IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SURVIVABILITY STRATEGIES FOR EMERGING WIRELESS NETWORKS

Received April 18, 2018, accepted May 13, 2018, date of publication May 31, 2018, date of current version July 6, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2842683

Improved Adaptation and Survivability via Dynamic Service Composition of Ubiquitous Computing Middleware

PAOLO BELLAVISTA[1], (Senior Member, IEEE), ANTONIO CORRADI[1], (Member, IEEE), LUCA FOSCHINI[1], (Member, IEEE), AND STEFANO MONTI[2]

[1]Department of Computer Science and Engineering, Università di Bologna, 40136 Bologna, Italy
[2]Imola Informatica SpA, 40026 Imola BO, Italy

Corresponding author: Luca Foschini (luca.foschini@unibo.it)

This work was supported by the Smart Architecture for Cultural Heritage in Emilia Romagna Project funded by the POR-FESR 2014-20 through CIRI under Grant J32I16000120009.

ABSTRACT These days, ubiquitous computing has radically changed the way users access and interact with services and content on the Internet: novel smart mobile devices and broadband wireless communication channels allow users to seamlessly access them anytime and anywhere. Middleware infrastructures to support ubiquitous computing need to support an extremely dynamic and ever-changing scenario, where novel contents/services, devices, formats, and media channels become available. Service-oriented architectures and service composition techniques have proven to be the key in designing flexible and extensible platforms that are able to reliably support ubiquitous computing. However, current trends in service composition for ubiquitous computing tend to be either too formal and, therefore, poorly used by average final users, or too vertical and poorly flexible and extensible. This paper proposes novel service composition middleware for ubiquitous computing that relies on a translucent composition model to achieve a flexible, extensible, highly-available, but also easily understandable and usable platform. The proposed system has been widely tested, benchmarked, and deployed on a number of different and heterogeneous ubiquitous scenarios.

INDEX TERMS Middleware, ubiquitous computing, context-aware services, service-oriented computing.

I. INTRODUCTION

In the last decade, ubiquitous computing has dramatically changed the way users exploit services and contents available on the Internet [1]. Novel smart mobile devices, advances in wired and wireless communication channels, and the proliferation of new, more dynamic, and user-centric services and contents (from social networks to tactile Internet and multisensory human bound communications), more and more push users access contents and services at any time, with any kind of devices, and via any (either wired or wireless) connectivity type.

This scenario poses non-trivial architecture issues in terms of heterogeneity, dynamicity, and reliability. Middleware platforms to support users in ubiquitous content/service exploitation need to face a complex and ever-changing landscape of services, devices, and contents, with the goal to merge and integrate them into a comprehensive solution. They should be able to (self-)adapt in order to grant continuity and survivability of service provision under harsh conditions, notwithstanding possible wireless medium idiosyncrasies, intermittent disconnections, and abrupt changes in the communication infrastructure.

At the same time, in the last decade, Service-Oriented Computing (SOC) has gained momentum as a flexible, modular, and powerful architectural approach. SOC allows to tame large complex software systems by decomposing them into a manageable ecosystem of well-defined cooperating services.

Service composition techniques push the service-oriented approach further to allow easily aggregating and orchestrating services into larger and more complex components of business logic, hence promoting modularization and software reuse. Thus, SOC and service composition techniques have become founding approaches to cope with increasingly dynamic and heterogeneous situations of typical ubiquitous systems [2], [3].

Two main trends have emerged in service composition for ubiquitous computing. Some proposals rely on *formal approaches* to neatly model and verify service composition platforms [4]–[6]. These approaches, while powerful and general-purpose, tend to be extremely complex and hardly usable by final users. Other proposals, on the contrary, rely on *large-scale*, *user-friendly tools* (of different nature, such as web-oriented, graphically appealing service mashup platforms) [7]–[10]. They are easily usable by average end users, but frequently tend to be tailored to specific limited scenarios and hardly extensible and reusable.

To overcome above limitations, we propose a novel proposal that exhibits several original characteristics.

- It leverages a translucent approach that makes visible to final users only needed details and hides details not to propose too complex scenarios.
- It proposes a model for adaptive ubiquitous service composition that provides a user friendly, but also flexible and extensible, abstraction to describe service features and user requirements in terms of aggregation and service compositions.
- It presents a real middleware platform implementation that is also able to grant survivability of service provisioning by dynamically and continuously (re-)adapting to possible abrupt changes in the service delivery context.
- It assesses and evaluates the performance of the implemented prototype under typical ubiquitous scenarios and operating conditions, thus allowing us to automatically distill the main assumptions and simplifications to realize an effective service composition implementation, easy-to-use also by non-expert users.

II. RELATED WORK

SOC strongly promotes aggregation and reuse of soft-ware artifacts (services) to increase modularity and flexibility of distributed systems [11].

Modularity and flexibility could come at the expense of reliability: unmanaged/unplanned failures in one single service may disrupt the overall service composition, if the composition has not been designed to be intrinsically resilient.

The following subsections discuss the main research trends in both service composition and service survivability/resilience.

A. SERVICE COMPOSITION

In this subsection, without pretension of completeness, we present a selection of works that, similarly to our proposal, aim to address service composition as a way to put together existing services to realize novel value-added service aggregates in ubiquitous computing scenarios.

The extremely vast and heterogeneous landscape of service composition proposes several different approaches and proposals that target extremely different scenarios.

Early service composition platforms focused on rather static scenarios (especially Enterprise Application Integration [12]) that required to coordinate a (usually limited) number of services in a well-defined and de-terministic way. Seminal proposals therefore aimed at providing methods and tools to clearly define static and immutable compositions of services by explicitly ex-pressing how services had to cooperate, e.g., the order in which they needed to be invoked and all involved oper-ational parameters (e.g., input/output). BPEL4WS [13] is one of the most widespread standards for service composition and proposes an XML-based grammar to define compositions of Web Services.

However, this kind of approach has proven to be very limited, especially in ubiquitous and pervasive computing scenarios, for some compelling reasons [3]. The first crucial one is that designing a service composition in such a way is typically a completely user-dependent process. This obviously requires composition designers to possess a wide and high-level expertise in both the application domain the task relates to, and in the formal grammar used to express the composition. The second problem with early static approaches is the fact that they inherently fall short in more dynamic scenarios. Indeed, the initial set of available services may vary in time (by either growing or shrinking), an exact match between a specific subtask and a concrete service may not be available, and the overall final task cannot be ex-pressed in a precise and unambiguous way, either because the final service composition user has little expertise of the application domain, or because the requirements themselves are unclear.

Many different approaches tried to face these issues in such dynamic scenarios; basically, two main tendencies outstand and sometimes even coexist.

The adoption of semantic descriptions allows to capture service and service composition features that go beyond traditional basic operational features (such as input/output parameters) and provides a higher-level description of both requirements the composition need to fulfill, and service features such as behavior and interoperability constraints. WSDL-S [14] and OWL-S [15] are two of the most notable XML-based proposals in the field of semantic metadata service description and enforcement. Along this direction, various works adopt a semantic approach to provide users with richer and more detailed descriptions of services [2]–[4]. This has the obvious benefit of being clear and neat to inexperienced users. However, a richer service description also allows capturing details such as what a service is able to do rather than how it does it. That represent a crucial aspect in ubiquitous computing environments to automate (by the use of inference) compositions of suitable services each time no clear solution is evidently achievable [3], [10].

Other proposals aim at providing much more theoretical formal service composition models not only to describe service compositions but also to help reasoning on them, for instance to detect inconsistencies and possible deadlock conditions, and to infer novel and better compositions from previous ones. Typical approaches that fall in this category model service compositions by means of Petri Nets [16] or of some variants of process algebras (e.g., Calculus of

Communicating Systems [17] and Calculus of Sequential Processes [18]). Other approaches [19] define semantics in terms of a first-order logic, namely the situation calculus [20] and, based on that semantics, they describe service compositions by means of a Petri Nets model. Formal approaches, such as Petri Nets and first-order logic-based ones, have proven to be extremely powerful, especially when it comes to reason on a certain application domain and set of service compositions. Some models are able to determine whether a composition not only satisfies initial requirements but also if it is correct and provide no deadlock conditions and unreachable states. Other models allow to automatically infer novel service compositions from existing ones in order, for instance, to provide optimized compositions (e.g., service composition with equivalent overall behavior but with less services involved) and alternative versions [5], [6].

Another interesting trend in service composition directly relates to the emerging Web Mashup scenarios: users more and more are provided with Web-enabled user-friendly appealing tools to aggregate contents over the Web [7]. Several industrial mashup support tools are available since many years, such as Yahoo Pipes, Intel Mashmaker, IBM Mashup Center, and Google Maps-based mashups. These tools let non-expert users participate in the process of content creation and aggregation, by helping and guiding them throughout such a non-trivial task. However, these solutions are typically vertical and ad-hoc: allowed contents and services are usually Web pages (typically XML-based formats such as RSS) and users can exploit such contents basically by means of the sole Web browser. Some seminal academic research efforts have been aimed to ad-dress these limitations by exploiting the SOC model as a promising way to extending and broadening mashup platform support [8], [9]. However, fast computation of service mashup compositions in heterogeneous ubiquitous scenarios is still widely recognized as a challenging open issue [10], [21].

B. SERVICE SURVIVABILITY/RESILIENCE

From a research perspective, resilient/survivable service composition and adaptation is a relatively unaddressed area.

Recently, [22], [23], and [24] proposed a service composition model specifically targeted at highly dynamic, QoS-oriented service environments.

