A Tile-based Approach for Self-assembling Service Compositions

Luca Cavallaro
DEI, Politecnico di Milano
Milano, Italy
cavallaro@elet.polimi.it

Elisabetta Di Nitto
DEI, Politecnico di Milano
Milano, Italy
dinitto@elet.polimi.it

Carlo A. Furia
Chair of Software Engineering
ETH Zurich, Switzerland
caf@inf.ethz.ch

Matteo Pradella
CNR IEIIT-MI
Milano, Italy
pradella@elet.polimi.it

***Abstract*—This paper presents a novel approach to the design of self-adaptive service-oriented applications based on a new model called *service tiles*. The approach allows designers to develop a service-oriented system by building an assembly of component services that accomplishes the given goal. The assembly is computed automatically starting from the specification of a subset of the whole system, a few constraints, and the goals the application should fulfill. An application designed according to the service-tile model can also dynamically self-adapt by replacing, in part or entirely, services in the assembly whenever they fail or the application context changes. The service-tile design technique has been implemented in a prototype and some experiments with several examples demonstrate the feasibility of the approach and its practical efficiency.**

I. Introduction

Service-oriented architectures are based on the integration of third-party components published on a network as software services. Applications built in this scenario cannot rely on the traditional closed-world assumption, where developers know all available components *a priori* and can model and analyze their interactions exhaustively. Such applications are usually situational: changes in the context can trigger the selection of new components, possibly unforeseen by the developer, at runtime. This entails that a developer should be able to defer at runtime part of the decisions about which components should be included in the application. Moreover runtime *self-adaptation* features are required to compensate for the lack of control on used software services and to react to fast context changes.

To cope with this uncertainty a developer should be able to build an application by specifying a subset of the whole system, a few constraints, and the goals the application should fulfill. Then, from the provided specification, a suitable composition of available services is computed. This problem is widely treated in literature (see Section VI). Most existing approaches, however, suffer from significant overheads in computing suitable aggregates of services.

In this paper, we develop an approach that identifies proper service assemblies very efficiently by exploiting the the notion of *tile-based systems*.

This research has been partially funded by the European Community's FP7/2007-2013 Programme, grant agreement 215483 (S-Cube), IDEAS-ERC Programme, Project 227977 (SMSCom), and MIUR PRIN project "D-ASAP: Dependable Adaptable Software Architecture for Pervasive Computing".

Tile-based systems are models of computation that have been introduced to describe biological processes such as DNA recombination (e.g., [1]) and have been exploited in the context of formal languages (see e.g., [2], which introduces and studies *Tiling Systems*; [3], which defines the equivalent *Wang Systems*; and [4], where the problem of parsing and automatically generating pictures from tiling systems is considered). Tile-based systems define computations as processes that assemble atomic units called *tiles*. Each tile can be composed only with certain other tiles, according to the symbols they carry. The resulting assembly process is similar to building a jigsaw puzzle with the pieces from a given box.

Recently, various authors (e.g., [5], [6]) have suggested that analogous formal models can be proficiently used to describe highly dynamical software and networked systems. In particular [5] proposes to model elementary components with a variant of Wang tiles, where constraints for building the system are specified only locally on each single tile, independently of the actual setup of the overall system. The properties of the system are then satisfied by construction as they *emerge* out of the composition of tiles according to the local compatibility rules.

Working in the same line, we tackle the problem of service composition by means of a variant of tile-based system that we call *service tiles*. In a sense our concept of tile is a generalization of a standard Wang tile, since the latter is limited to a fixed square structure, thus bearing at most four symbols, while we also consider an arbitrary number of symbols. *Service tiles* also enable systems to compute proper self-adaptation strategies in case of failure of component services or of changes in the context. The "local nature" of tile-based systems allow for the identification of replacement strategies that are usually confined to the region of failure and does not require an expensive re-design of the whole system from scratch. In this paper, we rely on self-assembly capabilities of tile-based systems to address the aforementioned problems.

Compared to other approaches for automatic composition, ours offers a terse model that allows a very efficient computation of service compositions. To demonstrate this, we implemented the tile-based model using integer linear programming, and we built a proof-of-concept framework (see Sections III-A and IV). The experiments we performed (see Section V) confirm that our approach can efficiently automate

the task of computing a proper service assembly.

Another interesting aspect of the approach is that is does not assume, as many others do, the existence of a single orchestrator for the assembly. Instead, it allows each component to be responsible for coordination with its "neighbor services", thus reducing the burden of tasks demanded to a centralized executor.

II. Motivating example

Commuters prefer to use public transportation to move from home to work every day. Unfortunately, public transportation may suffer schedule delays or denials of service due to overcrowding, technical problems of trains or buses, or severe weather conditions. We assume that public transportation users are equipped with portable computing devices, which enable them to retrieve information and suggestions about their planned route. To this end, public transportation authorities provide mobility information services by gathering and combining data about vehicles moving along the network. Mobility information services can signal, for example, delays of trains or buses, or traffic jams.

Information services can also forecast the status of the transportation network by elaborating sensed weather conditions. For instance, a likely delay of buses can be forecast based on the presence of snow. When problems are signaled on his/her route, a commuter may re-plan his/her way, for example by switching from a bus to a train. In order to do so, he/she also needs access to a public transportation route planning service and to a ticket vending service.

The transportation network is divided into five zones and a different information service is available in each of them. This gives more detailed information to commuters, but it also complicates the selection of the services to be used according to the the zone of the city a commuter is in or wants to reach.

This scenario makes the life of the designer significantly harder compared to a traditional system as it is unlikely that the designer can get a total knowledge of the services involved in the overall system. First of all, invoked services should be selected according to the application context, since, for instance, a commuter is interested to contact the information services from those zones where he/she is, or is planning to go through in his/her route. Moreover, the capability of information services to provide up-to-date information depends on vehicle information and weather sensors spread around the network, which may be unreliable since vehicles may go through zones where it is impossible to send their data, and sensors can fail for technical problems. Thus, the application should be able to react to these failures replacing some of the unavailable information sources on-the-fly. Consider, for instance, the case of weather data: if sensors in one area of the network fail it is possible that sensors of a neighbor area are still reachable and can replace the unavailable ones. These new sensors would likely be less accurate and contribute to a solution of lower quality, but by using them the application can at least continue to work.

