Satisfying Requirements for Pervasive Service Compositions

Luca Cavallaro, Pete Sawyer, Daniel Sykes, Nelly Bencomo, Valérie Issarny

Satisfying Requirements for Pervasive Service Compositions

Luca Cavallaro
LERO, Ireland
luca.cavallaro@lero.ie

Pete Sawyer
Lancaster University, UK
sawyer@comp.lancs.ac.uk

Daniel Sykes, Nelly Bencomo, Valérie Issarny
Inria, France
first.last@inria.fr

ABSTRACT

Pervasive environments are characterised by highly heterogeneous services and mobile devices with dynamic availability. Approaches such as that proposed by the Connect project provide means to enable such systems to be discovered and composed, through mediation where necessary. As services appear and disappear, the set of feasible compositions changes. In such a pervasive environment, a designer encounters two related challenges: what goals it is reasonable to pursue in the current context and how to use the services presently available to achieve his goals. This paper proposes an approach to design service compositions, facilitating an interactive process to find the trade-off between the possible and the desirable. Following our approach, the system finds at runtime, where possible, compositions related to the developer's requirements. This process can realise the intent the developer specifies at design time, taking into account the services available at runtime, without a prohibitive level of pre-specification, inappropriate for such dynamic environments.

1. INTRODUCTION

Service oriented applications allow programmers to build distributed applications (i.e. *service compositions*) by putting together existing services developed by third-party providers. The newly-built compositions are meant to satisfy complex *application goals* that single services are not capable of addressing. The main issue that can arise with a service composition is that services are issued and controlled by third parties, and consequently their life cycle cannot be controlled by the composition designer. In this scenario, one or more services, selected by a designer to be bound and invoked by the composition, could misbehave or become unavailable at runtime. When that happens, it is necessary to replace the failed service with one available for discovery, binding and invocation.

The above makes the task of a composition designer particularly hard and, therefore, we have identified two issues. On the one hand, he has to *identify the application goals* and to design a proper workflow for the composition. On the other hand, he has to *select the services* that will realise the designed workflow, attempting to foresee which services will be available at runtime. The latter task is not always successful at design time, due to the lack of control over the services, and the lack of standardisation in service interfaces and behaviours.

To address the two aforementioned issues, existing literature proposes several frameworks to design service compositions starting from application goals (see for instance [1, 2, 3, 4]), or to enable the application to tolerate variability of the interface and the behaviour of the invoked services (see for instance [5, 6, 7]). However, the addressing of the two issues together through a uniform framework has been only considered in [8], where, the service composition is specified through the BPEL workflow language.

Similarly to [8], we propose a runtime framework to bring together the identification of application goals and the selection of services to be invoked. We use the framework to support the methodology we propose in this paper. Our methodology mandates that a composition designer specify the application goals at design time, using a *novel application of the KAOS goal model* [9]. The so specified compositions goals are then used by our framework, at runtime, to automate the service selection step, by discovering, binding and invoking those services that are available. The runtime framework relies on some of the authors' previous works in [2] and in [6], to discover and select a set of services capable of realising the application goals and to deal with the possible differences in their interfaces or behaviours. We here provide the details of the goal model and we demonstrate how the methodology works on a realistic case study. This case study involves a traveller, equipped with a smart phone, who has arrived in an unfamiliar foreign city, Paris for instance. The application's goal is to help the traveler to take part in some leisure or cultural activity such as visiting a theatre or riding a rollercoaster. We generalise this goal as *visiting some attraction*. Being unfamiliar with the city attractions, services and transport infrastructure, the traveller would like the application, installed on his smartphone, to find and organise such activities. The *CityGuru* application is designed to satisfy the *visiting some attraction* goal and is used as a case study to illustrate our framework.

In this paper, we will mainly focus on combining goal models with runtime composition. Tackling it requires *a)* the identication of the application goals, using goal models in our case; *b)* the maintenance of such goals at runtime, and *c)* the ability to update and change such runtime goal models according to changes of the user's preferences and status of services during runtime. We cover in this paper a) and b), by proposing a goal model based on KAOS, and building a runtime framework obtained combining some of the authors' previous works presented in [2, 6]. The ability to update and change the goal model at runtime is left for future work, as it is outside our scope.