Reference [25] proposed a service indexing and search methodology specifically aimed at quickly identifying and recovering services and service compositions in case of outages.

A few years ago, [26] surveyed services and mechanisms designed for the protection of service-oriented architectures.

From a business perspective, on the other hand, service resilience seems to be a more investigated topic. Current hype on MicroServices Architectures (MSA), namely, the tendency to decompose traditional monolith applications into a set of loosely coupled, finer-grained cooperating services, pushes service survivability and requirements for resilience to the extremes.

In highly scaled, densely populated, (micro-)services ecosystems that fuel modern online services and applications (e.g., Netflix), even a slight increase in service latency can cause cascading disruptions [27].

Modern architectural approaches and tools (e.g., Hystrix [28]) are designed to embrace failure from service inception, and, for instance, promote the adoption of circuit breakers, namely, tools to automatically temporarily disable (latent and/or failing) parts of the service composition while maintaining (large portions of) the overall composition still operational.

III. DESIGN PRINCIPLES AND COMPOSITION MODEL

Current ubiquitous scenarios are more and more providing large-scale mass-market services, contents, and devices that reach an ever-increasing number of average end users [1], [3]. On the one hand, these scenarios call for support platforms that should be extremely user-friendly, and easily and intuitively usable even by average and non-skilled users. On the other hand, they need to cope with extremely heterogeneous, dynamic, and diverse situations, such as (not-so-infrequent) discontinuities of the wireless access due to lack of coverage or failures, hence calling for the ability to dynamically adapt to extend and support novel and unforeseen contents/services, user devices, and communication channels.

In our opinion, both user friendliness and extensibility/flexibility are crucial in the realization and diffusion of a ubiquitous service-oriented support system, to let non-skilled end users intuitively select, aggregate, and interact with services according to their needs.

We claim a middleware approach to help designing support platforms that manage such heterogeneous situations in a flexible and extensible way. However, we claim that a translucent approach is the key to grant our service composition middleware both user friendliness and extension/flexibility. The translucent approach makes possible, at the same time, for our proposal to hide unnecessary intricacies to non-experienced users, and to remain flexible and open, to conveniently deal with complex, unpredicted, and heterogeneous situations.

Our translucent semantic middleware clearly separates semantics into different layers at different abstraction levels and let average end users access only higher-level ones. Composition templates are at the heart of our model: they provide users with abstract, high-level, and extremely intuitive composition schemas that can be easily used to require service compositions. Ideally, in fact, users should only specify which kind of services they are interested in (service metadata) and how to arrange them (templates). Lower-level semantic items of our model, then, take care of translating user requirements into concrete operational items of the platform (e.g., services and aggregates of services) that provide users with the required scenarios. From these design guidelines we distilled our middleware architecture that consists of three main layers (see Fig. 1).

The bottom business logic layer includes services and workflows: services are the basic building blocks of our

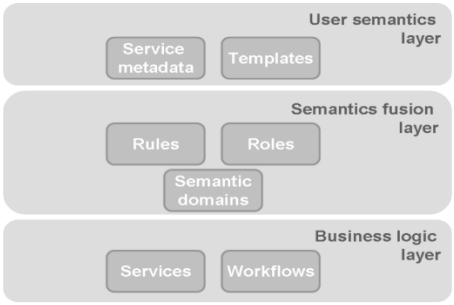

FIGURE 1. Translucent semantic model.

application system and workflow [29] is the concrete means to make them cooperate. Workflows describe structured activities and their complexity can range from simple sequences of services activated after one another, to complex compositions of both services and control blocks, such as conditional branches, forks, joins and so on.

The top user semantics layer includes service metadata to convey high-level information about semantic features of services, e.g., their typology (content generation and retrieval, transcoding, etc...) and QoS-related aspects (such as average computational load and cost) used to drive service choice for users. In addition, the templates model abstract flows of activities, i.e., flow definitions whose nodes need to be (either partially or fully) filled in with concrete business logic. In other words, templates are a suitable abstraction to help users in sketching out arrangements of services (so to express service coordination logic).

Finally, the middle semantics fusion layer provides features that allow to translate abstract templates into workflows. Rules express constraints over the pieces of business logic that participate in the realization of a template, whereas roles allow to express such constraints not on a specific business logic element (e.g., a service or a workflow node) but to abstract counterparts, and to share and reuse them across different elements of the template. Finally, semantic domains convey a useful way to partition semantic features into different spaces, so to avoid a fixed and immutable semantic knowledge base, but rather to permit insertion of novel semantic concepts while still keeping older ones consistent.

Following subsections better detail each of these layers and their inner main components; we will use a bottom up presentation approach.

A. BUSINESS LOGIC LAYER

Business logic layer provides the low-level facilities that concretely realize application scenarios. Entities of this layer could be completely invisible to average final users: it is up to our middleware to concretely manage business logic implementation details to realize user requirements. Nevertheless, coherently with our translucent approach, skilled users are still allowed to manually arrange services into concrete workflows, by taking care of directly connecting services with each other, as in more traditional static approaches.

1) SERVICES

Following the SOC paradigm, we model pieces of application logic as services that can be plugged in by need to extend middleware ubiquitous features support. Hence, support for novel content types as well as novel formats (and consequent adaptation/transcoding logic) and novel user interaction channels can be easily added by simply adding new services.

Definition 1: We define S *as the set of all available services in the system.* S *may grow in time but it always contains a finite set of elements.*

2) WORKFLOWS

In traditional SOC approaches, aggregation and coordination of services help realizing more complex value-added application scenarios out of basic building blocks, thus promoting business logic reuse and modularity. Workflows can range from simple sequences of services to more complex aggregates with conditional branches, fork/join nodes and so on. Managing execution of logic entails concretely invoking services after one another; hence, workflows are in charge of tasks such as parameter passing between subsequent stages and exception handling.

Definition 2: We model workflows as directed graphs consisting of nodes and links $\boldsymbol{WF} := (\boldsymbol{WFN}, \boldsymbol{WFL})$. *Workflow nodes (WFN) can be either concrete services or control blocks (e.g., fork, join, and conditional nodes). Workflow links (WFL) are directed connections that interconnect two workflow nodes.*

Services and workflows are concrete entities of the system in charge of realizing user-driven ubiquitous scenarios. Once identified, workflows and services need no semantic interpretation. On the contrary, semantics is used to decide whether a given (more or less formal) description of requirements can be satisfied and translated into a concrete workflow of services.

B. SEMANTICS FUSION LAYER

The semantics fusion layer realizes the glue that helps translating high level user requirements into concrete workflows of available services.

Rules are at the heart of that translation and basically represent constraints on some features of the services that can participate in a composition. Semantic service features may be any kind of relevant piece of information for a given scenario. Since providing a priori a complete, extensive, and monolithic description of semantic features is simply unrealistic for extremely dynamic and mutable scenarios such as the ubiquitous computing ones, we break down "homogeneous" semantic pieces of information and group them into semantic domains. Finally, rules in our model can either refer to a single concrete item of the system (e.g., a given service), or relate to groups of items. Since the latter case may require a tedious and error-prone association of the same rule to different items, we introduced the notion of role. Roles allow to group items and easily express rules on all of them with one action,

providing a more compact and manageable way to express groups of rules. Suppose for instance users require to access different data sources (whose number may not be defined a priori), each one providing HTML content; our platform may more conveniently provide a rule stating that "each service willing to play the role of *datasource* should provide *HTML* content", instead of explicitly specifying for each datasource service something like "service X should provide *HTML* content".

1) SEMANTIC DOMAIN

A number of different proposals exist to specify semantic information on services; some approaches are extremely tailored to specific areas of interest and application domains, whereas other proposals aim at giving generic purpose models and languages to describe any kind of semantic feature. As an example, Web Service Semantics [14] and OWL-S [15] promote standard XML formats to describe semantics.

Our model does not rely on a specific service semantic description, but rather can be used by any standard, thus improving flexibility and reuse. Furthermore, a monolithic and predetermined set of semantic notions does not fit well with intrinsically dynamic scenarios where semantic description itself may need to grow and adapt to ever-changing scenarios.

We therefore prefer providing our system with a way to conveniently add novel semantic information and make it coexist with already existing concepts. To cope with such intrinsic heterogeneity and openness, we propose the notion of semantic domains to conveniently group semantic information based on, for instance, metadata area of interest and even metadata format. As an example, we distinguish metadata related to service quality rather than binding features. Novel semantic domains can be introduced to capture novel aspects and give novel and different interpretations to pieces of business logic.

Definition 3: We define $\boldsymbol{D}$ *as the set of available semantic domains. Each domain can specify and contain semantic attributes (i.e., named properties that describe specific features) and values related to such attributes. We define* $\boldsymbol{A_d}$ *the set of available semantic attributes over semantic domain d and* $\boldsymbol{V_{ad}}$ *is the set of available semantic values for semantic attribute a of domain d.*

As an example, assumed the Syntax semantic domain, some of its possible attributes are (from now on, examples will be reported using a different font):

$$A_{syntax} = \{input, output\}$$

and possible values for attribute input could be:

$$V_{input,syntax} = \{application/xml, text/plain, \ldots\}.$$

Another semantic domain characterizes QoS attributes. Typical attributes for the QoS semantic domain are:

$$A_{QoS} = \{estimated\ ComputationLoadinput,\ billing, \ldots\},$$

and possible values for attributes could be numerical values representing the average estimated computational load and the cost of the service if its use is not free-of-charge.

Semantic attributes and values can be associated to any item in our model. Associations between an element of our model and an attribute and value can be either direct or indirect.