Offered Operations	Required Operations
commuter buddy	route planning mobility information *[location]* ticket booking

TABLE I: The *Commuter Buddy* Application

We consider the problem of formalizing this scenario. The application residing of the commuters' portable devices is called *Commuter Buddy*; Table I summarizes it in terms of offered and required operations. In this model, which we are going to formalize in the following sections, the concepts of offered or required operations are *application-specific* and could represent, for instance, a service operation or a UML-style offered or expected *interface*.

Table II lists a possible set of other services, available to *Commuter Buddy* at one instant of the application lifetime. Different services provide data for different zones and a less accurate service covering forecasts for the whole city exists as well. This entails that the selection of weather forecast (and mobility information) services depend on the departure and destination points. This is represented in Table I and in Table II by parameters in square brackets, appended to offered or required operations. For instance, *Mobility Information Center Milan* offers a mobility information service specific for the city center area, through the operation *mobility information*. We represent this offered service locality by appending the parameter *[CM]* to the name of the offered operation.

As the reader can notice, there is no required operation *mobility information [CM]* for the *Commuter Buddy Application* (see Table I), while a *mobility information [location]* appears between required operations. In this case we use the *[location]* parameter to point out that the zone for which *mobility information* is required is left unspecified at design time. The decision about which zone the operation is required for will be made at runtime, when the context information will be available (e.g., when the user is going through the city center).

The situational nature of the application makes it difficult to have a complete knowledge at design time about which services will be suitable for binding at runtime in a given instant. Moreover, some particular characteristics call for a decentralized coordination of the services. Consider for instance the services providing mobility information for the various parts of the city. Their efficiency depends on the weather services covering the corresponding part of the city. If we consider the traditional central orchestrator hypothesis the mobility information service would be seen as a black box and, in case of failure of the corresponding weather service, it would stop working and need to be replaced. This would make commuters lose information about the corresponding zone.

In the following sections, the *Commuter Buddy* application will be the running example to demonstrate the service-composition model based on tiles. The examples will hint at how some of the aforementioned difficulties in developing a service-based application can be approached and overcome.

SERVICE NAME	OFFERED OPERATIONS	REQUIRED OPERATIONS
Vehicle Information 1	vehicle information *[V1]*	
Vehicle Information 2	vehicle information *[V2]*	
Vehicle Information 3	vehicle information *[V3]*	
Vehicle Information 4	vehicle information *[V4]*	
Route planner	route planning	
Weather forecast Center Milan	weather forecast *[CM]*	
Weather forecast North Milan	weather forecast *[NM]*	
Weather forecast South Milan	weather forecast *[SM]*	
Weather forecast East Milan	weather forecast *[EM]*	
Weather forecast West Milan	weather forecast *[WM]*	
Weather forecast Milan	weather forecast	
Mobility Information Center Milan	mobility information *[CM]*	weather forecast *[location]* vehicle information *[vehicle]*
Mobility Information North Milan	mobility information *[NM]*	weather forecast *[location]* vehicle information *[vehicle]*
Mobility Information South Milan	mobility information *[SM]*	weather forecast *[location]* vehicle information *[vehicle]*
Mobility Information East Milan	mobility information *[EM]*	weather forecast *[location]* vehicle information *[vehicle]*
Mobility Information West Milan	mobility information *[WM]*	weather forecast *[location]* vehicle information *[vehicle]*
TicketBooking	TicketBooking	Payment
Payment gateway 1	Payment	
Payment gateway 2	Payment	

TABLE II: An example of available services

III. TILE-BASED SYSTEMS AND THEIR APPLICATION TO SERVICE COMPOSITION

This section presents a formal model of services and service composition based on the notion of *tile*. A developer approaching the problem of designing a service-based application with this model would first abstract the structure of the process of the application and would define it only in terms of offered and required operations. The designer builds a workflow for the application and defines which external services should be invoked. These external services are associated to an identifier that can be a plain string or a logical formula that represents a goal. In both cases it is made available through a service facet [7].

Our model represents these facets as atomic symbols. Correspondingly, we develop an analysis technique which allows the discovery of an assembly of suitable services for the application under design. The developer would bind the resulting composition at design time. At runtime, the assembly can be totally or partially recomputed, in reaction to changes in the application context or to failures of services.

A. Service tiles: tiles for services

Let us start by introducing a new kind of tile-based system that can suitably represent the composition of a finite set of services. We remark that our model is quite different from the traditional literature on the subject, such as *tiling systems* or *Wang systems*: the service-tile model is only loosely inspired by these forerunners.

The rules of our model definition must ensure that services are composed correctly. While the notion of (correct) service composition is a very broad one, here we focus on a specific — yet significant — aspect, namely the problem of creating a composition by selecting services in such a way that every service request is satisfied by some other unit in the composition. In other words, the solution to our problem is a composition that "works" when considered in its entirety.

We represent offered and requested operations (i.e., facets) syntactically, with symbols from a finite alphabet Σ. For instance, *mobility_information* and *payment* are two operations of the running example. Offered and requested operations are grouped into *service units* that represent the *tiles* of our system. Each unit comes with a vector of (scalar) costs of arbitrary size (possibly empty). An element in the cost vector represents the magnitude of a certain cost dimension resulting from using the operations offered by the service unit.

Definition 1. A *service unit* U is a tuple $\langle \Sigma, C, O, R \rangle$ of offered $O = \{o_1, o_2, \ldots\}$ and requested $R = \{r_1, r_2, \ldots\}$ operations from alphabet Σ, together with a cost vector $C = [c_1, c_2, \ldots] \in \mathbb{N}^{|C|}$.

