ASTES

*Advances in Science, Technology and Engineering Systems Journal Vol. 4, No. 4, 244-250 (2019)*

*www.astesj.com*

*Special Issue on Advancement in Engineering and Computer Science*

**ASTESJ**

**ISSN: 2415-6698**

# Automatic Service Orchestration for e-Health Application

Anatolii Petrenko, Bogdan Bulakh*

*System Design Department, National Technical University of Ukraine, Igor Sikorsky Kyiv Polytechnic Institute, 03056, Ukraine*

ARTICLE INFO

*Article history:*
*Received: 31 May, 2019*
*Accepted: 13 July, 2019*
*Online :30 July, 2019*

*Keywords:*
*Service orchestration*
*Semantic technologies*
*e-Health*

ABSTRACT

*This paper describes an architectural approach to the development of dynamic service-oriented systems for e-Health using the service orchestration mechanism and semantic technologies. The main idea is the dynamic synthesis of the complex functionality required by user or by software agent. This idea should help to build and easily extend the applied loose-coupled systems without strict dependencies on concrete web services and their invocation details. A sample scenario of such dynamic orchestration is covered and analyzed, possible ways of further improvement of this approach are given.*

## 1. Introduction

This paper is an extension of work originally presented at the 2018 IEEE First International Conference on System Analysis & Intelligent Computing (SAIC) [1]. Since then the original research has been continued in the direction of development of service-oriented e-Health application. This paper covers more details of dynamic service orchestration and how it can be applied to the e-Health software design.

## 2. Problem Overview

Modern network applications widely use web service interaction to retrieve or update data. There may be a lot of interconnected services of different types (like SOAP or REST) interacting with each other and external third-party services, thus forming so called "service ecosystem". The interaction of these services may be implemented in different ways (see Figure 1):

- "Hard-coded" interaction: direct invocation of the service at specific URL is fixed in source code with all the invocation details. In this case even simple relocation of services to another URLs requires changing (and consequent recompiling, redeployment, retesting) of all dependent software modules (other web services, for instance).
- Services could use service discovery mechanism by means of some registry (in its simplest case by using some "name server" to translate service name to concrete URL. UDDI registry is an advanced example). This makes it easier to modify and scale the service ecosystem, but services are still tightly coupled by data formats and interaction patterns.
- Service choreography approach (e.g. "publisher-subscriber" model) could be implemented by means of some message broker. This event-driven service architecture consists of loosely coupled services that can be dynamically connected to each other through a subscription to different message types.
- Service orchestration approach: all interactions are controlled by external orchestration software. In such system services could be truly agnostic to their environment while publishing only their interface description.

It must be noted that in many real-life systems composed of hundreds of services all of these approaches may be implemented within a single service ecosystem (forming "heterogeneous" service ecosystem).

In any case it is only skilled developer who is capable to organize the inter-service communication to bring some functionality to the system. It is almost never possible for the end users to create new functionality without skills and knowledge in programming. This problem is partly solved by so called "workflow management systems" (like scientific workflow systems [2] or engineering ones [3]). Most of service-oriented workflow systems typically rely on the orchestration approach.

* Bogdan Bulakh, Email: bogdan.bulakh@gmail.com

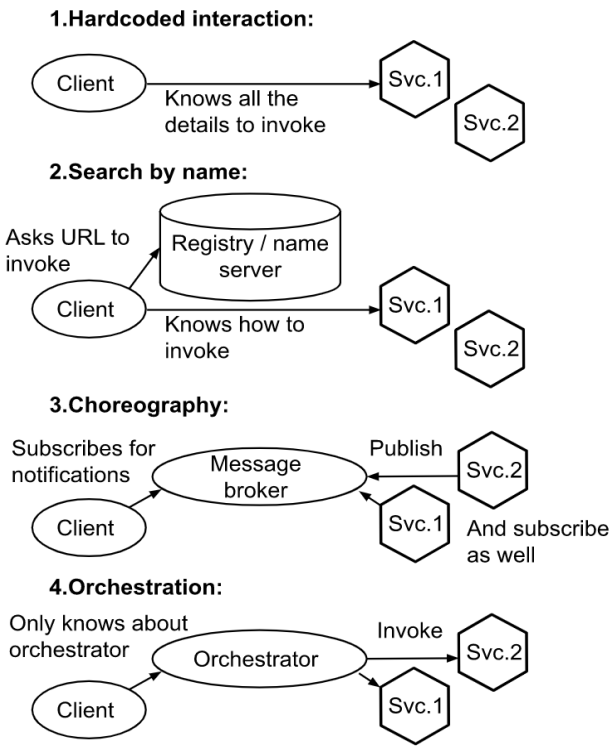


Figure 1: Some existing service interaction patterns (client could be implemented as another service as well)