Directly associating an attribute and value to an item means tagging it with a certain semantic meaning; as an example, we will use this approach in the following for service semantic metadata association.

Even if this is a perfectly viable approach, sometimes it is much more helpful to provide a way to express certain semantic features for an entire class (or group) of items without having to explicitly bind any one to that feature. Furthermore, sometimes it could be impossible to specify semantics for an item since it is not a concrete one but rather is an abstract item our service composition engine needs to concretely substitute with pieces of business logic.

To overcome these problems we provide *indirect attribute associations* via the notion of role.

2) ROLE

Roles allow to create classes of model elements that share common semantic features. The addition of a semantic feature (either attribute or value) to a role means that each item willing to play that role should provide the specified attribute.

Roles are a convenient way to realize indirect semantic association, hence they can be used to express semantic on elements that are not concrete yet (such as template elements).

Definition 4: We define R as the set of available roles and A^r_d *as the set of available semantic attributes of domain d for role r.*

For instance, given the content generator (*generator*) role, the attribute

$$output^{generator}_{syntax} = \{generator, syntax, output\}$$
identifies the semantic attribute output (of semantic domain syntax) for business logic willing to play the role of generator.

3) RULE

The rules are the concrete way to drive selection and arrangement of concrete services into workflows that realize user requirements. Rules are the concrete means to drive selection and arrangement of concrete services into workflows that realize user requirements.

Rules provide semantic composition constraints by comparing semantic attributes and values of a specific semantic domain for one or more pieces of business logic; hence they are used to concretely evaluate whether a real composition of services can be arranged to fulfill user requirements.

We distinguish consistency rules and scoring rules as follows.

Definition 5: Consistency rules (cr) evaluate whether a certain set of semantic attributes and values are compatible with each other. $cr := \left[A_D^R \cup V_D\right]^n \rightarrow \{0, 1\}$.

Definition 6: Scoring rules (sr) evaluate the degree of compatibility of a certain set of semantic attributes and/or values. We indicate the degree of compatibility with a real value.

$$sr := \left[A_D^R \cup V_D\right]^n \rightarrow R$$

As an example, we provide the following rules:

$$output_{syntax}^{generator} = input_{syntax}^{deliverer} \sum estimatedComputationLoad_{QoS}^{role_i}, role_i \in \{generator, transcoder, deliverer\}$$

The former one is a consistency rule that determines whether the semantic attribute *output* (of semantic domain *syntax*) of role *generator* and semantic attribute *input* of role *deliverer* are compatible; the latter one sums up values of attribute *estimatedComputationLoad* (semantic domain *QoS*) for roles *generator*, *transcoder*, and *deliverer*, in order to evaluate the overall computational cost for each piece of business logic that plays one of the aforementioned roles.

C. USER SEMANTICS LAYER

User semantics layer provides facilities that can easily assist users in choosing the right services, in arranging them, and in deciding how to exploit them.

User friendliness here stems from the fact that users are only asked to choose:

- which *kind of services* they are interested in (service metadata);
- how to *aggregate* them (templates).

It is then up to our composition engine to decide whether these requirements can be satisfied, given the available services, and to automatically build compositions of services that can correctly and consistently cooperate, from both an operational and a semantic point of view.

1) SERVICE METADATA

Services represent atomic pieces of business logic related to content production, transcoding, adaptation, and so on, and are described by means of semantic *service metadata*, to express both low-level grounding connection features and high level semantic information.

Definition 7: Given $\boldsymbol{S}$ *the set of available services, we define the service metadata property as:*

$$p_{d,a}^S = (s, a_d, v_{ad}) \ \ where\ s \in S, \quad a_d \in A_d, \ v_{ad} \in V_{ad}$$

The service metadata property (or simply the property) is the value v_{ad} *of semantic attribute a over semantic domain d for service s. Similarly,* P_d^s *denotes the set of properties of service s on semantic domain d and* P^s *the set of properties of service s.*

In a typical example, metadata for a text-to-speech (tts) synthesis service may express the provided bit-rate synthesis as shown in Fig. 2.

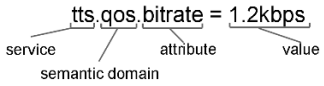

FIGURE 2. Semantic metadata property example.

2) TEMPLATE

Templates represent abstract workflow schemas our platform should fill in with concrete business logic (e.g., services) to satisfy user needs. Templates are modeled as directed graphs and are made up of nodes that can represent either concrete service logic or abstract *placeholders* with some semantics associated.

By adopting a graph-based description, we are able to easily and graphically convey information of what a template does to final users; in fact, graph-based representations easily allow users to perceive the flow of control between subsequent stages of a complex aggregate of business logic. Not surprisingly, intuitive and user-oriented Web 2.0 mashup tools such as Yahoo Pipes exploit the same approach and provide a *drag-and-drop* graphical interface that allows to arrange blocks (services) into more or less complex graphs.

Definition 8: We define the set ofavailable template nodes as $N := S \cup CB \cup PL \cup T$. *Thus, each node in a template can be a concrete service (S), a control block (CB), a placeholder (PL), and a template (T) in its turn.*

Node definition makes templates inherently recursive: each template node may for instance hold atomic concrete logic as well as other sub-templates (that aggregate and put together atomic services).

Definition 9: Nodes are connected by links that represent directed connections between two nodes. We define $L := (N \times N)$ *as the set of links connecting available nodes.*

Control blocks (CB set) are nodes expressing forks, joins, conditions, and so on, and they are typically used to manage and control the flow of execution among successive stages.

Placeholders(PH set) are the key elements in templates because they represent the abstract nodes our platform must substitute with concrete business logic in order to fulfill user requirements. In order to do so, we typically impose consistency rules on placeholders, thus expressing semantic constraints on the concrete business logic that will replace placeholders. Typically, consistency rules may involve different placeholders and can be also shared and reused for different sets of placeholders in the same template. A typical example would be a rule to constrain each service willing to replace any of the placeholders to have a computation load (e.g., estimatedComputationLoad semantic attribute) below a certain threshold value. Indirect semantic association by means of roles is a straightforward method to avoid specifying such a rule for each placeholder.

As a consequence, we provide a way to explicitly associate roles to placeholders, hence allowing for the sharing and reuse of rules across the template.

Definition 10: We define the set of Placeholder-Role Relations as $PRR := \{prr : PH \rightarrow R\}$, *and* PRR_p *as the set of Placeholder-Role Relations for placeholder p.*

Finally each template carries a set of rules RU that drive the process of filling placeholders by evaluating semantic attributes over declared placeholders roles.

Definition 11: We define a template $T := \{N, L, PRR, RU\}$ *Given a template* $t : N_t, L_t, PRR_t, RU_t$ *identify respectively the nodes, links, placeholder-role relations, and rules of template t.*

D. USAGE SCENARIO – USER REQUIREMENTS

In the following we will describe a typical ubiquitous content aggregation scenario from the user standpoint (see Fig. 3). User requires to gather information from different content sources (such as an RSS feed, a newsletter, and a plain HTML portal); furthermore, user requires to receive aggregated content via an SMS message on her mobile phone at a certain hour every day.

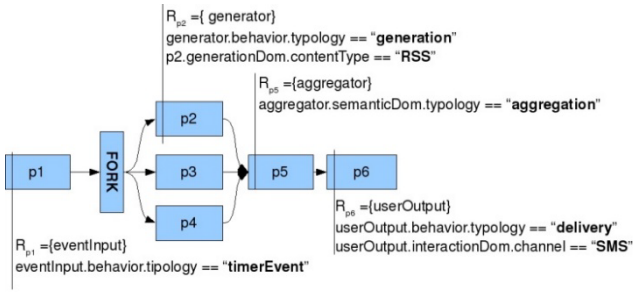

FIGURE 3. Example template and rules.

In our vision, the average end user should have no deeper technical information about her requirements and it is up to the platform to arrange available business logic components to satisfy user needs (if it is possible).

Our platform provides a 'Content Aggregation' template that features a couple of initial and final placeholders and a variable number of placeholders in between (in the following we will consider three generator nodes), each one of them playing a generator role.

This template already comes with a rule that constrains services willing to play the *generator* role to provide the value ''generation'' for semantic attribute *typology* of domain *behavior*.

User marks the initial placeholder (p1) as an eventInput node to tell the system she wants the composition be activated asynchronously by means of an event. This action brings into the template a novel rule (associated to the role eventInput) that constrains services willing to play the eventInput role to provide the value ''timerEvent'' for semantic attribute typology of domain behavior.

Similarly, she marks the final placeholder (p6) as an userOutput node to tell the system she wants the composition to send its output via an SMS message. This brings into the template a novel rule (associated to the role userOutput) that constrains services willing to play the userOutput role to provide the value ''delivery'' for semantic attribute typology of domain behavior.

By performing these simple choices, user has constrained the template to behave and interact in a well-defined way, i.e. by asynchronously reacting to an event and by notifying the user of the elaboration result via an SMS message.

Available semantic attributes over the generationDomain semantic domain relate, for instance, to content type (contentType). Users can therefore select semantic values (e.g., by means of convenient Web user interfaces) for such attributes, to impose constraints on each placeholder. Our platform therefore adds a rule to the template that forces service (or service aggregates) willing to replace node p2 to provide the semantic value ''RSS'' for attribute contentType. By following the same approach, user configures nodes p3 and p4 to produce newsletter- and HTML-related content. Note that rules that can (or need to) be shared among different placeholders (e.g., rules on the generator role) should be expressed indirectly by means of attributes over a role that marks more than one placeholder. To force placeholder-specific semantic values, we use roles specific to each placeholder (e.g., by convention, a role with the same name as the placeholder). This is the case with ''contentType'' attribute for generator nodes: each generator placeholder should feature a different value, hence a different rule, to force the platform select different kinds of contents. Finally, user requires output to be of type SMS.