A service unit U will be conveniently represented as $[c_1, c_2, \ldots; +o_1, +o_2, \ldots, -r_1, -r_2, \ldots]$.

A collection of service units over a common alphabet with cost vectors of the same size constitutes a *service base*.

Definition 2. A *service base* S is a tuple $\langle \Sigma, k, \mathcal{U} \rangle$, where $\mathcal{U} \subseteq \mathbb{N}^k \times 2^\Sigma \times 2^\Sigma$ is a set of service units over operations in the common alphabet Σ, each with its cost vector of size k.[1]

Figure 1 pictures a possible service base for the running example (cmp. Table II), with omitted costs.

[1]More explicitly, each element $(C, O, R) \in \mathbb{N}^k \times 2^\Sigma \times 2^\Sigma$ represents a service unit $\langle \Sigma, C, O, R \rangle$.

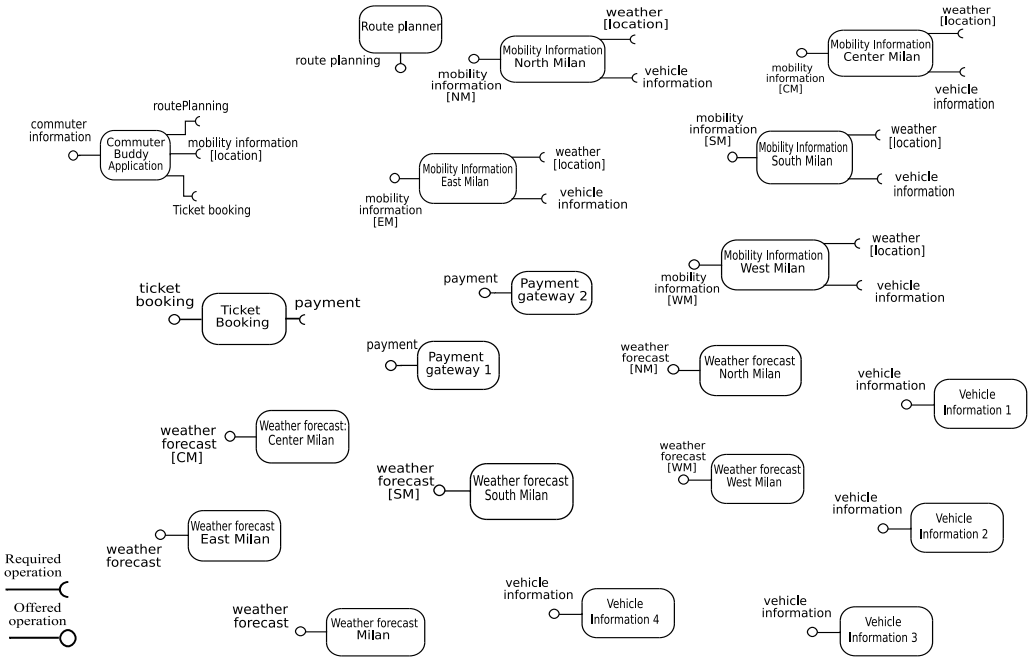

Fig. 1: Service units of the Application.

B. A formalization of the service-building problem

We are interested in finding consistent compositions of service units from a given service base. Intuitively, a service unit in a compound can offer its operations in O only if it works properly, that is only if all its operation requests in R are satisfied by some other chosen unit. Also, an operation request $-x$ is satisfied only by an operation offer $+x$. Correspondingly, the *service system building problem* requires to compose service units from a service base in such a way that they can function properly, that is all requests of all instantiated units are satisfied. This is the formalization of the service composition problem that we consider in this paper.[2]

Definition 3 (Service-building problem)**.** Given:

- a service base $S = \langle \Sigma, k, \mathcal{U} \rangle$;
- a multi-set $I : \mathcal{U} \to \mathbb{N}$ of initial service units;
- a cost bound $K = [K_1, \ldots, K_k] \in (\mathbb{N} \cup \{\infty\})^k$;

determine if there exists a multi-set $T : \mathcal{U} \to \mathbb{N}$ of service units from the base such that:

1) $I \subseteq T$.
2) Every operation request from a service unit in T is satisfied by the offer of some service unit also in T. This is formalized by requiring that $\mathcal{R} \subseteq \mathcal{O}$, where $\mathcal{R} : \Sigma \to \mathbb{N}$ is the multi-set of operation requests defined by[3] $\mathcal{R}(x) = \sum_{\substack{U \in \mathcal{U} \\ x \in U|_R}} T(U)$ and $\mathcal{O} : \Sigma \to \mathbb{N}$ is the multi-set of operation offers defined by $\mathcal{O}(x) = \sum_{\substack{U \in \mathcal{U} \\ x \in U|_O}} T(U)$.
3) For each cost dimension $1 \leq i \leq k$, the total cost of the service units in T is bounded by K_i.

Notice that in our formal model any operation of any unit can be connected with the corresponding operation of any other unit in the solution. Hence, a notion of bi-dimensional locality — usually present in traditional tile-based systems — is lost, because the requirements for a correct composition are ultimately just on the cardinality of chosen tiles and not on their spatial displacement.

Intuitively, I represents the input that the system designer provides to the the problem. This input usually consists in one or more service units that represent the core of the application. In the running example of Section II, I would represent a single instance of the *Commuter Buddy* application, for which the designer wants to find a suitable binding.

$\mathcal{U}$ represents all service units that are available at the instant in which the solution for the service-building problem is computed. Available service units can be found, for instance, by querying a registry. Figure 1 pictures such a service base in our example.