The rest of the paper is organised as follows: Section 2 provides an overview of our design methodology and how it is

integrated with our runtime framework, Section 3 presents our goal model, demonstrating its application on our case study, Section 4 presents our runtime framework, explaining how the goals specified at design time are used as a runtime model to drive service discovery and composition, Section 5 provides an analysis of the related work, finally Section 6 draws some conclusions and introduces some future research directions.

2. METHODOLOGY OVERVIEW

Designing a service composition in a pervasive environment poses to the composition designer the problem of specifying the services that need to be bound and used by his application, without the information about the actual availability of those services at runtime, when the composition will be used. New services may become available at any time, and known services may cease to be available. This calls for the assumptions that services can be discovered at runtime, and that they can be developed independently and thus may exhibit incompatibilities in their interfaces and in their behaviours.

Our methodology aims at bridging the gap between the available and the needed information. On the one hand, it allows the designer to specify the composition, in terms of the goals it should fulfil. On the other hand, it uses the goals specified at design time as a *runtime model*. This model is used by the framework we explain in Section 4 to *discover* an appropriate set of services that realise the designer's goals, and to compose them in a working service composition. This composition step is performed through the use of ontology-based techniques. We make the hypothesis that the ontology used in our framework is one agreed upon by developers in a particular domain, and that it defines concepts for all the relevant operations and composite functionality.

Critical to the establishment of such compositions is the adequate description of available services. To this end, we introduce the *networked system model* that comprises descriptions of *a)* The high-level functionality of the service expressed as a *capability*. A capability refers to a concept (which in Connect we call an *affordance*) in a domain ontology giving the semantics of the functionality, and is either *provided* (by the service to others) or *required* (by the service from others). *b)* The interface (API) of the service, with each operation and data parameter annotated with a concept from a domain ontology. Normally WSDL is used. *c)* The behaviour of the service expressed as a labelled transition system in which actions are operations from the interface.

Our runtime framework comprises several *enablers*, each one is in charged of the handling a part of the composition process:

a) **The discovery enabler** exhibits a plugin architecture that enables it to use various dynamic discovery protocols (such as WS-Discovery and UPNP) to populate a repository of descriptions of services available on a network. These descriptions follow the form of the networked system model given above. Upon discovery of a new service, this enabler also find pairs of services that have compatible affordances (normally one provided and one required). *b)* **The learning enabler** is invoked (by discovery) when the description furnished by a particular discovery protocol does not include all the required detail. Both the affordance and the behaviour can be learned given an interface [10, 11]. *c)* **The synthesis enabler** is invoked (by discovery) when a pair of services are found with compatible affordances. The synthesis enabler analyses the behaviour of the pair of systems alongside a specific goal for the pair and, if necessary, generates a mediator. This mediator is composed with the services (creating a *connected system*), enabling the two services to interact in such a way as to achieve the goal. *d)* **The deployment enabler** is called by the synthesis enabler to execute the abstract mediator and hence perform the interaction between the services [12].

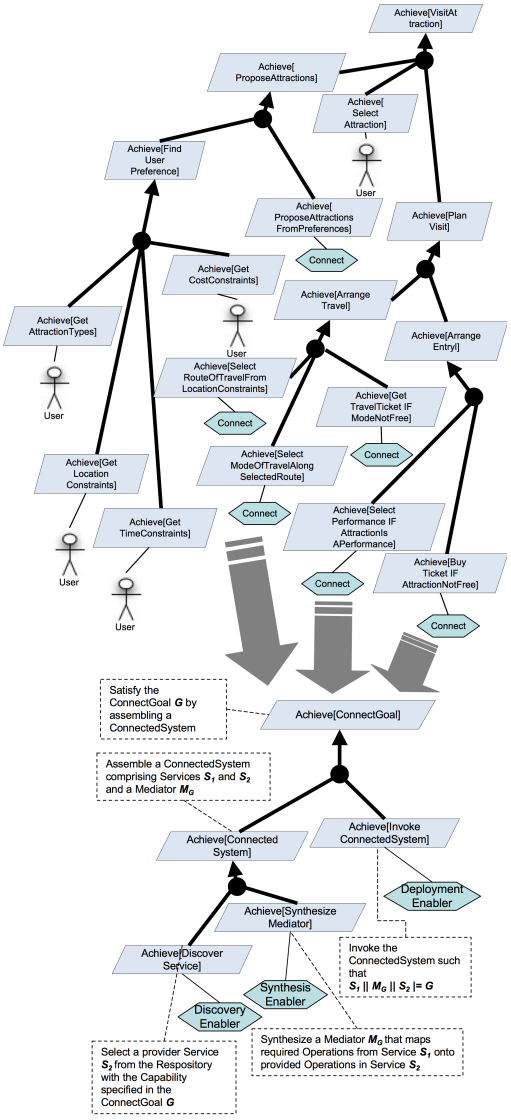


Figure 1: Connect Goals

In this paper we concentrate on the enablers most concerned with the user goal, the discovery and synthesis enablers.

