Proc. - Fifth Int. Conf. Commer. (COTS)-Based Softw. Syst., 2006, pp. 90–99, https://doi.org/10.1109/ICCBSS.2006.31.
[56] X. Li, Y. Fan, F. Jiang, A classification of service composition mismatches to support service mediation. Proc. 6th Int. Conf. Grid Coop. Comput. GCC 2007, 2007, pp. 315–321, https://doi.org/10.1109/GCC.2007.1.
[57] F. Duchon, A. Babinec, M. Kajan, P. Beno, M. Florek, T. Fico, L. Jurišica, Path planning with modified a star algorithm for a mobile robot. Procedia Eng., Elsevier Ltd, 2014, pp. 59–69, https://doi.org/10.1016/j.proeng.2014.12.098.
[58] S. Newman, Building microservices: designing fine-grained systems, O'Reilly Media, Inc., 2015.
[59] C. Pautasso, RESTful web services: Principles, patterns, emerging technologies. Web Serv. Found., Springer New York, 2014, pp. 31–51, https://doi.org/10.1007/978-1-4614-7518-7_2.
[60] M. Macero, Learn microservices with spring boot, Apress, 2017, https://doi.org/10.1007/978-1-4842-3165-4.
[61] A. Mohamed, G. Ruhe, A. Eberlein, COTS selection: past, present, and future. Proc. Int. Symp. Work. Eng. Comput. Based Syst., 2007, pp. 103–112, https://doi.org/10.1109/ECBS.2007.28.
[62] J. Kontio, G. Caldiera, V.R. Basili, Defining factors, goals and criteria for reusable component evaluation. Proc. 1996 Conf. Cent. Adv. Stud. Collab. Res., CASCON'96, IBM Press, 1996, pp. 21–32.
[63] G. Grau, J.P. Carvallo, X. Franch, C. Quer, DesCOTS: a software system for selecting COTS components. Conf. Proc. EUROMICRO, 2004, pp. 118–126, https://doi.org/10.1109/eurmic.2004.1333363.
[64] H.J. Shyur, COTS Evaluation using modified TOPSIS and ANP, Appl. Math. Comput. 177 (2006) 251–259, https://doi.org/10.1016/j.amc.2005.11.006.
[65] X. Franch, N.A.M. Maiden, Modelling component dependencies to Inform Their Selection. Lect. Notes Comput. Sci. 2580, Springer Verlag, 2003, pp. 81–91, https://doi.org/10.1007/3-540-36465-x_8.
[66] J. Clark, C. Clarke, S. De Panfilis, G. Granatella, P. Predonzani, A. Sillitti, G. Succi, T. Vernazza, Selecting components in large COTS repositories, J. Syst. Softw. 73 (2004) 323–331, https://doi.org/10.1016/j.jss.2003.09.019.
[67] M. Morisio, M. Torchiano, Definition and Classification of COTS: A Proposal, in: Lect. Notes Comput. Sci. 2255, Springer Verlag, 2002, pp. 165–175, https://doi.org/10.1007/3-540-45588-4_16.
[68] S.B. Sassi, L.L. Jilani, H.H.B. Ghezala, COTS characterization model in a COTS-based development environment. Tenth Asia-Pacific Softw. Eng. Conf. 2003, 2003, pp. 352–361, https://doi.org/10.1109/APSEC.2003.1254389.
[69] M. Sjachyn, L. Beus-Dukic, Semantic component selection - semaCS, Proc. - Fifth Int. Conf. Commer. (COTS)-Based Softw. Syst. (2006) 83–89, https://doi.org/10.1109/ICCBSS.2006.25.
[70] N. Yanes, S.B. Sassi, H. Hajjami, B. Ghezala, Ontology-based recommender system for COTS components, J. Syst. Softw. 132 (2017) 283–297, https://doi.org/10.1016/j.jss.2017.07.031.
[71] Y. Yu, J. Lu, J. Fernandez-Ramil, P. Yuan, Comparing Web services with other software components. IEEE Int. Conf. Web Serv. (ICWS 2007), 2007, pp. 388–397, https://doi.org/10.1109/ICWS.2007.64.
[72] Z. Zheng, H. Ma, M.R. Lyu, I. King, Qos-aware web service recommendation by collaborative filtering, IEEE Trans. Serv. Comput. 4 (2011) 140–152, https://doi.org/10.1109/TSC.2010.52.
[73] J. Ma, Q.Z. Sheng, K. Liao, Y. Zhang, A.H.H. Ngu, WS-finder: a framework for similarity search of web services, in: C. Liu, H. Ludwig, F. Toumani, Q. Yu (Eds.),

Lect. Notes Comput. Sci., Springer Berlin Heidelberg, 2012, pp. 313–327, https://doi.org/10.1007/978-3-642-34321-6_21.