### *2.1. Problem Definition*

This paper focuses on a slightly different issue: how to make it possible for *both users and program agents* to request and execute the desired functionality *without any service-level knowledge* (existence of specific services and their interfaces' details)? In other words, dependencies between the clients and web services are forcing developers to update clients when service ecosystem changes. This costs time and extra QA efforts. We propose an approach that helps to avoid the update of the clients in case of changes in service ecosystem and thus makes it easier to maintain and improve the overall system.

Instead of making calls to specific services, or even "constructing workflows of services", software clients can issue "functionality requests" to some broker. While these requests can be fulfilled, the clients need no updates. The fulfilment of these requests could be organized using the orchestration approach, if there is no single service capable to provide the requested functionality. This is what we call here a "*dynamic semantic-based orchestration*": the automatic synthesis of a service workflows to provide the requested functionality (including some goal and output data). One of the main benefits of the dynamic orchestration is that service clients and services themselves are not coupled in any way except the functionality semantics. There is no need to keep track of web service interface changes on each client, it's only a knowledge base about services (which binds functionality semantics with service invocation details) that should be kept updated instead.

## 3. Possible Application Scenarios

There are a lot of application fields for dynamic semantic-based orchestration. Some of them are listed below.

*Scientific and engineering workflows*

There are a lot of existing 'Scientific workflow systems' and many of them support invocation of the remote services (Taverna workflows [4], Apache Airavata [5] etc.). The main problem there is a high complexity of constructing the service workflows for non-IT-domain specialists and scientists, because the workflow elements are bound to concrete services, their operations, and data formats. With the dynamic semantic-based orchestration the end users could just set the goals of the orchestration and receive the automatically built workflow. No need to deal with service-level details for the end users.

*Performance monitoring dashboards*

This feature can be implemented in medical (see further), scientific, engineering, industry production, financial and other domains. Given the ecosystem of services to fetch different parameters of the objects to monitor (patients' health data, the status of lab equipment or industrial equipment etc.). User needs a tool that can be used to easily construct and customize the KPIs. Possible solution: a user describes the functionality he needs in terms of some knowledge base, and new dashboard indicator will get its values from the dynamically orchestrated services providing the functionality requested.

*Business analytics and data mining*

Same as previous kind of scenario but it also involves the services for mining new knowledge from existing data helping to improve existing business processes. These workflows are more complex, compute intensive and provide valuable information (insights). It is a service-oriented analytics solution for business enterprises of different scale. In [6] a workflow management system for 'big data' mining is described. But its workflows are still composed manually by the end user.

*Smart house and IoT*

The dynamic semantic-based service orchestration can be used in controlling IoT devices, e.g. as a part of smart house. For example, anyone in the smart house environment can easily add new automation scenario, e.g. to control lighting, climate or heating or water supply in a smart way, just by describing the desired functionality. In a similar way this approach can be applied, say, to control production processes on factories or for stock management. The PROtEUS++ [7] is a promising example of a specialized workflow engine for IoT tasks, and by the way it supports dynamic service discovery using the semantic service description.

*Workflows for e-Health*

Workflows for patient's diagnostics and treatment (DTWf) could also be implemented with the help of automatic semantic-based service orchestration. The DTWf typically can contain invocation of basic services for diagnostics (like service to check blood pressure, controlling blood pressure data received from devices or manually entered by the patient), services of patient's health status prediction (based on the data provided by other services) and so on [8]. This application field will be mainly referred further as it is studied by authors in scope of implementation of the project of the mobile e-Health platform development.

## 4. Implementation

The main idea of the proposed solution is to hide as much low-level service interface specifics as possible from the client (functionality consumer), allowing the client to operate only with its goals and data. It is similar to how declarative programming languages (like SQL or SPARQL) differ from imperative ones: when working in a declarative style you need to specify what goal to reach but not how it should be reached. You actually don't worry about implementation details. Ideally the client just declares the new computational goal (e.g. "to get specific output results from specific input data" etc.) without referencing particular services. He only operates with terms and facts from the knowledge base that can help him to express his goal. The rest process is automated by means of automatic service discovery and automated orchestration.