3. A GOAL MODEL FOR COMPOSING PERVASIVE SERVICES

We use the KAOS method [9] to represent the application's requirements. First, we need to represent the overall goals of the application and refine these *application goals* to the point where we are able to assign responsibility for their sat-

isfaction, either to human actors (in which case the goals are, in KAOS terms, *expectations*), or to software components or services (in which case the goals are KAOS *requirements*).
The top part of Figure 1 shows a KAOS goal model for the CityGuru composition in which the root goal, expressed as *Achieve[VisitAttraction]*, is to enable a user to visit an attraction of interest in a city. An attraction may be any cultural or entertainment institution, such as a museum or theatre, or a specific event such as particular exhibition or a theatre performance. This top-level goal is satisfied by proposing a set of attractions that satisfy some preferences, such as interest, time, cost, etc., constraints set by the user; selection of one of the proposed attractions; and planning the user's visit to the attraction. Some of the (sub-)goals in the Figure, such as selecting an attraction from the set of those proposed, ultimately decompose to expectations on the user. Others, such as proposing a set of attractions consistent with the user's constraints, decompose to requirements on Connect.
We refer to requirements on Connect as *Connect Goals*. The Connect system acts here as a proxy for a service that will ultimately satisfy the goal, e.g. *Achieve[BuyTicket IF AttractionNotFree]*, because such a service can only be identified dynamically.
The bottom part of Figure 1 shows a KAOS goal model of Connect that represents how a Connect Goal is satisfied by discovering a service, synthesising a mediator and then invoking the resultant connected system. Responsibility for satisfying these three KAOS requirements is allocated to the Discovery, Synthesis and Deployment Enablers. The relationship between Connect Goals and Services is modelled by the object model in Figure 2.

A Connect Goal is expressed formally as a temporal logic formula:

DEFINITION 1 (GOAL FORMULA). *A goal G is a formula of linear temporal logic (in particular CLTLB(D), a LTL extension that allows us to predicate on the data flow between communicating services, as described in [13]). over the behaviour of the connected services such that $S_1 \| M_G \| S_2 \models G$ for some services S_1, S_2 and synthesised mediator M_G. The goal language makes use of standard temporal logic operators:*

- `[]` ***f*** *– formula f must hold in all (future) states.*
- `<>` ***f*** *– formula f must eventually hold in some (future) state.*
- `X` ***f*** *– formula f must hold in the next state.*
- ***f*** `U` ***g*** *– formula f must hold until formula g holds.*
- `||`, `&&`, `!`, `->`, `<->` *– with their usual meanings in propositional logic.*

Three predicates can be combined with the above operators:

- `executed(c)` *– the operation represented by concept* `c` *must have been executed.*
- `received(c)` *– the data (input or output) represented by concept* `c` *must have been received.*
- `sent(c)` *– the data (input or output) represented by concept* `c` *must have been sent.*

The concepts `c` *above are defined in the domain ontology as data or operations. Each such operation is associated (in the ontology) with another concept that determines the affordance associated with the goal.*

A typical goal expressed in this way could have the form `<>executed(desiredOperation)` (the desired operation must eventually be performed) or `<>(received(love) || received(money))` (love or money should eventually be received).
Consider for instance the goal (in Figure 1) *Achieve[SelectRouteOfTravelUsingLocationConstraints]*, from which we derive the goal formula: `<>received(StationLinesList)`. In this formula, "StationLinesList" is a concept in the ontology associated with the affordance "TransportProvider". From the repository of discovered services, we select those that provide or require this affordance, and attempt to synthesise a goal-satisfying mediator between their respective protocols.

4. RUNTIME FRAMEWORK FOR REALISING GOALS

The goals specified at design time by the composition designer define the intent of the system. This intent needs to be realised in order to build the composition. The realisation step consists in *discovering* those services that may satisfy one or more goals and *synthesising* a composition that combines some services, selected from among the discovered ones, and can enable their communication. The discovery and synthesis steps need to be performed at runtime because they need to take into account the services available in the moment in which they are performed.