Given I and $\mathcal{U}$, the service-building problem consists in finding a composition of instances of service units from $\mathcal{U}$ which satisfies all the requests of the service units in I and do not exceed the bounds on costs. For instance, unit *Ticket Booking* $= [1; +\mathit{ticket\_booking}, -\mathit{payment}]$ and unit *PaymentGW1* $= [3; +\mathit{payment}]$ can be composed in such a way that all requests are satisfied, the total cost is 4, and there

[2]Multi-sets are sets with repeated elements. With standard notation, a multi-set A over elements in D is represented by a function $A : D \to \mathbb{N}$ such that $A(d)$ is the cardinality of element $d \in D$ in A. The notion of subset is extended to multi-sets as customary: $A \subseteq B$ for multi-sets $A, B : D \to \mathbb{N}$ iff $A(d) \leq B(d)$ for all $d \in D$.

[3]$A|_B$ denotes the projection of tuple A over set B, i.e., B's component in A.

is exactly one instance of each unit. Since in general a service unit both offers and requests operations, including a service unit in the solution may trigger the need for more service units, in an iterative fashion.

IV. Implementation and practical issues

This section discusses how service-tile models can be implemented and analyzed using integer linear programming techniques and tools (Section IV-A), and how context information can be modeled using service tiles (Section IV-B). The section also outlines a proof-of-concept framework to support the integration of the service-tile approach in a real service-based application (Section IV-C). The framework is based on *SCENE* [8], which allows a developer to define and execute *self-adaptable* service compositions with dynamic service binding.

A. Encoding the service-building problem with integer linear programming

Integer Linear Programming (ILP) problems consist in minimizing the value of a linear function of some integer-valued variables subject to a set of linear inequality constraints on the same variables [9]. ILP is a natural formulation for many optimization problems. The technology for ILP solving is very mature: even if ILP is an NP-complete problem, a variety of practically very efficient off-the-shelf ILP solvers are available.

We show how to solve instance of the service-building problem (Definition 3) by encoding them as ILP problems. Given that the service-building problem is NP-complete (see the proof in the Appendix), this is the best possible encoding from a worst-case complexity viewpoint.

A service-building problem for generic service base $S = \langle \Sigma, k, \mathcal{U} \rangle$, initialization I, and cost bound K can be solved by encoding it as an ILP problem of size polynomial in $|\mathcal{U}|, |\Sigma|, k, \max_x I(x), \max_{1 \leq i \leq k} K_i$. To this end, consider $|\mathcal{U}|$ integer variables $u_1, u_2, \ldots, u_{|\mathcal{U}|}$, and let $\Sigma = \{x_1, x_2, \ldots, x_{|\Sigma|}\}$. Let us introduce the binary-valued functions $R(x_k, u_i), O(x_k, u_i)$ with $x_k \in \Sigma$ defined as the indicator functions $\mathbb{1}_{U_i|_R}(x_k)$ and $\mathbb{1}_{U_i|_O}(x_k)$ of the sets $U_i|_R$ and $U_i|_O$, respectively. $C_i = [c_{i1}, c_{i2}, \ldots, c_{ik}]$ denotes the cost vector of the service unit corresponding to u_i. The system of $|\mathcal{U}| + |\Sigma| + k$ linear inequality constraints and one objective function reported in (1) encodes the service-building problem.

The first $|\mathcal{U}|$ inequalities encode the data about the initial service units I; the following $|\Sigma|$ inequalities encode the requirement that the total amount of offers for any service must be no fewer than the total amount of requests for the same service; the last k inequalities constrain the total costs not to exceed the values in K.

A solution of the ILP problem of (1) is a vector of nonnegative integer values, one for each service unit in $\mathcal{U}$. Each value in the vector specifies how many instances of that service unit are included in a the composition. Correspondingly, it is straightforward to compute the the total cost of the composition.

$$
\begin{cases}
\min \sum_{1 \leq i \leq |\mathcal{U}|} \left(\sum_{1 \leq j \leq k} c_{ij} \right) u_i \quad \text{s.t.} \\
u_1 \geq I(u_1) \\
\vdots \\
u_{|\mathcal{U}|} \geq I(u_{|\mathcal{U}|}) \\
\sum_{1 \leq i \leq |\mathcal{U}|} (O(x_1, u_i) - R(x_1, u_i)) u_i \geq 0 \\
\vdots \\
\sum_{1 \leq i \leq |\mathcal{U}|} (O(x_{|\Sigma|}, u_i) - R(x_{|\Sigma|}, u_i)) u_i \geq 0 \\
\sum_{1 \leq i \leq |\mathcal{U}|} -c_{i1} u_i \geq -(K_1 + 1) \\
\vdots \\
\sum_{1 \leq i \leq |\mathcal{U}|} -c_{ik} u_i \geq -(K_k + 1)
\end{cases}
\tag{1}
$$

Service Unit	*Cardinality*	*Response Time (seconds)*
Vehicle Information 1	0	2
Vehicle Information 2	0	3
Vehicle Information 3	0	4
Vehicle Information 4	1	1
Route planner	1	1
WeatherForecast Center Milan	1	3
WeatherForecast North Milan	0	4
WeatherForecast South Milan	0	5
WeatherForecast East Milan	0	6
WeatherForecast West Milan	0	7
WeatherForecast Milan	0	3
Mobility Information Center Milan	1	1
Mobility Information North Milan	0	2
Mobility Information South Milan	0	3
Mobility Information East Milan	0	4
Mobility Information West Milan	0	5
TicketBooking	1	1
Payment gateway 1	1	1
Payment gateway 2	0	1
Total Cost (response time)		8

TABLE III: An example of solution for the service-building problem formulated as ILP.

Table III shows a possible solution to the ILP model of the service-building problem for the example introduced in Section II. The solution is a "snapshot" of a possible valid assembly under the assumption that the current location is *CM (Center Milan)* and where a mono-dimensional cost representing the response time of services is considered. The *Cardinality* column represents how many instances of a service unit of a certain type are needed in the solution.

B. Context representation in the service-building problem

Let us point out how we represented context information in the service-tile model of our running example. We augmented the set $\mathcal{U}$ of available service units with a few *virtual* service units. These do not represent any "real" service unit but have the only purpose of modeling the context. Each virtual service unit offers some operation required by one of the service units in $\mathcal{U}$ and requires some other operation which is used to select the service unit which is appropriate in the current context.