[74] J. Chen, Z. Feng, S. Chen, K. Huang, W. Tan, J. Zhang, A novel lifecycle framework for semantic web service annotation assessment and optimization. 2015 IEEE Int. Conf. Web Serv., 2015, pp. 361–368, https://doi.org/10.1109/ICWS.2015.55.

[75] R.E. Korf, Real-time heuristic search, Artif. Intell. 42 (1990) 189–211, https://doi.org/10.1016/0004-3702(90)90054-4.

[76] R. Mirandola, P. Potena, P. Scandurra, Adaptation space exploration for service-oriented applications, Sci. Comput. Program. 80 (2014) 356–384, https://doi.org/10.1016/j.scico.2013.09.017.

[77] E. Riccobene, P. Potena, P. Scandurra, Reliability Prediction for Service Component Architectures with the SCA-ASM Component Model. 2012 38th Euromicro Conf. Softw. Eng. Adv. Appl., 2012, pp. 125–132, https://doi.org/10.1109/SEAA.2012.53.

[78] V. Cortellessa, R. Mirandola, P. Potena, Managing the evolution of a software architecture at minimal cost under performance and reliability constraints, Sci. Comput. Program. 98 (2015) 439–463, https://doi.org/10.1016/j.scico.2014.06.001.

[79] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison, ACM Comput. Surv. 35 (2003) 268–308, https://doi.org/10.1145/937503.937505.

[80] F. Rosenberg, M.B. Múller, P. Leitner, A. Michlmayr, A. Bouguettaya, S. Dustdar, Metaheuristic optimization of large-scale QoS-aware service compositions. 2010 IEEE Int. Conf. Serv. Comput., 2010, pp. 97–104, https://doi.org/10.1109/SCC.2010.58.

[81] A.R. Soltani, H. Tawfik, J.Y. Goulermas, T. Fernando, Path planning in construction sites: performance evaluation of the dijkstra, A*, and GA search algorithms, Adv. Eng. Informatics 16 (2002) 291–303, https://doi.org/10.1016/S1474-0346(03)00018-1.

[82] J.A. Asensio, J. Criado, N. Padilla, L. Iribarne, Emulating home automation installations through component-based web technology, Futur. Gener. Comp. Syst. 93 (2019) 777–791, https://doi.org/10.1016/j.future.2017.09.062.

Javier Criado is an Assistant Professor at the Department of Informatics, University of Almería (Spain). In 2009, he joined the Applied Computing Group (TIC-211. Since then, he has participated in four national research projects (refs. TIN2007-61497, TIN2010-15588, TIN2013-41576-R and TIN2017-83964-R) and a regional research project (ref. P10-TIC6114). From 2011–2015, he was supported by an FPU grant (ref. AP2010-3259). He received his Ph.D. (2015) in Computer Science from the University of Almería. His research interests include: Model-Based Engineering, Component-Based Software Engineering, Model Transformations, Model-Driven Development for User Interfaces, COTS components, Trading, Ontology-Driven Engineering, Internet of Things and the Web of Things.

Luis Iribarne is an Associate Professor at the Department of Informatics, University of Almería (UAL), Spain. He received the Ph.D. degree in Computer Science from the UAL. From 1991 to 1993, he worked as a Lecturer at the University of Granada, and collaborated as IT Service Analyst at the University School of Almería. Since 1993, he has served as a Lecturer at the UAL and worked in several national and international research projects. In 2007, he has founded the Applied Computing Group (ACG). His main research interests include simulation and modeling, model-driven engineering, machine learning, and software technologies and engineering.

Nicolás Padilla received his Ph.D. degree in Computer Science from the University of Almería. From 1991 to 1993 he served as Associate Professor at the University of Granada. Today, he serves as a Professor in the University of Almería. He has participated as a researcher in different national research projects (refs. TIN2006-06698, TIN2007-61497, TRA2009-0309, TIN2010-15588, TIN2013-41576-R, and TIN2017-83964-R) and a regional research project (ref. P10-TIC-6114) since 2006. His research interests include Model-Driven Engineering (MDE), Component-based Software Engineering (CSE), Smart Cities, Home Automation and Digital Home.