4.1 Discovery

Given a goal formula G, the discovery enabler must first find the subset of services in its repository that have the potential to satisfy it by virtue of their implementing an appropriate affordance. Since G ordinarily contains a predicate such as `executed(op)`, discovery can look up `op` in the domain ontology. Each operation therein is associated with an affordance concept. Discovery uses this relation to find the affordances that include this operation. We call these the goal's affordances. The respository of discovered services is indexed by affordance, making it efficient to find the services providing or requiring the goal's affordances. If a matching pair is found, discovery initiates synthesis to generate a mediator for the pair of services and G.

4.2 Synthesis

Once a pair of services have been selected, we need to enable their communication. To do so, the automatic synthesis of a mediator for the pair is required. The process for generating a mediator takes as input two services, S_1 and S_2, and considers their alphabets Σ_{S_1} and Σ_{S_2}, their behaviours P_{S_1} and P_{S_2}, and an application goal G, specified using the model introduced in Section 3. This process produces a mediator M that enables the communication of the two services, as defined in [5], and is such that $P_{S_1} \times M \times P_{S_2} \models G$.
The process goes through two main steps:
a) Ontology matching: this step matches each operation in the interface of each of the services with a domain specific ontology. The matching information is used to *align* Σ_{S_1} and Σ_{S_2} with a common alphabet Σ_M.
b) Mediator synthesis: this step tries to determine, through SMT-based model checking, a mediator M that enables the communication of the two services and ensures the application goal is realised.

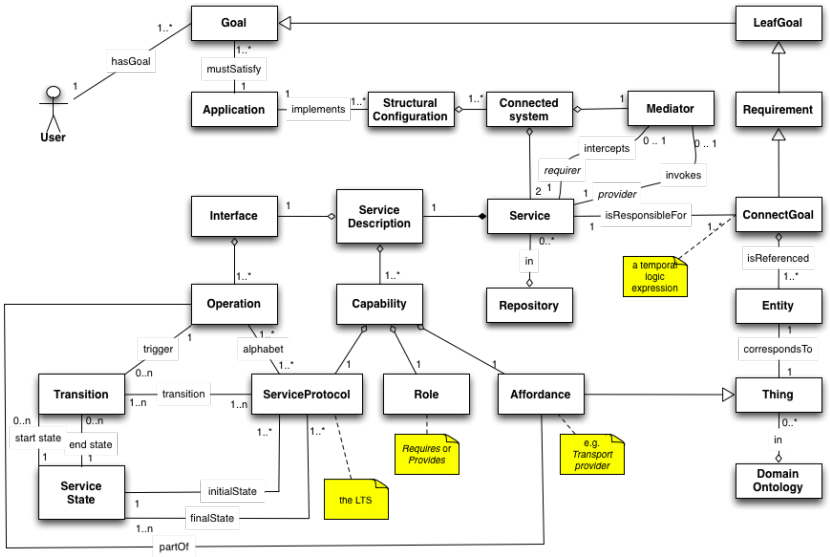


Figure 2: Connect Object Model

4.2.1 Ontology matching

The ontology *matching* step, as defined in [14], consists in *aligning* the alphabets of the required and provided protocols, in order to substitute them with a common alphabet Σ_{M_k}. In practical terms the alignment process considers pairs of service operation o_1, o_2, such as o_1 is in Σ_1 and o_2 is in Σ_2, and tries to find an ontology concept O_3, such that the concepts O_1 and O_2, used to annotate the two considered operations are both subclasses of O_3. The two considered operations' annotations are then substituted by the common superclass. Consider for instance the operation *startLocation* on the *TransportProvider* capability of CityGuru (see Figure 3). The operation provides input to the service for retrieving a route. For this reason *startLocation* extends the ontology concept *query*. Let us make the hypothesis that the discovery enabler found the RATP service[1], offering the *TransportProvider* capability, and that now the synthesis enabler should make the latter service communicate with CityGuru. The *TransportProvider* capability of the RATP service presents an operation annotated by the same *query* concept. Consequently we can align the operations on the query *concept*. The same rationale can be applied to the *endLocation*, *time*, *mode* and *query* operations of CityGuru. Let us now consider the input parameters of the aligned operations. The alignment process works similarly for them, plus, we consider that a parameter's semantics depends on the operation it is used as parameter for. Consider for instance the only *startLocation* input parameter. This parameter is associated with the ontology concept *depart* and so is the *depart* parameter of the operation *query* in the RATP interface. Since both the operations were aligned with the ontology concept *query*, we can align their *depart* parameters on *query.depart*. At the end of the process the *TransportProvider* capability of CityGuru and the corresponding capability of the RATP service will be aligned with a common alphabet Σ_M, and can be used as input for the mediator synthesis phase. For the sake of space we do not report more details about the ontology matching technique here, but we refer the reader to many of the approaches present in the literature (e.g. [15]).