Consider for instance the example reported in Figure 2. The virtual service units *Mobility Information [location] Center Milan* and *Weather [location] Center Milan* represent the context dependences of operations *mobility information[location]* of unit *Commuter Buddy* and *weather [location]* of unit *Mobility Information Center Milan*, respectively. For instance, virtual service unit *Mobility Information [location] Center Milan* offers operation *mobility information [location]* and requires operation *mobility information [CM]*. This guarantees the inclusion of unit *Mobility Information Center Milan* whenever the location is *CM*. Virtual service unit *Weather [location] Center Milan* serves a similar role for operation *weather forecast*.

C. A framework for service-tiles

In order to demonstrate how an analysis technique based on service tiles can be integrated in a real service-oriented application, we implemented a framework based on *SCENE* [8]. *SCENE* provides a runtime execution environment to execute service compositions and capable of binding external services at runtime. At deployment time, a component analyzes the composition, identifies the cases in which dynamic rebinding had been foreseen by the designer, and generates some proxies that will manage these cases. By masking what services are actually invoked, the proxies enable a transparent dynamic replacement of services without affecting the workflow.

The standard *SCENE* environment provides proxies only for the part of the composition executed by a centralized orchestrator, while invoked services are treated as black boxes. We extend this basic setting by enabling the creation of proxies local to any service. More precisely, we create a proxy for every requested operation; the proxy is responsible for the selection and binding of the corresponding offered operation of some other unit.

Figure 3 sketches the framework architecture for the case study presented in Section II. Notice that proxies are created for requested operations such as the *mobility information SCENE* proxy. When *Weather forecast Center Milan* is selected, the proxy receives requests from *Mobility Information Center Milan* and forwards them to the currently bound service. Whenever *Weather forecast Center Milan* is unavailable, an alternative binding with *Weather forecast Milan* is established. In this case requests are forwarded to the latter unit, while *Mobility Information Center Milan* remains unaware of the change.

V. Evaluation

We implemented a simple tool to input and solve the service-building problem as defined above. Our prototype is based on a simple DSL (domain specific language), which is used to easily define the problem instance and its basic constraints,[4] such as needed services and their cardinality. The actual set of service-tile system model is computed and translated into a GNU MathProg script (a subset of AMPL [10]), which is then interpreted by GLPK (the GNU Linear Programming Kit [11]). We implemented both the DSL and the GLPK translator and interface in Common Lisp.

We experimented with the example of Section II. The basic mapping between elements of the example and tiles is straightforward (see Figure 1). The initial set of services is chosen simply to correspond to *Commuter Buddy*, while the set $\mathcal{U}$ is based on the content of Table II. These sets were augmented with some *virtual tiles* to represent the context. The preliminary experimental results with the tool are quite encouraging. Solutions to the example from Section II can be found in negligible time and require a minimal amount of memory (0.6 Mb). The experiments were run on a PC equipped with AMD Athlon 64 X2 4600+, 4 Gb RAM, Linux OS. The GLPK tool used was version 4.29.

We also performed additional experiments to push the limits of our simple prototype. To this end, we devised an abstract complex model, parametric with respect to some sets of numbers. By changing these parameters we were able to generate sets of more than 30,000 different tiles. The total time used by GPLK, the time to solve the actual model after it is built, and the memory usage are reported in Table IV. The leftmost column in the table represents the number of service units available when the problem solution is computed (i.e., the size of the set $\mathcal{U}$). In these experiments we assumed that all the services available in a given moment would be used to try to build up the solution. The experiments show that the bottleneck in the solution process is the computation of the model in memory: in the last example the constraint matrix alone contained more than 450 million elements, and GLPK had to use almost all the available memory. On the other hand, the actual model solving time is usually a small fraction, less than 4 seconds for the biggest example.

The real world examples that we may consider, such as our trip-planning application, exhibit usually much smaller sets of tiles. In practice, not all available services are usually considered when building a service assembly; instead, a filtering phase usually precedes the actual computation of a solution. In common practice, in fact, a designer builds a set of candidates for her assembly by querying some registries according to cost or functionality criteria. This reduces the candidate set size before calling the solver, in a situation in which the service building problem with our model can be solved with small overhead.

[4] A DSL model describing the example in Section II is available at: http://home.dei.polimi.it/cavallaro/tiles-experiments.html. The downloadable material includes the prototype translator to MathProg and the data used for the experimental evaluation.

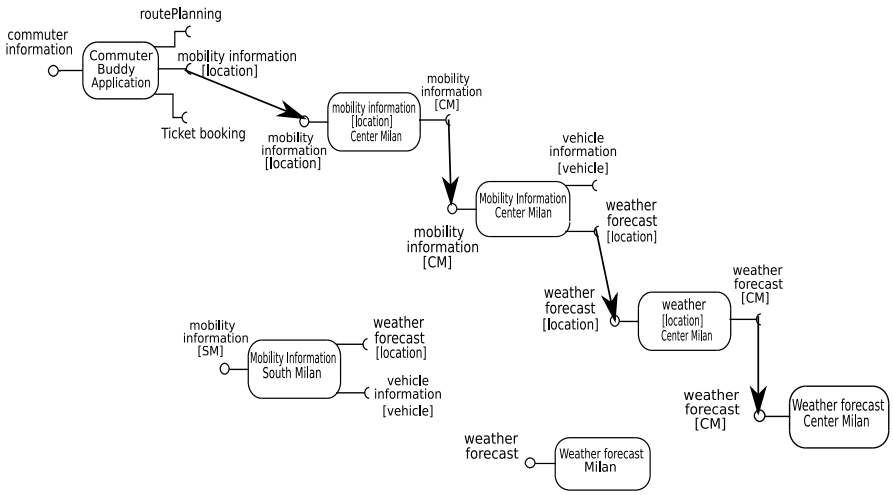

Fig. 2: An example of virtual service units to represent context for Mobility Information and Weather Forecast