The service composition layer is now in charge of deciding whether currently available services (or service aggregates) can satisfy user needs.

In a similarly simple way, the user could have required a synchronous direct pull-based interaction, for instance by configuring both input and output on an HTTP channel. Note that more skilled users may access a more sophisticated interface by means of which they can modify the template graph (e.g., by inserting and removing templates) in order, for instance to provide two alternative input and output placeholders.

IV. COMPOSITION CONSISTENCY AND SCORING

The main goal of the composition layer is to translate abstract templates into concrete workflows made up of available business logic (services): we call this process composition reification. This activity basically entails filling template placeholders with either services or sub-templates that are suitable to play the roles declared by the placeholder.

Consistency of services with user requirements is determined by evaluating all consistency rules that involve roles of the placeholder to be filled. In real deployment scenarios a large number of services are available, and some of them may provide similar business logic. We therefore expect

that, given certain user requirements, more alternative sets of services may be found to satisfy them. Even if equivalent from a functional standpoint, these alternatives will probably feature different non-functional characteristics, such as cost and Quality of Service (e.g., responsiveness and availability). Our model therefore provides a way to evaluate different alternative solutions and to rank them for different criteria via scoring rules.

Composition scoring allows to realize a wide variety of strategies to satisfy both user needs and platform performance requirements; for instance, it allows to provide users either with the ''best'' solution out of all possible ones or the first and possibly non-optimal one the platform is able to find. ''Best'' here can be interpreted in its broadest sense: some users, in fact, may require the most responsive solutions, whereas others may require the least expensive ones (also a balance of the two policies); moreover, under specific high load conditions, our system may decide to provide users with the most lightweight solutions (even if less performing).

In this section we will describe the evaluation models for both the consistency phase (to determine whether a given set of services satisfy consistency rules) and the scoring phase (to determine ''how well'' a given set of services satisfy user requirements).

A. EVALUATION FACILITIES

This section describes basic tasks at the heart of the template reification process, namely service substitution, consistency, and scoring evaluation.

1) CONSISTENCY EVALUATION

Consistency evaluation refers to the process of determining whether semantic values are consistent with each other for a specific meaning.

Definition 12: We define consistency as a function that compares semantic values to check whether they are consistent:

$$f_{consistency} = [V_{AD}]^n \rightarrow \{0, 1\}, \quad n \geq 2$$

The most common consistency function imposes that two or more semantic values have to be equal; nevertheless, our platform is able to deal with any kind of consistency function, thus providing a convenient way to model complex relationships. For instance, in a typical heterogeneous content format scenario, some kind of business logic (e.g., audio transcoding) can be compatible with any type of MIME audio input type (''audio/$*$'').

2) SCORING EVALUATION

Scoring evaluation refers to the process of determining the degree of consistency of semantic values with each other for a specified meaning.

Definition 13: We define scoring as a function that weighs the degree of consistency of two or more semantic values:

$$f_{score} = [V_{AD}]^n \rightarrow \Re, \quad n \geq 2$$

The most typical scoring functions compute both total and average values from a set of two or more semantic values; nevertheless, our platform is able to deal with any kind of consistency function, thus providing a convenient way to model evaluations.

3) SUBSTITUTION

Concrete services are meant to substitute placeholders in playing certain roles. Since each role may be associated with semantic attributes, the substitution function is in charge of extracting the service semantic property whose attribute matches with the one of the role. This value is then used to either concretely verify whether consistency rules are satisfied or to evaluate scoring rules

Definition 14: We define substitution as a function:

$$f_{sub} = \left[A^R \times S\right] \rightarrow V_{AD} \cup \emptyset$$

Given a service s and an attribute a_x, a substitution either returns the semantic value v_x if service declares a corresponding metadata property (s, a_x, v_x), or an empty set in case no solution is found.

B. SERVICE AND TEMPLATE RULES

Rules usually do not tie to a particular service, instead, they are expressed in terms of roles; hence roles allow to abstract and reuse rules across services. Indeed, each service willing to play a specific role must satisfy each rule that involves such roles.

The rule descriptions given in previous sections are generic: in this section we refine their definition and we identify significant subsets of both *consistency*and*scoring rules*.

Service rules bind a semantic attribute of a candidate service to a concrete semantic value; hence service rules constrain the choice of a single service.

Template rules compare semantic attributes of candidate services to semantic attributes of other candidate services; hence template rules establish relationships among different service candidates.

By following the above considerations (and by explicitly including consistency and scoring functions), we refine consistency and scoring rules as follows.

Definition 15: We define SCR as the set of service consistency rules (scr) defined as follows:

$$SCR := \left\{scr_r : \left(a_d^r, v_{ad}, f_{consistency}\right) | a_d^r \in A_d^r, v_{ad} \in V_{ad}\right\}$$

A service consistency rule scr_rtherefore binds a specific attribute of a role r to a specific semantic value. Each service willing to play role r needs to provide a semantic property whose value is consistent (by verifying the consistency function $f_{consistency}$) with v_{ad}.

Definition 16: We define TCR as the set of template consistency rules (tcr) defined as follows:

$$TCR := \left\{tcr_{r_1,\ldots,r_m} : \left(a1_d^{r_1}, \ldots, an_d^{r_m}, f_{consistency}\right) | \right.$$
$$\left. a1_d^{r_1}, \ldots, an_d^{r_m} \in A_d\right\}$$

A template consistency rule $tcr_{r_1,\ldots,r_m}$ therefore binds n attributes of m roles to each other (m<=n since more attributes of the same role can participate in the rule).

Similarly, we impose the same behaviour over scoring rules and we define service scoring rules (ssr) and template scoring rules (tsr).

Definition 17: We define SSR as the set of service scoring rules (ssr) defined as follows:

$$SSR := \left\{ssr_r : \left(a_d^r, v_{ad}, f_{score}\right) | a_d^r \in A_d^r, v_{ad} \in V_{ad}\right\}$$

A service scoring rule therefore evaluates "how well" a specific attribute of a role compares to a specific semantic value v_{ad}.

Definition 18: We define TSR as the set of template scoring rules (tsr) defined as follows:

$$TSR := \left\{tsr_{r_1,\ldots,r_m} : \left(a1_d^{r_1}, \ldots, an_d^{r_m}, f_{score}\right) | a1_d^{r_1}, \ldots, an_d^{r_m} \in A_d\right\}$$

A template scoring rule therefore evaluates a set of specific attributes.

Even though rules of our model may be extremely generic, from an operational standpoint, rules described in Definition 15-18 are the most relevant and useful ones. As a consequence we make the following operational assumption.

Assumption 1: Each template declares only service consistency, template consistency, service scoring, and template scoring rules: $\forall t \in T, RU_t \subset (SCR \cup TCR \cup SSR \cup TSR)$.

Even though our model fits complex and complete composition systems where composition rules may be any kind of constraint, from an operational standpoint we acknowledge that the most useful kinds of constraints are the ones that relate to services and templates, and allow to both verify their consistency and to score them.

C. RULE EVALUATION

In order for a placeholder to be filled with a candidate service, rules related to the placeholder roles (PRR relations) must be evaluated. Consistency rule evaluation determines whether a service (or a set of services) can play the required role(s), whereas scoring rule evaluation determines "how well" the candidate service can play the required role(s).

Definition 19: We define service rule consistency evaluation as a function that determines whether a given service can play a given role according to a given scr.

$$eval_{scr} : [SCR \times R \times S] \rightarrow \{0, 1\}$$

Specifically, given a service s, a role r, and a service consistency rule $scr_r : \left(a_d^r, v_{ad}, f_{consistency}\right)$, consistency evaluation takes place by substituting service s to the corresponding roles r in the rule, and then by applying the consistency function declared by the rule itself.

$$eval_{scr}\left(scr_r, r, s\right) = f_{consistency}\left(f_{sub}\left(a_d^r, s\right), v_{ad}\right)$$

Definition 20: We define template rule consistency evaluation as a function that determines whether a given set of services can play a given set of roles according to a given tcr.

$$eval_{tcr} : \left[TCR \times [R \times S]^n\right] \rightarrow \{0, 1\}$$

Specifically, given a template consistency rule

$$tcr_{r_1,\ldots,r_m} : \left(a1_d^{r_1}, \ldots, an_d^{r_m}, f_{consistency}\right)$$

and a set of role-service substitutions (r_j, s_k), $j \in (1, m)$, $k \in (1, p)$ evaluation takes place by substituting services to the corresponding roles in the rule, and the by applying the consistency function declared by the rule itself:

$$eval_{tcr}\left(tcr_{r_1,\ldots,r_m}, (r_1, s_1), \ldots, \left(r_m, s_p\right)\right) = f_{consistency}\left(f_{sub}\left(a1_d^{r_1}, s_1\right), \ldots, f_{sub}\left(an_d^{r_m}, s_p\right)\right).$$

Definition 21: We define service rule scoring as a function that evaluates "how well" a given service can play a role according to a given ssr.

$$score_{ssr} : [SSR \times R \times S] \rightarrow \Re$$

Specifically, given a service s, and a service scoring rule $ssr_r : \left(a_d^r, v_{ad}, f_{score}\right)$, scoring takes place by substituting service s to the corresponding role r in the rule, and then by applying the scoring function declared by the rule itself: $score_{ssr}\left(ssr_r, r, s\right) = f_{score}\left(f_{sub}\left(a_d^r, s\right), v_{ad}\right)$.