The solution proposed is based on the following three main components (also implemented as services), as shown on Figure 2:

- *Service for execution* (SE) of functionality requests, which serves as a kind of broker and the single point of access for clients. It does not only find some concrete web services able to fulfil the client's request, but also is responsible for execution of any service operations and results provision. In other words, clients don't ask SE to find some services, they ask to do some actions, provide input data and wait for output. In order to fulfill client's request SE communicates with other two components.
- *Service registry* (SR). This service searches for available services that could provide the requested functionality (according to registered service's semantic annotations). In fact, it is an interface to the knowledge base containing facts about services and their functions. The second mission of SR is to find the possible service orchestration scenario if no single service matches the requested functionality. Then this scenario is passed to the Service orchestrator.
- *Service orchestrator* (SO). This service is responsible for execution of service workflows, created by SR. Workflows actually can consist either of a single service, or describe the orchestration scenario involving multiple services. This service complies with a SaaS model: new workflows can be deployed in a cloud infrastructure and can be interacted via the REST interface (invoked, queried about status and results, canceled, disposed etc.). It should be noted that today there are a lot of orchestration tools available. The orchestrator engine is a core part of SO and it can be implemented on top of many of existing tools, for example: BPEL engine, Taverna Workflows, Netflix Conductor (a microservices orchestration engine).

### 4.1. Service Registry and Knowledge Base

The SR has an extended functionality compared to UDDI registries. Instead of binding to standard fields and database schema, it is based on flexible knowledge base. Administrator-level users can add new facts or modify existing facts about services registered and extend the ontology with new classes. The SR can be queried about services in a similar way like Triple stores are queried with SPARQL queries. But the functionality of SR is not limited by simple querying the triple store. It includes the matchmaking logic helping to construct the complex service workflow according to the goals set and inputs provided. To do this the SR should contain the formalized knowledge on web service interaction details (protocols, procedures, interfaces), and the knowledge on basic data formats used and how to extract and transform the data from them (xml, JSON, CSV etc.). That's why the knowledge base should consist of domain ontology (to set goals, to describe services functionality and operations, inputs and outputs), service ontology (to allow automatic invocation, see Figure 3), and data formats ontology (to allow automatic data flow building).

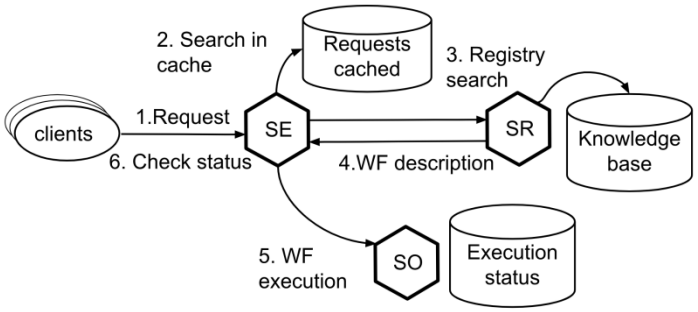


Figure 2: Main components of the proposed solution

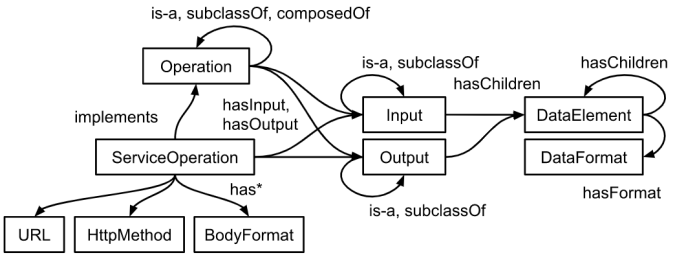


Figure 3: Service ontology (fragment)

### 4.2. Service Matchmaking

Existing standard solutions in service registration like UDDI are not enough for advanced semantic search. Instead it is proposed to rely on service ontologies, similar to OWL-S ontology [9], which will give more relevant results. Service matching can be done using the semantic proximity of the service ontology elements (from registry knowledge base and those from the "request for functionality" query) [10]. In order to compare these ontology elements, the following elementary proximity estimations are needed to be computed: proximity of classes, proximity of classes and instances, proximity of instances, proximity of predicates. Some service A can be called 'relevant' to some search query R only in case the proximity estimate is higher than some threshold value $M_T$ (see further).