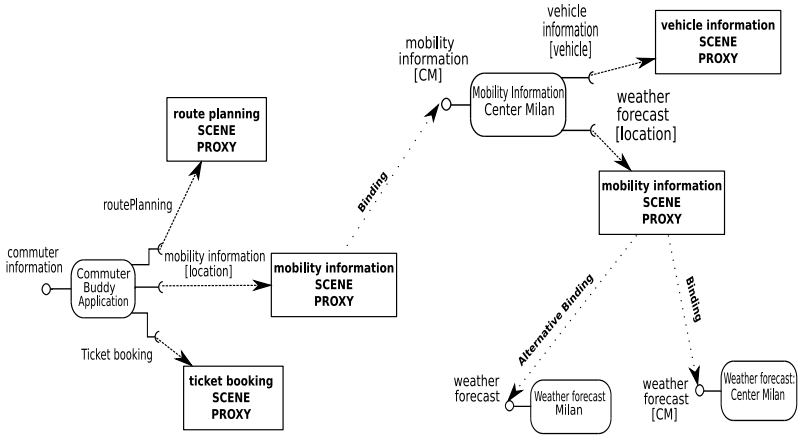

Fig. 3: An example of use of *SCENE* proxies in service tiles framework

# OF TILES	TOTAL GLPK TIME (S)	SOLVING TIME (S)	MEMORY (MB)
122	0.1	$\sim$ 0.0	1.4
677	1.3	$\sim$ 0.0	14.0
2602	7.1	0.1	95.2
10202	34.5	0.9	701.8
22802	119.6	2.5	2307.5
30302	183.1	3.8	3556.9

TABLE IV: Results of some experiments on the complex parametric example.

VI. RELATED WORK

The state-of-the-art can be partitioned according to the techniques that are used to model and analyze service systems: some approaches are model-driven, others are based on optimization techniques, and others borrow from the artificial intelligence repertoire.

In model-driven approaches, the developer designs the configurations that the system can go through, possibly assisted by some generative or verification tools. Model-driven approaches work under the assumption that relevant information about system behavior and its possible reconfigurations are available at design time, and can be formalized into a model which is used to guide the system through reactions due to component failures or changes in the application context. [12] presents the PLASTIC approach to develop self-adaptive services able to respond to changes in the application context (defined by users preferences) and to maintain certain levels of quality-o service. With PLASTIC, the system is designed by building a model. From this model, several variants of the service code can be generated. Different variants are used to support adaptation in response to changes of context or quality-of-service requirements. The current PLASTIC framework provides adaptation at discovery time (i.e., the deployed application is customized at binding time) but not at runtime. The service-tile model frames the compositional aspect of building self-adaptive service-oriented systems and enables re-configuration at runtime. Consequently, an extension of the PLASTIC framework with service tiles would allow applications to be reconfigured both at design time and at runtime. In [13] and [14] two model-

driven approaches are presented. They require to include all the possible variation points (i.e., aspects in which the system can change its configuration in order to perform adaptation) in the model at design time. This limits the situations the system can react to at runtime, since the number of configurations that need to be explicitly enumerated at design time can be hardly handled by a human developer. Service tiles also offer a model-driven approach to design self-adaptive service oriented applications, focusing on reducing the amount of information to be elicited at design time. In fact, service-tile models allow to build a system designing a part of it (i.e. the initial set), while the rest of the system is determined using the self-assembly capability of the tile-based system. As showed in the paper, this technique can be used also to find possible reconfigurations for the application, without having the designer to explicitly enumerate them at design time.

Optimization-based techniques, such as linear programming or genetic algorithms, are used to select a proper assembly of services to be invoked. At runtime, if one or more services fail or the quality of service of the composition becomes inadequate re-optimization is exploited to react to the situation. Linear programming based approaches can be found in [15] and [16]. These approaches model the service selection problem as an optimization problem: they model a workflow in terms of the operations it requires, define a complex quality-of-service model, and solve optimization problems to find the assembly of services to be allocated for each of the tasks, so as to maximize the overall quality-of-service. Service-tile models are also implemented using integer linear programming; however, their focus more on providing a flexible tool to design self-adaptive service-oriented systems than on optimizing the binding according to a sophisticated quality-of-service model.

In [17] the authors present a genetic-algorithm-based and quality-of-service-constrained service selection mechanism. In this approach, binding of a service-oriented system with a set of services, which meet some non-functional constraints, can be performed at runtime. This solution allows computing and reconfiguring only the portion of the system affected by changes. Unfortunately, genetic algorithms are usually computationally expensive. This prevents their usage at runtime for dynamic reconfiguration.

Artificial intelligence based techniques are also used to determine a proper service assembly, typically by encoding service selection as a planning problem. Both the approaches in [18] and [19] use planning techniques to determine which service assemblies can fulfill some goals specified by a developer at design time. The reconfiguration can take place both ad design time and at runtime, allowing also subsets of the system to be reconfigured. The approaches depend on the hypothesis that a centralized executor will orchestrate the process execution and will take care of reconfigurations when needed. Applications built under this hypothesis fail to provide suitable reconfigurations in scenarios similar to that presented in Section II where invoked services are needed to reorganize autonomously. Service tiles overcome this limitation modeling system component services in terms of offered and required operations.

[20] also specifically addresses the problem of building component-based applications, with a goal-oriented approach. Components are selected by adaptive planning (i.e. by a selection strategy that varies in response to the system's variability). This solution, however, does not focus on the distributed application domain, while the service-tiles model is more flexible in this respect.

VII. Discussion

Service-based applications rely on compositions of services which should be selected according to context and which are out of developer's control. This entails that the criteria according to which they are selected cannot be totally foreseen at design time. This was the case of the example of Section II, where the application should invoke an external service to retrieve mobility information for the current city zone: the application should be able to retrieve information for all city zones, bu the developer does not necessarily have a complete knowledge of which services are available in which city zone.