Definition 22: We define template rule scoring as a function that evaluates "how well" a given set of services can play a given set of roles according to a given tsr.$score_{tsr} : [TSR \times [R \times S]^n] \rightarrow \Re$.

Specifically, given a template scoring rule, $tsr_{r_1,\ldots,r_m} : \left(a1_d^{r_1}, \ldots, an_d^{r_m}, f_{score}\right)$, and a set of role-service substitutions $\left(r_j, s_k\right), j \in [1, m], k \in [1, p]$ scoring takes place by substituting services to the corresponding roles in the rule, and then by applying the scoring function declared by the rule itself:

$$score_{tsr}\left(tsr_{r_1,\ldots,r_m}, (r_1, s_1), \ldots, \left(r_m, s_p\right)\right) = f_{score}\left(f_{sub}\left(a1_d^{r_1}, s_1\right), \ldots, f_{sub}\left(an_d^{r_m}, s_p\right)\right).$$

D. TEMPLATE REIFICATION

We call *template reification* the process of filling each placeholder node in a template with a suitable service and, in general, of completely finding a concrete substitution for abstract elements with concrete counterparts. A reifiable template is a template whose placeholders can be substituted by at least a set of services that satisfy the following two consistency properties, namely service consistency and template consistency.

Definition 23: Service consistency requires that each service willing to replace a placeholder should satisfy all the service consistency rules associated with any one of the roles associated with the placeholder.Given a placeholder p, a template t, and a candidate service (for placeholder p) $c_p \in S$, *c_p is service-consistent for placeholder p if:*

$$\forall r | \exists\, prr_n \rightarrow \{r\}, \quad \forall scr_r \in RU_t \rightarrow \{r\}, \quad eval_{scr}\left(scr_p, r, c_p\right) = 1.$$

Definition 24: Template consistency requires that each set of services willing to replace a set of placeholders should satisfy all the template consistency rules associated with any one of the roles associated with each placeholder. Given a placeholder p, a service candidate service s_p to substitute p, and a set of other candidate services CS, s_p is template consistent in CS if: $\forall r | \exists prr_n \rightarrow \{r\}, \forall tcr_{r_1,\ldots,r_m} \in RU_t | \exists r_i = r, \exists \{s_1, \ldots, s_{m-1}\} | s_i$ *is service consistent in*

$$p_x \in PH_t \forall i, \quad eval_{tcr}\left(tcr_{r_1,\ldots,r_m}, (r_1, s_1), \ldots, (r_i, s_p), \ldots, (r_m, s_{m-1})\right) = 1.$$

Template consistency verifies that a service willing to replace a placeholder can satisfy all of the template consistency rules that involve one (or more) role of the placeholder to be replaced.

Definition 25: Given a template t and a set of candidate services CS=$\{s_1,\ldots s_n\}$, template t is reifiable in $\{s_1,\ldots s_n\}$ if: $\forall s_i \in CS$, s_i *is service consistent in* $p_i \in PH_t$, s_i *is template consistent in* $\{s_1, \ldots, s_n\}$.

Each service in $\{s_1, \ldots, s_n\}$ can therefore be used to substitute a corresponding placeholder in a way that guarantees satisfaction of all the consistency rules. So, the composition platform can build a concrete workflow out of the template by consistently replacing its abstract placeholders with existing services (set $\{s_1, \ldots, s_n\}$).

Definition 26: Given a template t that is reifiable in {s1,...sn}, we call {s1,...sn} reification set.

Each reification set therefore represents a solution that can be used to translate an abstract template into a concrete workflow; the resulting workflow is granted to be consistent with user requirements (e.g., all composition rules).

E. USAGE SCENARIO – CONSISTENCY AND SCORING

In Section III.D we showed how user selects a 'Content Aggregation' template and how user choices translate into concrete consistency rules for one template. Service composition layer now inspects available services to determine whether services exist whose semantic properties can satisfy specified rules and therefore can be used to translate the abstract template into a concrete workflow. If more solutions can be found, our platform also ranks them according to given criteria.

1) CONSISTENCY EVALUATION

To grant template consistency, one of the most typical consistency rules in our templates relates to syntactical consistency, e.g., to the ability to check whether services can interoperate in terms of basic interconnection features such as input/output parameters and pre-/post-conditions satisfaction.

Link consistency rule(*lcr*) is a specific template consistency rule that can be for example of the form: *lcr* := ((*producer*, *syntaxDom*, *output*), (*consumer*, *syntaxDom*, *input*)), where*producer* and *consumer* roles mark subsequent nodes and the above rule states that each service willing to play the producer role should declare a semantic property *output* (in the semantic domain *syntaxDom*) whose value is consistent with the value of semantic property *input* of the service willing to play the role of *consumer*. To guarantee that each preceding service output is compatible with the following service input, it is sufficient to add a link consistency rule to each couple of adjacent (i.e., connected by a link) services.

This kind of rule is associated to each template in our system, however it is completely transparent to final users (i.e., users usually do neither influence nor perceive their presence at all). This allows to grant system correctness, without requiring users to delve into the details of manually checking link consistency.

Given the template and rules shown in Section III.D, our system detects that an available RSSReader service (whose *typology* is "generation") provides the "RSS" value for attribute *contentType*. This means it can play the role *generator* and thus fill in placeholder *p2*. The same applies to placeholders *p3* and *p4* and NewsReader and HTMLReader services. By providing "aggregation" as typology attribute, the Aggregator service is suitable to fill in placeholder *p5*. Finally, the SMSSender service metadata allow SMSSender to fill in placeholder *p6*.

This allows to translate the abstract template and original user requirements into a concrete workflow of services as depicted in Fig. 4.

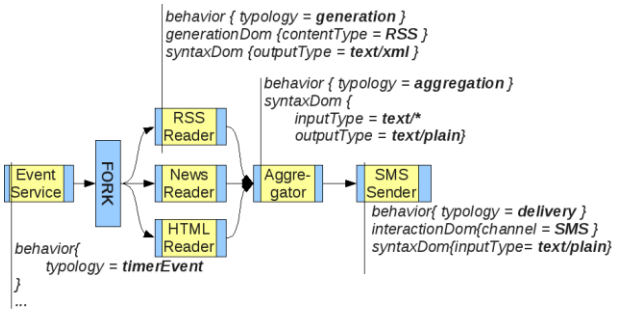

FIGURE 4. Template reification.

2) SCORING EVALUATION

Suppose no link consistency violation occurs and different alternatives exist to both the SMSSender service and the HTMLReader, with different QoS characteristics (in the example in Fig. 5, cost and responsiveness). Hence user requirements can be satisfied by different concrete workflows.

Two template scoring rules may drive the choice among these alternatives:

- a *min_cost* scoring function may sum all *cost* values for any service involved in the template;
- a *max_responsiveness* scoring function may sum all *responsiveness*values for any service involved in the template.

Some users may prefer maximum responsiveness, no matter the cost: our platform (see Fig. 6) then provides a solution

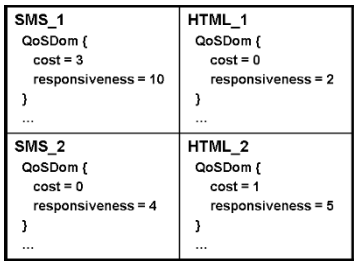

FIGURE 5. Example service alternatives.

	min_cost	max_responsiveness
..., SMS_1, HTML_1, ...	3	12
..., SMS_1, HTML_2, ...	4	15
..., SMS_2, HTML_1, ...	0	6
..., SMS_2, HTML_2, ...	1	9

FIGURE 6. Solution scoring.

that features *SMS_1* and *HTML_2*, hence achieving the maximum responsiveness (i.e., 15; we suppose services other than *SMSSender* and *HTMLReader* provide no contribution to *cost*and *responsiveness*). At the opposite, other users may prefer minimizing the costs, no matter the quality of responsiveness of the overall service: in this case, our platform provides a composition that leverages *SMS_2* and *HTML_1*. Obviously, mixed strategies may coexist to provide a tradeoff between the two alternatives.

V. IMPLEMENTATION

Our proposal aims at targeting large deployment scenarios, with lots of users and services available and with possibly complex requirements in terms of different service aggregates (templates) and requirements on them (rules). System scalability thus becomes crucial to derive from an open and flexible model an efficient concrete system implementation.

Scalability issues are mainly related to two main factors: the enlarging repertoire of services satisfying a template and the potential of vesting other templates. On the one hand, in fact, the ever-increasing number of services may lead to an increasing number of different alternative solutions (reification sets) to satisfy user requirements; thus, algorithms to evaluate all possible solutions may quickly become greedy time- and resource-consuming. On the other hand, the inherent model recursion allows to build larger and more complex templates up from other templates (e.g., by replacing placeholders with templates); that opens up the possibility to infer a (possibly) unlimited number of increasingly larger and complex template aggregates, all correctly satisfying initial user needs.

A. IMPLEMENTATION GUIDELINES

To limit the impact on scalability, we devise two main guidelines. First, we are not targeting completeness: our system may decide to find only a subset out of all potential solutions. From a user standpoint, in fact, each reification set is granted to satisfy all requirements user has expressed: hence our platform may decide (for instance, under heavy load conditions) to stop computing alternative solutions once the first suitable one (in alternative only a given number) is found and provide it to users; this may result in non-optimal solutions (e.g., solutions whose score may be worse than the one of non-calculated solutions). On the contrary, under certain ''lightweight'' operating conditions, or for some ''premium'' users, our platform may decide to find out all possible solutions.