Another aspect is to choose what information should be included into comparison. It is proposed to use the following information categories: C (context), IOPE (inputs, outputs, preconditions, effects) and QoS.

The context C form search query can be formally defined as any information that could implicitly and explicitly impact the query generation by user (it can be profile-oriented, history-oriented, process-oriented, other context). An explicit context is

directly provided by user, while implicit one is that being gathered by any automated tool.

The service functionality can be matched by use of IOPE attributes. The *hasInput*, *hasOutput*, *hasPrecondition* and *hasEffects* query attributes (R) are matched against the corresponding ones from each service's description (A). Matching can be estimated as a one of following outcomes: "exact match", "plug-in" match when R is a subclass of A, "subsume" when R and A have common partial match, "failure" is A and R have nothing common.

QoS attributes include service price, performance, reliability, stability, scalability, security etc. The total match level estimation combines those estimations mentioned above:

$$M(R, A) = K_C M_C + K_I M_I + K_O M_O + K_P M_P + K_E M_E + \\ + K_{QoS} M_{QoS} > M_T$$

If we group IOPE and C requirements as functional requirements (FR) and QoS we refer to as non-functional requirements (NFR), and if we use mismatch-level (or level of matching error) as an indicator of matching failure (for practical reasons: to be able to use mismatch penalties instead of proximity estimations), then we'll have the following general matching error function:

$$E(R, A) = K_{FR} E_{FR} + K_{NFR} E_{NFR} < E_T$$

Both measures M and E could be used combined. First, we look for the single service with the highest M(R,A). If $M(R,A) > M_T$ then we accept it as a service providing requested functionality. If $M(R,A) < M_T$ then we need to compose a workflow of services with less possible $E(R,A) < E_T$. In order to estimate E for a workflow of services {A} we can use this general approach:

$$E(R, \{A\}) = K_{FR} E_{FR}^* + K_{NFR} E_{NFR}^*,$$

where * marks summary values for a workflow, for instance, internal inputs and outputs that are connecting services are excluded from error estimation.

During the practical implementation of a e-Health product with less than hundreds of service operations and without third-party services we found it sufficient to consider only IOE requirements (E for effects in form of functionality requested), without NFR. So, we used the following brief construct for the functionality request:

$$R = (F, \{I\}, \{O\}),$$

where F (function) – action to perform, I (input) – available inputs specification, O (output) – requested outputs specification. In a practical implementation we used JSON format, flexible enough to express complex I and O descriptions (see Figure 4). Here is a simple example of request to SE to get user name by ID:

```
{
  "operation":"get",
  "input":[
    {
      "name":"user",
      "properties":[
        {
          "name":"id",
          "value":"user001"
        }
      ]
    }
  ],
  "output":[
    {
      "name":"user",
      "properties":[
        {
          "name":"firstname"
        },
        {
          "name":"lastname"
        }
      ]
    }
  ]
}
```

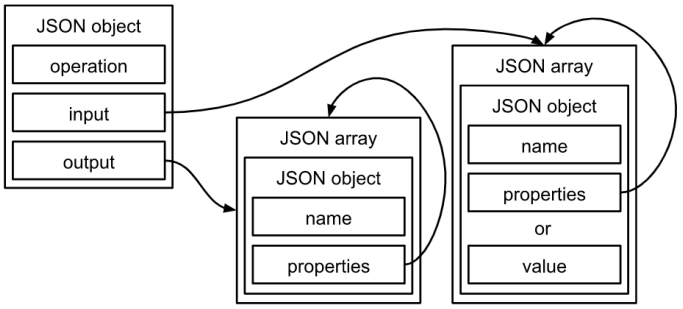


Figure 4: Inputs and outputs specification format

### 4.3. e-Health Application

The are many research papers studying the use of service-oriented architecture for medical software [11,12], and some of them mention service orchestration as a part of medical software, but this paper focuses on slightly different things, related to orchestration mechanism itself.

Workflow composition or, in other words, automatic semantic-based orchestration scenario synthesis, makes is easier to extend existing functionality of the system. It could be possible to get this new functionality even without development of new services: when the new functionality request could be satisfied with an orchestration scenario involving existing services. Before describing an example of such scenario, let us briefly describe the e-Health application that uses the proposed approach.