The service-tile model helps to build an application having only a partial vision of the system at design time: the designer focuses on a limited subset of the system, while the self-assembly capability will find a suitable instantiation of the other components. In the example, this implies that the developer only specifies which service operations are required by the *Commuter Buddy* application (i.e. specifies the initial set I) and chooses a desired cost. When the needed context information is available (likely at runtime in our example) a registry is queried to retrieve the available services (i.e. determines the set $\mathcal{U}$). Finally, an assembly satisfying the given constraints is determined by calling an ILP solver (see Section IV).

The context information can change during the application life-cycle or some of the used services can fail. Service-tile models can help managing context information, too: the simplicity of the model makes it possible to automatically recompute the assembly when needed, introducing the new constraints. This amounts to remove from $\mathcal{U}$ the failing services, while I is unchanged. Consider, for instance, the service units that provide information about the mobility in our running example. All of these service units receive information about the weather and gather data from vehicles. Unit *Mobility Information Center Milan*, which is required by the *Commuter Buddy Application*, requires operations *weather forecast* and *vehicle information*. If the units providing these operations fail, *Mobility Information Center Milan* would also fail. However, it would be sufficient to replace *Weather Forecast Center Milan* with the other service unit *Weather Forecast Milan* to restore functionality of the whole system (possibly with a lower quality of service). This new binding is local to the failed unit and is easily recomputed with the tile-based model.

Finally, service tiles represent a service in terms of offered and required services; in this respect, they can help

enforcing distributed service coordination and self-adaptation. This breaks, to some extent, the "traditional" black-box view of service applications; however, it does not significantly complicates the view of the system, because the designer needs only to specify an initial subset of services (I), while leaving to the self-assembly capability of the tiling system to determine the rest.

VIII. Conclusion

We presented a new approach to build self-adaptive service-oriented system. The main contributions of this paper are:

- A theoretical framework for modeling self-adaptive service-oriented systems based on *service tiles*.
- A proof-of-concept tool used to demonstrate the feasibility of our approach.

Preliminary experiments showed encouraging results and pointed out at the following advantages:

- The development is simplified, by allowing the specification of systems at component level (abstracting away low-level details);
- The solution of combinatorial problem arising in service composition scenarios is achievable with acceptable overhead even in large-sized examples;
- The enforcement of distributed self-adaptiveness is better supported.

The model and framework we developed are part of ongoing work, and tackling other significant aspects belongs to future work. Among them, let us mention the problem of developing a rigorous process to define labels associated to service requests and offers. We plan to borrow relevant notions from the literature dealing with similar problems, such as goal models using wide-spread methodologies such as i* [21] or Kaos [22].

References

[1] E. Winfree, F. Liu, L. Wenzler, and N. C. Seeman, "Design and self-assembly of two-dimensional DNA crystals," *Nature*, vol. 394, pp. 539–544, 1998.

[2] D. Giammaresi and A. Restivo, "Two-dimensional languages," in *Handbook of Formal Languages, Vol. 3, Beyond Words*. Springer, 1997.

[3] L. de Prophetis and S. Varricchio, "Recognizability of rectangular pictures by Wang systems," *Journal of Automata, Languages and Combinatorics*, vol. 2, no. 4, pp. 269–284, 1997.

[4] M. Pradella and S. Crespi Reghizzi, "A SAT-based parser and completer for pictures specified by tiling," *Pattern Recognition*, vol. 41, pp. 555–566, 2008.

[5] Y. Brun, "A discreet, fault-tolerant, and scalable software architectural style for internet-sized networks," in *In Proceedings of ICSE COMPANION*, 2007.

[6] Ö. Babaoglu, G. Canright, A. Deutsch, G. D. Caro, F. Ducatelle, L. M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes, "Design patterns from biology for distributed computing," *TAAS*, vol. 1, no. 1, pp. 26–66, 2006.

[7] M. Colombo, E. Di Nitto, M. Di Penta, D. Distante, and M. Zuccalà, "Speaking a common language: A conceptual model for describing service-oriented systems," in *In Proceedings of ICSOC*, 2005.

[8] M. Colombo, E. Di Nitto, and M. Mauri, "SCENE: A service composition execution environment supporting dynamic changes disciplined through rules," in *In Proceedings of ICSOC*, 2006.

[9] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & sons, 1998.

[10] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

[11] GNU Project, "GNU linear programming kit (GLPK), http://www.gnu.org/software/glpk/."

[12] M. Autili, P. D. Benedetto, and P. Inverardi, "Context-aware adaptive services: The plastic approach," in *In Proceedings of FASE*, 2009.

[13] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *In Proceedings of ICSE*, 2006.

[14] N. Bencomo and G. S. Blair, "Using architecture models to support the generation and operation of component-based adaptive systems," in *In Proceedings of Software Engineering for Self-Adaptive Systems*, 2009.

[15] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions Software Engineering*, vol. 30, no. 5, pp. 311–327, 2004.

[16] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions Software Engineering*, vol. 33, no. 6, pp. 369–384, June 2007.

[17] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *In Proceedings of GECCO*, 2005.

[18] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso, "ASTRO: Supporting composition and execution of web services," in *In Proceedings of ICSOC*, 2005.

[19] A. Lazovik, M. Aiello, and M. P. Papazoglou, "Planning and monitoring the execution of web service requests," *International Journal on Digital Libraries*, vol. 6, no. 3, pp. 235–246, 2006.

[20] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "A case study in goal-driven architectural adaptation," in *In Proceedings of Software Engineering for Self-Adaptive Systems*, 2009.

[21] E. S.-K. Yu, "Modelling strategic relationships for process reengineering," Ph.D. dissertation, Toronto, Ont., Canada, Canada, 1996.

[22] A. van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *In Proceedings of RE*, 2001.