The second aspect aims at limiting the adoption of recursion in the process of finding solutions: our implementation by default does not search for substituting placeholders (or groups of placeholders) with other templates. Recursion should be allowed only in case no reification sets can be found for the given template and user requirements. Under certain conditions, in fact, substituting placeholders of the original template with other templates may easily allow novel solutions that satisfy user requirements. With the goal of efficiency, we identified some well-known situations in which a recursive approach may effectively help finding out novel solutions without imposing much effort on computational resources and we have enabled recursion only to handle these situations. The most typical case involves link consistency check failures: suppose we have a simple (portion of a) ''sequence'' template in which service A output feeds service B input (see Fig. 7) and suppose no services can be found for which A output format and B input format match (link consistency failure).

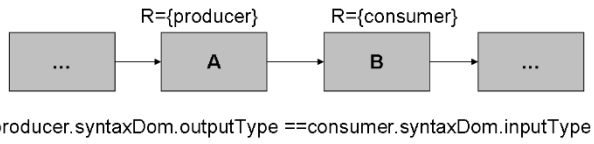

FIGURE 7. Sequence template example.

The above problem can easily be solved by replacing placeholders A and B with an adaptation template made up of three nodes (see Fig. 8); the first and last nodes mimic features of placeholders A and B, and the middle node (called T) is meant to hold a format translation service, e.g., it declares a service consistency rule that constrains service typology to be of type ''transcoding'' (meaning that only services able to perform format transcoding should be placed in between) and two link consistency rules (i.e., A output compatible with T input, and T output compatible with B input). Our system may now find a correct arrangement of services to satisfy A and B nodes, and a third service to place in between that acts as a format transcoder, hence granting link consistency.

As a concrete example, the Aggregator service in Section IV.E provides an XML-based output instead of plain text. That violates the link consistency check with the following SMSSender. By applying the adaptation template, our platform may recursively place a suitable *XML-to-Text*

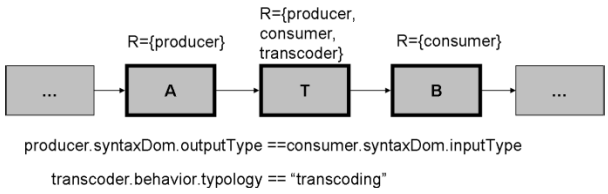

FIGURE 8. Recursive template definition example.

transcoder service in between to guarantee link consistency satisfaction (as in Fig. 9).

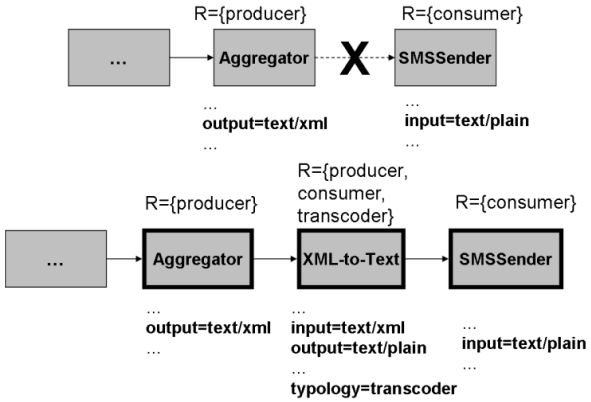

FIGURE 9. Recursion example in a real template.

B. CONSISTENCY CHECK IMPLEMENTATION

Many different implementations may be conceived to realize the reification algorithm of our model. Basically, however, its core activity is to verify template consistency of different alternative solutions; this usually entails replacing a finite number of template placeholders with services that satisfy service and template consistency rules.

Constraint Satisfaction Problem (CSP) techniques [30] aim at solving the problem of assigning values (from a finite set/domain of available values) to a finite number of variables, by granting the satisfaction of constraints over the values of variables.

Consistency check may therefore be easily implemented as a CSP problem in which a finite number of template placeholders (variables) need to be filled with services (from a large but finite catalogue), by satisfying service and template consistency rules (constraints). Note here that rules may easily be intended as constraints on services to fill placeholders; in fact rules expressed directly in terms of placeholders are full-fledged constraints on our system variables (placeholders), whereas rules expressed on roles may easily be turned into constraints on placeholders by simply applying them to each placeholder that declares the involved roles.

Given user requirements in terms of a template and a set of rules, the main steps to translate them into one concrete workflow (and also more than one) of services entails the following main logical steps:

1) translate rules (coming both from template definition and user requirements) into constraints, by replicating rules for each declaring placeholder;
2) find solutions via usual CSP techniques;
3) if more solutions are found, score solutions;
4) if no solution is found, try recursive solutions.

Constraint satisfaction problems have long established strategies and techniques to generate and verify correct solutions [31]; some are extremely simple to implement but quite naïve and resource-wasting whereas others are extremely complex and sophisticated, and grant for better performance [32].

The key aspects of any constraint satisfaction strategy mainly relate to solution exploration (the strategy for finding out solutions) and to algorithm completeness (i.e., how many solutions to identify). Both aspects have a relevant impact on performance of the CSP solution, hence both of them should be carefully taken into account and optimized to realize effective constraint satisfaction strategies.

1) SOLUTION EXPLORATION

The brute-force solution exploration strategies (called ''generate and test'' strategies) typically blindly instantiate all possible permutations of variable assignments and check constraints ''a posteriori'', only when a complete solution is generated.

The number of possible solutions heavily depends on the number of available services and of placeholders to fill: $cardinality\,(S)^{cardinality(PH)}$.

Obviously, this approach quickly becomes unmanageable as the service and placeholder number increases, making it almost infeasible even in limited and simple real deployment scenarios.

Smarter solutions typically generate solutions by visiting a solution tree where nodes represent variables to assign and outgoing branches represent value assignments: this model allows to easily evaluate partial assignments during the process, hence allowing to ''a priori'' prune branches (excluding even large sets of solutions) that represent wrong assignments (assignments that conflict with constraints). Algorithms described in the following pursue the latter approach.

2) ALGORITHM COMPLETENESS

A complete strategy typically explores the whole solution tree and is therefore able to provide all possible solutions for a CSP problem, whereas non-complete strategies usually explore only portions of the tree and provide only a limited set of solutions (sometimes only the first one).

As the complexity of the CSP grows (the increasing number of services, constraints, and placeholders to fill), the solution tree to explore may become very large, thus making the choice between complete and non-complete strategies more and more relevant and the decision crucial. Furthermore, non-complete strategies may not always be a viable option, since some end-user Quality-of-Service policies may require to

find out the best possible solution, instead of providing just a good one.

Our implementation focuses on both "extremes" (complete and non-complete strategies) to give a full and wide-range perspective of scalability and performance of our platform and to evaluate upper and lower bounds in terms of system performance. At this stage we are neither specifically interested in implementing and testing a number of possible optimizations for a given strategy nor in evaluating different heuristics on service, variable, and constraint ordering during the evaluation phase, even if research has showed that they could increase the overall performance of the system.

The non-complete strategy relies on a standard backtracking approach: this algorithm explores a solution tree where each node represents a variable to assign (placeholder to fill) and each outgoing branch represents a possible assignment in the variable domain (available services). Thus, one solution is a path from the root of the tree to a leaf that identifies one assignment for each variable. The standard backtracking algorithm explores the solution tree depth-first and, at each level checks whether constraints are satisfied by current assignments. If it is so, the algorithm may go down one level and proceed with next assignments; if not satisfied, the algorithm tries to assign another value in the domain. If no valid assignment can be found at a given level N (for a given variable), then assignments from level 0 to level N-1 should be revised: the algorithm then steps back to level N-1 and tries another assignment and so on.

The complete strategy relies on an extension of the standard backtracking algorithm; this strategy again explores the solution tree depth-first, but does not stop once the first solution is found, and keeps visiting the solution tree until all possible constraint-consistent assignments are explored.

Both solutions represent a major improvement with respect to brute-force ones that blindly evaluate all possible permutations of variable assignments; in fact, checking constraints at each level allows to detect and prune (early in the algorithm execution) all of the branches of the solution tree (e.g., set of possible assignments) that will not lead to feasible solutions.

VI. EXPERIMENTAL RESULTS

In order to prove the feasibility of our approach, we realized a number of concrete deployments for real usage scenarios and challenged our service composition platform against them. These deployments also allowed us to distill useful hints about the level of complexity average end users are able to tame. Typically, in fact, users seem to use templates with three to five placeholders and we perceived that more complex templates require a lot more effort on them (e.g., more constraints to specify) and ultimately tend to be less intuitive and usable for average end users. As for service metadata, over-described services (e.g., high number of metadata properties) again tend to be confusing and users rather focus on simple, limited, and understandable properties.

The aforementioned guidelines from real-life scenarios, allowed us to devise suitable testbeds (reported in the following) to intensively test our implementation under real operating conditions. The reference implementation and all tests ran on a single computation node equipped with commodity hardware (Pentium4 3 GHz core, 2 GB RAM, Linux kernel 2.6.27) and the prototype has been developed in Java (Java JDK 1.6).

A. CONSISTENCY PHASE EVALUATION

Since the template consistency phase (i.e., the CSP solution) is the most complex task in the process of our service composition platform, we have first evaluated average service times for both the standard (non-complete) backtracking strategy and for the complete one. Specifically, we were interested in evaluating variations of the average response time when varying two main coordinates of the system: the number of available services and the complexity of the template (specifically, the number of placeholders to fill in).