[1]RATP is the Paris public transportation provider http://www.ratp.fr/

4.2.2 Mediator synthesis

A mediator is a piece of software that, given P_{S_1} and P_{S_2} is capable of ensuring, within the product $P_{S_1} \times M \times P_{S_2}$, the existence of a sequence of operations $\mathcal{R} = R_1, R_2, \ldots, R_k$ that reaches a final state. This sequence is such that each step $R_g = [r_1, r2]$, where: $r_1 = R_g|_1 \in P_1 \cup \{\epsilon\}$, and $r_2 = R_g|_2 \in P_2 \cup \{\epsilon\}$, with $1 \leq g \leq k$.

Intuitively, this happens when M guarantees that when an operation of r_2 is invoked, its input data have been provided by some already invoked operations r_1. Moreover, an operation of r_1 successfully finishes its computation if eventually all of its parameters are returned by some r_2 operations. An interaction respecting the aforementioned conditions can be termed *feasible*.

The rationale of this definition can be understood by thinking of the interaction of two services in terms of a client-service interaction. The process happens through the invocations of service operations performed by the client. When invoking a service operation the client provides the input parameters required by the operation and expects as output the return parameters provided by the operation (i.e. *progress* property). In order for the interaction to be completed successfully, on the one hand, each of the service operations should receive those input parameters it is expecting and, on the other hand, each client operation should receive the return parameters it expects (i.e. *consistency* property). Due to space limitation we omit here the formal definition of mediator and of feasible interaction and we refer the interested reader to [16].

Having intuitively defined a feasible interaction, we can say that $P_{S_1} \times M \times P_{S_2}$ satisfies some application goal G, if there exists a a sequence of operations $\mathcal{R}$, in which the application goal is verified. As pointed out in Definition 1, the application goal is a linear temporal logic formula, consequently it is satisfied if it is satisfied in R_1, the first step of the $\mathcal{R}$

sequence (see [13] for formal details of CLTLB(D) formula satisfaction).
In practical terms our problem of finding a mediator satisfying a given application goal is reduced to the problem of finding a feasible interaction between a client and a service that satisfies application goals. The mediator satisfying a goal is a subset of the mediator comprising all the possible feasible interactions. This, on one hand makes the mediator synthesis problem smaller, on the other hand ensures that the mediated communication between the two networked systems enforces a behaviour in which the user is interested.

4.3 Synthesis at work

To demonstrate how our synthesis enable works, we consider, for instance, the goal *Achieve[SelectRouteOfTravelUsingLocationConstraints]*, specified in Section 3, from which we derive the goal formula `<>received(StationLinesList)`. We are making the hypothesis that we selected the RATP service from the repository of discovered services, and we attempt to synthesise a goal-satisfying mediator between CityGuru and that service (see Figure 3b for their protocols representation).
In practical terms this means that we need to find a feasible interaction between CityGuru and the RATP service that realises the `<>received(StationLinesList)` goal.
An example of such an interaction can be the following:
$\mathcal{R}_{[CityGuru-RATP]} = ([startLocation(depart), \epsilon]$,
$[endLocation(arrival), \epsilon]$, $[time(time), \epsilon]$,
$[mode(mode), \epsilon]$, $[query() : stationsLinesList, \epsilon]$,
$[\epsilon, query(depart, arrival, time, mode) : stationsLinesList]$,
$[buyTicket(stationsLinesList, paymentfo) : ticket, \epsilon]$,
$[\epsilon, buyTicket(stationsLinesList, paymentInfo) : ticket])$.

This interaction is feasible, as in each step the progress and consistency properties hold, it arrives in a final state for both the behaviours, and the property expressed by the `<>received(StationLinesList)` goal holds.