IX. Appendix

This section proves the NP-completeness of the service-building problem introduced in Definition 3. The ILP encoding of the service-building problem shown in Section IV-A proves that the problem belongs to complexity class NP. Hence in this section we are left with proving the problem is NP-hard.

First, let us consider a special class of service-building problems where the cost bound K is taken to be a vector of ∞ values; correspondingly, the cost vectors of every unit become irrelevant as long as it is not zero, hence without loss of generality we set them to be the singleton $\{1\}$ and we omit considering it in the following proofs. The resulting special class of problem instances is called *flat service-building problem*. It is obvious that the NP-hardness of the flat service-building problem entails the NP-hardness of the (standard) service-building problem of Definition 3.

Then, we introduce a generalization of the flat service-building problem where service units are defined by *multi-sets* O and R rather than plain sets; this generalized problem is called *multi service-building problem*. Let us show that it is at least as complex as the flat service-building problem.

Lemma 4. *If the multi service-building problem is NP-hard, then the flat service-building problem is also NP-hard.*

Proof: A poly-time reduction of the multi service-building problem to the flat service-building problem will prove the statement.

In the following reduction, primed items denote items of a generic multi service-building problem, whereas unprimed items denote items of the corresponding flat service-building problem. Let us start by setting $\Sigma = \Sigma' = \{x_1, \ldots, x_m\}$. Then, for every $U \in \mathcal{U}'$, let n_U be the maximum multiplicity of any element in $U|_R$ or $U|_O$; correspondingly we add the n_U variables $y_U^1, \ldots, y_U^{n_U}$ to Σ. Correspondingly, n_U service units $V_U^1, \ldots, V_U^{n_U}$ are added to $\mathcal{U}$ as follows. For $1 \leq i \leq n_U$, V_U^i denotes the service unit with: (i) $x_j \in V_U^i|_O$ iff $U_O(x_j) \geq i$; (ii) $x_j \in V_U^i|_R$ iff $U|_R(x_j) \geq i$; (iii) $y_U^i \in V_U^i|_O$; (iv) if $i < n_U$, $y_U^{i+1} \in V_U^i|_R$; and (v) if $i > 1$, $y_U^{i-1} \in V_U^i|_R$. Finally, for all $1 \leq j \leq |I'|$: $I_j(V_U^1) = I'_j(U)$ for every U and $I_j(V_U^i) = 0$ for all $i > 0$.

The soundness of the reduction follows from the fact that, whenever a unit V_U^i is chosen, its requests over services y_k can be satisfied only by including all sibling units V_U^k for all $1 \leq k \neq i \leq n_U$. Together, these n_U units represent precisely the requests and offers of the original multi-set unit U. Notice that the reduction is poly-time only if we assume a unary encoding of multiplicities n_U. ∎

We are now ready to prove NP-hardness of the multi service-building problem.

Lemma 5. *The multi service-building problem is NP-hard.*

Proof: We provide a poly-time reduction of the set-covering problem to the multi service-building problem. The *set-covering problem* is as follows: given a family $F = \{Y_1, \ldots, Y_n\}$ of subsets of a finite set Y, and a budget b, determine if there exists a collection of at most b sets in F whose union is Y.

Given F, Y, b we build a corresponding instance of multi service-building problem. Let $\Sigma = Y \cup \{c\}$; without loss of generality assume $c \notin Y$. For every $1 \leq i \leq n$ we define service unit V_i as follows: $V_i|_O(y) = 1$ for all $y \in Y_i$ and 0 otherwise; and $V_i|_R(c) = 1$ and 0 otherwise. In addition, let V be the service unit defined as follows: $V|_R(y) = 1$ for all $y \in Y$ and 0 otherwise; and $V|_O(c) = b$ and 0 otherwise. Finally, let $\mathcal{U}$ be $V \cup \bigcup_i V_i$ and $I = \{\overline{I}\}$, where $\overline{I}(V) = 1$, and $\overline{I}(V_i) = 0$ for all i.

Let us show that the answer to the set-covering problem is affirmative iff the corresponding instance of the multi service-building problem has an affirmative answer.

The left-to-right implication is straightforward: given a collection Z of at most b sets in F whose union is Y, the set of service units $\{V_i \mid Y_i \in Z\} \cup \{V\}$ is such that all requests are balanced out by offers.

For the right-to-left implication let us first assume that there are no repeated elements in the solution T to the multi service-building problem (i.e., T is a regular set). In this case it is clear that the collection of sets $Z = \{Y_i \mid V_i \in T\}$ is a solution to the set-covering problem, as: (i) $\bigcup_{z \in Z} z = Y$ because every request of services in Y is satisfied by some unit, and (ii) $|Z| \leq b$ because every request for service c by every unit V_i is satisfied by some of the b offers of service c in V.

Next, assume that the solution to the multi service-building has a single instance of V. Then, the total number of V_i's cannot be more than b because otherwise not all requests of service c in the chosen V_i would be satisfied. Now, assume that some entry occurs more than once, i.e., $T(V_i) > 1$ for some V_i. As far as satisfying requests for services from Y in V is concerned, additional instances of V_i are useless because they have precisely the same services and every request for services in Y occurs exactly once in V. Hence, if we let $T(V_i) = 1$ we still have a valid solution. In all we reduce to the previous case of no repeated elements in T.

Finally, consider the case in which more than one instance of V appears in T. In this case, reasoning similarly as in the previous paragraph, we can actually consider only a single instance of V with all related V_i's and still have a valid solution.

Notice that even if b is encoded in unary, the reduction shows NP-hardness because the set-covering problem is *strongly* NP-hard. ∎

In all we have the desired result.

Corollary 6. *The service-building problem is NP-complete.*

Proof: The encoding of Section IV-A proves that the service-building problem is in NP. NP-hardness is shown as follows: Lemmas 5 and 4 prove that the flat service-building problem is NP-hard; then any NP problem is reducible to the flat service-building problem, which is easily reducible to the service-building problem of Definition 3 as it is a special case of it. ∎