Fig. 10 shows that the number of template placeholders has a huge impact on the overall performance of the system: reification of simple templates (e.g., with three placeholders) can be resolved both with complete and non-complete strategies, and still keeps reasonable average execution times, even for large sets of available services. When the number of placeholders increases, complete strategies quickly tend to become less performing and average execution times may "explode" to dozens of seconds (with very large service sets). Given this evaluation, we have devised some adaptive algorithms that may runtime switch between complete and non-complete solutions based on current system status (the number of available services) and user requirements (specifically, which template and how many placeholders in the template).

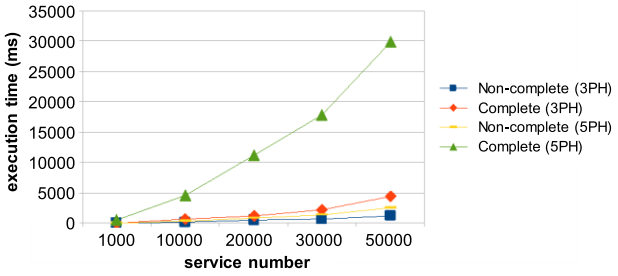

FIGURE 10. Average consistency execution times.

B. RECURSION EVALUATION

Our platform adopts recursion to handle limited and well-known consistency failures. We evaluated the impact of recursion on execution times in a typical test case (similar to the one described in Section IV.E). Specifically, users require to reify a template (made up of three placeholders) whose first to steps are constrained (among other consistency rules) by a link consistency rule. In our testbed, no services can be found to directly satisfy this requirement; hence, our algorithm is forced to recursively insert an adaptation template in between and tries to recalculate service consistency.

Different implementation strategies for recursion may be devised: the most naïve (and less performing) implementation recomputes the novel and more complex problem (more placeholders to fill in) from scratch. Other strategies may be optimized, for instance by restarting from the portion of reification sets that, even if failing to satisfy the original problem, may still be promising (e.g., reification sets that fail only for the target link consistency rule).

Currently, we are interested in proving that recursion is a viable option when no solution for the original problem can be found, rather than providing a deep and detailed review of recursion optimization strategies. Thus, we tested and reported results for the simplest and worst performing implementation (i.e., restarting service consistency from scratch) and we are currently working on implementing and evaluating more sophisticated solutions.

Fig. 11 provides evaluation results of service consistency execution times, for both complete and non-complete strategies, and reports execution times for both a non-recursive case (e.g., available services can directly reify the template) and a recursive one (e.g., available services do not directly reify the template, and hence require recursion adoption).

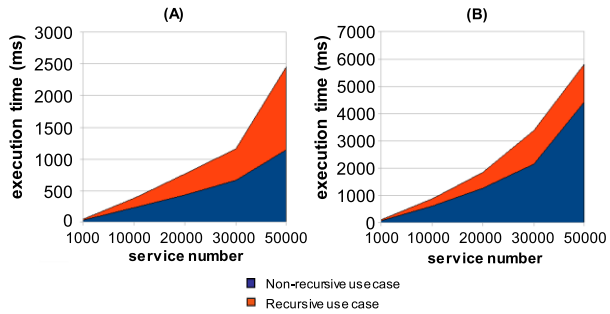

FIGURE 11. Recursive vs. non-recursive cases (average execution times in ms). (A) - Non-complete strategy (3 placeholders). (B) - Complete strategy (3 placeholders).

Results show that overall execution times remain reasonable and limited, especially for non-complete strategies. On the contrary, complete strategies obviously impose a much higher overhead on execution times than non-complete ones, since they basically compute a novel and more complex (at least one more placeholder to fill in) composition problem from scratch, hence needing to explore all the novel and more complex solution trees to find all possible solutions. In case recursion needs to be adopted, a simple optimization strategy to limit execution times could be to force the adoption of a non-complete solution strategy for the novel (recursive) problem.

We then focused on a recursive case and estimated the impact of the recursive phase in the overall execution times: Fig. 12 reports the percentage of time spent in the recursive phase of the algorithm with respect to the total amount of execution time.

Results show that the recursive phase costs more than the original problem solution phase and employs from 64% up

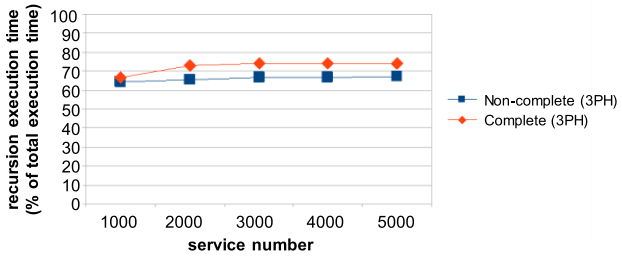

FIGURE 12. Recursive phase execution time (percentage with respect to total execution time).

to 74% of the whole execution time. This again is a result of our first naïve implementation of the recursion phase that basically solves a new and more complex problem from scratch.

C. MEMORY FOOTPRINT

To better estimate the footprint and impact on computational resources of our algorithm, we evaluated the average memory consumption during the overall composition phase. Again, we tested our platform by varying two main coordinates of the system: the number of available services (abscissae axes), the complexity of the template (specifically, the number of placeholders to fill in), and the type of strategy (complete and non-complete). Since our tests involve a large number of services (up to 50000), a non-negligible portion of the overall used memory is reserved to handle them; this is why in Fig. 13 we reported both the overall memory consumption and the memory portion reserved for test services.

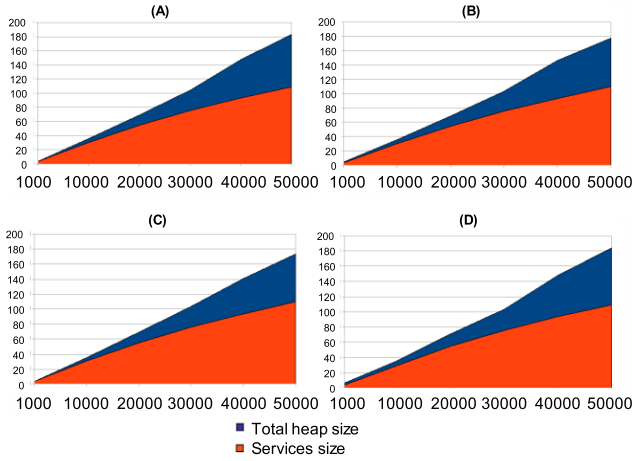

FIGURE 13. Memory footprint. (A) - Non-complete strategy (5 placeholders). (B) - Complete strategy (5 placeholders). (C) - Non-complete strategy (3 placeholders). (D) - Complete strategy (3 placeholders).

Results show that memory consumption grows almost linearly with the number of services (ordinate axis reports memory usage in MB). For limited sets of services (up to thousands), the most relevant portion of memory usage relates to services themselves. When the number of services grow to dozens of thousands, it is memory consumption other

than services that becomes more relevant (i.e., data structures involved in the algorithm become larger and more complex) but still grows linearly with the number of services, hence remaining viable and manageable even for very large set of services.

Comparison of graphs A, B, C, and D shows that memory consumption is very similar even when varying the strategy and the number of template placeholders: this allows to state that memory consumption of our algorithm is neither influenced by strategy completeness nor by template complexity.

D. SCORING PHASE EVALUATION

We have also evaluated the composition scoring phase and, as we expected, it proved to be the least relevant and time-consuming part of the algorithm, with average scoring times of hundreds of microseconds (at least three orders of magnitude below average CSP solution times).

This is rather obvious as a result because scoring usually takes place on a very limited number of solutions (in our test scenarios, up to a dozen) with respect to the consistency phase that evaluates large portions of huge solution trees. Nevertheless, this result suggests that trying to provide users with the "best" solution (either out of all possible solutions in complete cases, or out of a limited subset of solutions in case of non-complete strategies) to fit their needs is not a costly task and can always be pursued without imposing relevant overhead on the overall process.

E. SUMMARY AND ONGOING IMPROVEMENTS

To summarize, our service composition algorithm is basically made of two main parts: template consistency exploits typical CSP-based techniques (both complete and non-complete ones) to generate feasible solutions, and template scoring allows to rank these solutions for given criteria; in case no solutions can be found, our algorithm may selectively apply recursion strategies to overcome specific service composition violations; our tests showed that recursion comes at a non-negligible but reasonable cost. We stress that algorithm completeness (i.e., finding all possible solutions) may heavily impact on execution times, especially with increasing service number and complex templates; therefore, our algorithm is able to dynamically inspect current runtime conditions and force non-complete approaches, such as in case service number surpasses a given threshold.

Even though, the very basic strategies for solution generation and checking have proven to perform well under non-trivial execution conditions (high number of services and templates), we felt the need to estimate whether our model and implementation left room for improvements and, in case, to evaluate the degree of optimization that could potentially be reached.

Heuristics and optimizations for CSP problems are a long-debated field and usually, powerful and sophisticated heuristics tend to be extremely tailored to very limited and specific sets of problems (e.g., some heuristics behave extremely well for over-constrained CSP problems, whereas others largely improve scenarios with limited variable cardinality).

We implemented and evaluated a very simple and general-purpose heuristic for service selection; for each placeholder, such heuristic basically filters out all candidate services (from the set of available ones) that do not satisfy service consistency rules expressed on that placeholder. Service consistency rules, in fact, constrain metadata properties of services willing to replace a given placeholder, and therefore represent a convenient way to shrink out the set of candidate services before the algorithm tries to instantiate that placeholder. That results in a relevant reduction of the solution tree, hence in faster exploration strategies.