5. RELATED WORK

In this section we provide two main streams of related work to support pervasive service composition: architectural composition of services and runtime requirements models.

Architectural composition of services. Sykes et al. [17] tackle a similar problem from a different angle. In previous work, they have outlined a 3-layer architecture for self-adaptive systems. At the lowest level, individual components tune their behaviour for performance. At the middle level, combinations of components are selected to compose a complete system. At the uppermost level, high-level adaptation decisions are made. As in our work, Sykes et al. argue that this uppermost planning be done using goal-based reasoning. Our work presents an implementation of a high-level goal-oriented adaptation mechanism that would sit at the uppermost level of their architecture, and uses run-time requirements models as opposed to written component descriptions.

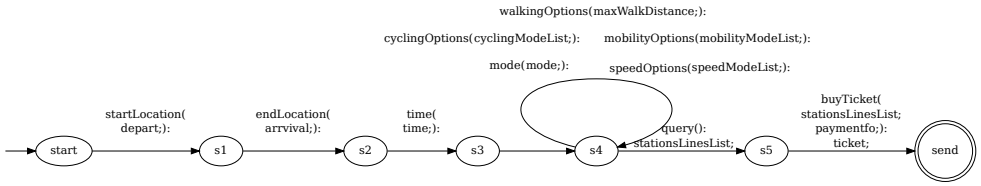
In [3, 18] two approaches to architectural composition of services are presented. The former approach presents the service-tiles framework for service composition, which allows designer to specify compositions in terms of offered and delegated requirements and then instantiates the so specified composition at runtime. In contrast to our work, this framework assumes that each label identifying an offered or delegated requirement, has a unique meaning in the whole composition (e.g. all services offering a trip planning service will be called *TripPlanner*). We overcome this limitation using ontologies. The second work describes a framework for declarative specification of service composition, in which the composition is instantiated at runtime starting from LTL declarative constraints imposed at design time. Our approach works similarly, but it uses a goal language for expressing the constraints and features a mediator synthesis step that in [18] is missing. An approach very similar to ours is presented in [8]. In this work a framework to realise at runtime a service composition starting from the goals a developer specifies at design time. Similarly to our work, this framework features a discovery and synthesis mechanism. The main difference of our work with respect to this paper is that here we focus on the development methodology.

Runtime requirements models. In [19] the authors argue that self-adaptive systems should be requirements-aware, i.e. the system should be able to deal with runtime representations of requirements. FLAGS [20] like in our case uses goal entities available at runtime. FLAGS aim is to approach self-adaptation. FLAGS dynamically updates the level of satisfaction of goals, uses the concept of adaptive goals to dynamically trigger adaptation actions, and propagates the corresponding changes to the underlying SOA-based implementation. Differently from our work, they concentrate on the design methodology only, plus they don't take into account the possibility of matching services that have interfaces and protocols that feature differences. Alrajeh et al.[21] apply machine learning and scenario specifications to goal models for elaborating requirements. Their approach implemented as a tool automates their design-time process. In contrast we are tackling runtime issues as well.

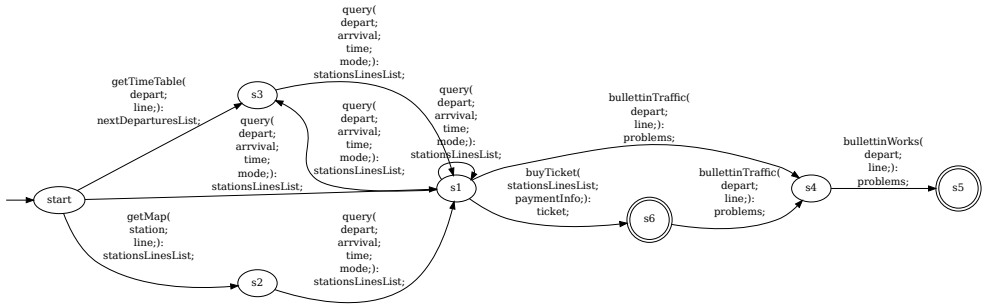
6. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a novel methodology to develop a ubiquitous service composition starting from its requirements. We defined a goal model to allow the elicitation of those requirements, extending the KAOS language, in Section 3. Our goal model is used, at runtime, to select those available services that can satisfy the specified requirements. This process is performed through the discovery, learning and synthesis enablers.