As reported in Table 1 (left sub column for each service number group), results show a relevant speedup of average execution times, ranging from 1.09 up to 3.72.

TABLE 1. Execution time speedup/filtering time (% of total exec.ltime).

Execution time speedup /Filtering time (% of total exec. time)

	Number of available services							
Template and strategy	1000		10000		30000		50000	
3 placeholders, complete	10,44	1,43	11,34	1,33	11,67	1,25	12,01	1,09
5 placeholders, complete	28,91	1,36	33,49	1,27	36,05	1,26	36,74	1,22
3 placeholders, non-complete	48,18	1,84	26,56	1,35	32,13	1,77	33,19	1,89
5 placeholders, non-complete	28,88	2,31	35	2,32	40,15	2,48	35,08	2,02

We also estimated the cost of adopting this heuristic and we reported (see Table 1 - right sub column for each service number group) the percentage of time spent during service filtering out of the overall process time. Results show that service filtering requires a non-negligible but always limited percentage of the overall execution time. Service filtering clearly depends on the size of the explored solution tree, since service filtering occurs at any instantiation of a variable (placeholder). For complete strategies, tests show that service filtering time, as expected, grows as the number of services: in fact, the consistency evaluation phase visits the whole solution tree, which gets larger with the number of services. For non-complete strategies, this kind of correlation is no longer recognizable since the visited solutions (hence the occurrences of filtering steps) are only subportions of the whole solution tree and their size does not necessarily depend on the number of available services; for instance, some larger instantiations may find the first solution earlier than smaller ones, and therefore require less filtering iterations.

VII. CONCLUSION

Ubiquitous computing scenarios and the fragility of emerging highly heterogeneous wireless networks stress the need for flexible and user-friendly service-oriented support middleware. This work proposes both an open and extensible composition model for reliable ubiquitous computing, and a concrete, user-friendly, and effective implementation of that model. To prove the viability of our approach we deployed our platform and evaluated its usability in several different

scenarios, with large and heterogeneous groups of users and services. Obtained results showed that the platform is intuitive, easily usable by average end users, and easily extensible to cope with new application scenarios.

We also extensively tested and benchmarked our platform and we observed that our implementation scales well even for very large and complex deployment scenarios, by keeping reasonable execution times able to grant fast recovery and survivability of service provisioning. We are currently working on optimizations of the composition algorithm and preliminary results show remarkable improvements in average execution times. That grants our platform directions of evolution and allows focusing our future work on even more complex, different, and challenging scenarios.

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 3, no. 3, pp. 3–11, Jul. 1999.
[2] S. Kalasapur, M. Kumar, and B. A. Shirazi, "Dynamic service composition in pervasive computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 7, pp. 907–918, Jul. 2007.
[3] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang, "Middleware for pervasive computing: A survey," *Pervasive Mobile Comput.*, vol. 9, no. 2, pp. 177–200, 2013.
[4] R. Wang *et al.*, "Web service composition using service suggestions," in *Proc. IEEE World Congr. Services*, Jul. 2011, pp. 482–489.
[5] H. Mukhtar, D. Belaïd, and G. Bernard, "Dynamic user task composition based on user preferences," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 1, 2011, Art. no. 4.
[6] D. Liu, H. Zhu, and I. Bayley, "From algebraic specification to ontological description of service semantics," in *Proc. 20th Int. Conf. Web Services (ICWS)*, Jun./Jul. 2013, pp. 579–586.
[7] D. Braga, S. Ceri, F. Daniel, and D. Martinenghi, "Mashing up search services," *IEEE Internet Comput.*, vol. 12, no. 5, pp. 16–23, Sep. 2008.
[8] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *IEEE Internet Comput.*, vol. 12, no. 5, pp. 44–52, Sep. 2008.
[9] A. H. H. Ngu, M. P. Carlson, Q. Z. Sheng, and H.-Y. Paik, "Semantic-based mashup of composite applications," *IEEE Trans. Services Comput.*, vol. 3, no. 1, pp. 2–15, Jan./Mar. 2010.
[10] G. Huang, Y. Ma, X. Liu, Y. Luo, X. Lu, and M. B. Blake, "Model-based automated navigation and composition of complex service mashups," *IEEE Trans. Services Comput.*, vol. 8, no. 3, pp. 494–506, May/Jun. 2015.
[11] M. P. Papazoglou and D. Georgakopoulos, "Introduction: Service-oriented computing," *Commun. ACM*, vol. 46, no. 10, pp. 24–28, Oct. 2003.
[12] C. Bussler, *B2B Integration: Concepts and Architecture*. Berlin, Germany: Springer-Verlag, 2003.
[13] OASIS. *OASIS Web Services Business Process Execution Language (WSBPEL)*. Accessed: Jun. 2018. [Online]. Available: https://www.oasis-open.org/committees/wsbpel/
[14] W3C Consortium. *Web Service Semantics—WSDL-S*. Accessed: Jun. 2018. [Online]. Available: http://www.w3.org/Submission/WSDL-S/
[15] W3C Consortium. *OWL-S: Semantic Markup for Web Services*. Accessed: Jun. 2018. [Online]. Available: http://www.w3.org/Submission/OWL-S/
[16] C.A. Petri, "Kommunikation mit Automaten," (in German), Ph.D. dissertation, Rheinisch-Westfäisches Inst. Instrum. Math., Univ. Bonn, Bonn, Germany, 1962.
[17] R. Milner, *Communication and Concurrency*. London, U.K.: Prentice-Hall, 1989.
[18] C. A. R. Hoare. *Communicating Sequential Processes*. London, U.K.: Prentice-Hall, 1985.
[19] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," in *Proc. Int. Conf. World Wide Web*, 2002, pp. 77–88.
[20] R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. Massachusetts, MA, USA: MIT Press, 2001.
[21] N. B. Mabrouk, N. Georgantas, and V. Issarny, "Set-based Bi-level optimisation for QoS-aware service composition in ubiquitous environments," in *Proc. Int. Conf. Web Services (ICWS)*, Jun./Jul. 2015, pp. 25–32.
[22] A. Moustafa, M. Zhang, and Q. Bai, "Trustworthy stigmergic service compositionand adaptation in decentralized environments," *IEEE Trans. Services Comput.*, vol. 9, no. 2, pp. 317–329, Mar./Apr. 2014.
[23] A. Imed and M. Graiet, "An automatic configuration algorithm for reliable and efficient composite services," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 1, pp. 416–429, Mar. 2018.
[24] N. Chen, N. Cardozo, and S. Clarke, "Goal-driven service composition in mobile and pervasive computing," *IEEE Trans. Services Comput.*, vol. 11, no. 1, pp. 49–62, Jan./Feb. 2018.
[25] Y. Wu, C. Yan, L. Liu, Z. Ding, and C. Jiang, "An adaptive multilevel indexing method for disaster service discovery," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2447–2459, Sep. 2015.
[26] P. Pal, M. Atighetchi, J. Loyall, A. Gronosky, C. Payne, and R. Hillman, "Advanced protected services—A concept paper on survivable service-oriented systems," in *Proc. 13th IEEE Int. Symp. Object/Compon./Service-Oriented Real-Time Distrib. Comput. Workshops*, Carmona, Seville, Spain, May 2010, pp. 158–165.
[27] B. Christensen. *Application Resilience in a Service-Oriented Architecture O'Reilly Radar*. Accessed: Apr. 5, 2018. [Online]. Available: http://radar.oreilly.com/2013/06/application-resilience-in-a-service-oriented-architecture.html
[28] *Hystrix*. Accessed: Apr. 5, 2018. [Online]. Available: https://github.com/Netflix/Hystrix/
[29] P. Barthelmess and J. Wainer, "Workflow modeling," in *Proc. CYTED-RITOS Int. Work. Groupware*, Sep. 1995, pp. 1–13.
[30] E. Tsang, "A glimpse of constraint satisfaction," *Artif. Intell. Rev.*, vol. 13, no. 3, pp. 215–227, 1999.
[31] G. Kondrak and P. van Beek, "A theoretical evaluation of selected backtracking algorithms," *Artif. Intell.*, vol. 89, nos. 1–2, pp. 365–387, 1997.
[32] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artif. Intell.*, vol. 14, no. 3, pp. 263–313, 1980.

PAOLO BELLAVISTA (M'98–SM'06) received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2001. He is currently an Associate Professor with the Università di Bologna. His research activities span mobile agent-based middleware solutions and pervasive wireless computing to location/context-aware services and management of cloud systems. He serves on the Editorial Boards of the IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, the IEEE TRANSACTIONS ON SERVICES COMPUTING, *Pervasive Mobile Computing* (Elsevier), and the *Journal of Network and Systems Management* (Springer).

ANTONIO CORRADI (M'77) received the degree from the University of Bologna, Italy, and the M.S. degree in electrical engineering from Cornell University, USA. He is currently a Full Professor of computer engineering with the Università di Bologna. His research interests include distributed systems, middleware for pervasive and heterogeneous computing, infrastructure for services, and network management.

LUCA FOSCHINI (M'04) received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2007. He is currently an Assistant Professor of computer engineering with the Università di Bologna. His research interests include pervasive wireless computing environments, system and service management, context-aware services, and management of cloud computing systems.

STEFANO MONTI received the Ph.D. degree in computer science engineering from the University of Bologna, Italy, in 2009. He is currently an IT Consultant with Imola Informatica SpA. His research interests include pervasive computing, mobile agent platforms, and service-oriented architectures, and service composition and integration.