The work presented in this paper is in its preliminary phase and several possible directions are open for future works. Among the other we here mention the two following main points. *a*) Our methodology needs to be more accurately evaluated, applying it to other realistic case studies. *b*) In this paper we make the hypothesis that it is always possible to find a service composition *exactly* realising a goal model. In real world scenarios this might not always be the case, as the goals specified at design time can be only partially realisable. We plan to introduce in our model, and consequently into our enablers, a mean to tolerate slight deviations from the goals specified at design time, similarly to what has previously been done in [22, 20] . This can result in more user involvement, as the system can suggest to the user which application goals are realisable, according to services available at runtime.



(a) Protocol representation for CityGuru



(b) Protocol representation for RATP

Figure 3: Representation of two protocols used in the case study in Section 4.3.

Acknowledgment. This paper is partially found by the EU Connect project, the EU EternalS project and the EU Marie Curie project Requirements-aware systems.

7. REFERENCES

[1] V. Issarny, B. Steffen, B. Jonsson, G. S. Blair, P. Grace, M. Z. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta, "Connect challenges: Towards emergent connectors for eternal networked systems," in *ICECCS*, 2009.

[2] W. Heaven, D. Sykes, J. Magee, and J. Kramer, "A case study in goal-driven architectural adaptation," in *SEAMS*, 2009.

[3] L. Cavallaro, E. Di Nitto, C. Furia, and M. Pradella, "A tile-based approach for self-assembling service compositions," in *ICECCS*, 2010.

[4] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H. Cheng, "Goal-based modeling of dynamically adaptive system requirements," in *ECBS*, 2008.

[5] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 292–333, 1997.

[6] L. Cavallaro, E. Di Nitto, and M. Pradella, "An automatic approach to enable replacement of conversational services," in *ICSOC*, 2009.

[7] C. Canal, P. Poizat, and G. Salaün, "Model-based adaptation of behavioral mismatching components," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 546–563, 2008.

[8] P. Bertoli, M. Pistore, and P. Traverso, "Automated composition of web services via planning in asynchronous domains," *Artif. Intell.*, vol. 174, no. 3-4, pp. 316–361, 2010.

[9] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. John Wiley & Sons, 2009.

[10] A. Bennaceur, V. Issarny, R. Johansson, A. Moschitti, R. Spalazzese, and D. Sykes, "Automatic Service Categorisation through Machine Learning in Emergent Middleware," in *FMCO - Formal Methods for Components and Objects*, 2011.

[11] F. Howar, B. Steffen, and M. Merten, "Automata learning with automated alphabet abstraction refinement," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Jhala and D. Schmidt, Eds. Springer Berlin / Heidelberg, 2011, vol. 6538, pp. 263–277.

[12] D. Bromberg, P. Grace, and L. Réveillère, "Starlink: runtime interoperability between heterogeneous middleware protocols," in *ICDCS*, 2011.

[13] M. M. Bersani, A. Frigeri, A. Morzenti, M. Pradella, M. Rossi, and P. S. Pietro, "Bounded reachability for temporal logic over constraint systems," in *TIME*, 2010.

[14] A. Bennaceur, G. S. Blair, F. Chauvel, G. Huang, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, A. Pathak, R. Spalazzese, B. Steffen, and B. Souville, "Towards an architecture for runtime interoperability," in *ISoLA (2)*, 2010.

[15] E. Cimpian and A. Mocan, "Wsmx process mediation based on choreographies," in *Business Process Management Workshops*, 2005.

[16] M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi, "Smt-based verification of ltl specification with integer constraints and its application to runtime checking of service substitutability," in *SEFM*, 2010.

[17] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "Exploiting non-functional preferences in architectural adaptation for self-managed systems," in *SAC*, 2010.

[18] G. Cugola, C. Ghezzi, and L. S. Pinto, "Dsol: a declarative approach to self-adaptive service orchestrations," *Computing*, vol. 94, no. 7, pp. 579–617, 2012.

[19] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier, "Requirements reflection: requirements as runtime entities," in *ICSE (2)*, 2010.

[20] L. Baresi and L. Pasquale, "Fuzzy goals for requirements-driven adaptatio," in *RE*, 2010.

[21] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *ICSE*, 2009.

[22] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "Relax: a language to address uncertainty in self-adaptive systems requirement," *Requir. Eng.*, vol. 15, no. 2, pp. 177–196, 2010.