The e-Health application we develop is a system providing a virtual office for doctors and patients as an online point of their communication. It is useful in conditions when the nearest medical center is far from patient's home. In this situation the initial diagnostics can start online, and in many cases, it could be enough to help patient without real-life visit to the hospital. Also, the virtual doctor's office remembers all the patient treatment history and can provide some intelligent tool helpful for doctors and patients, like prediction of the crisis state of patients' health based on ordinary indicators like blood pressure, glucose level or complains about pain. This is an example of extended functionality that can be built by re-using the existing functions. The general

architecture blocks of this e-Heath application are presented on Figure 5.

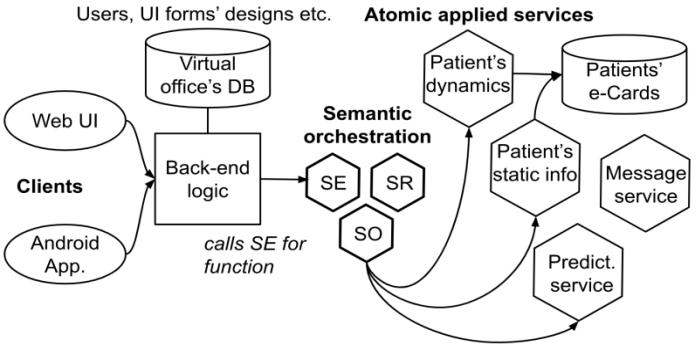


Figure 5: Virtual doctor's office using the dynamic semantic-based orchestration

Most of the office's functionality is not implemented just by back-end logic querying the database. There is a separate layer called "applied services" which is actually an API to do everything on patient's record. Conversations with doctors, diagnoses, treatment prescriptions, health indicators, reports – all of this is implemented with atomic REST services (we can call them microservices). The back-end logic's responsibility is to serve front-ends' requests and gather all necessary data for displaying to the user. Any call to operate patient's data is made as a "functionality request" with function, inputs and outputs provided to SE. This way we can truly separate back-end development and microservice API development, and only need to reflect important changes in a knowledge base of SR.

### *4.4. Service Workflow Composition Example*

One of the examples where the dynamic orchestration takes place is the prediction of diabetes, heart problems and so on based on the last entered health indicators. This prediction is just a hint for the doctor and should only attract his attention to some possible problems (by displaying a warning), so there was no need to achieve very high accuracy of prediction. To develop such "intelligent" hint test datasets were analyzed, prepared and used for neural network training. This ended up as a separate service with one single output (binary warning flag) and a number of inputs (like weight, insulin and glucose level and much more) developed by separate team with machine learning background. In order to get the diabetes warning flag, the back-end developers just made a simple request: "given patient's ID and current date, run diabetes prediction and return warning flag". At the same time the SR's knowledge base contains information about inputs and outputs of all services, including the diabetes predictor.

Let's take a look at the process of execution of that request mentioned above. After JSON describing this request is accepted and parsed by SR, the SR itself starts to search for single service capable to satisfy the request. It founds the service operation with function briefly described like "Diabetes prediction". But this service operation has a lot of inputs, despite it has the output and function matching the request. So, no single service is found, and SR tries to build the workflow. It finds a lot of combinations of atomic services, but the best one (according to matching error penalties) is presented on Figure 6. It is executed, and the result is returned to the back-end caller (the client).

The Figures 7 and 8 shows two screens of a mobile client developed using the described diabetes prediction workflow to display a warning icon (the application's language is Ukrainian). The first screen is a menu screen for a doctor with a following action on a selected patient: indicators, chat, log, complains, diagnosis, treatment. The second screen shows some sample conversation between a doctor and his patient in a chat mode.

When a doctor selects someone of his patients, the diabetes prediction workflow is called, and the client displays or hides the warning icon (a circle with Ukrainian letter "D" inside) at the top of the screen related to that patient (seen on both screens on Figures 7 and 8). In a similar way any other predictor (e.g. for heart problems) is called via the dynamic semantic-based orchestration.

### *4.5. Evaluation of Results*

The performance penalties for this sophisticated process of services invocation (compared with direct REST services calls) could be estimated through the workflow execution time $T_{exec}$ :

$$T_{exec} = T_M + T_V + T_{WF} + T_S$$

where $T_M$ is a matchmaking time, $T_{WF}$ is a workflow execution related overhead (depends on workflow execution engine speed), $T_S$ is a total time for the execution of the longest sequence of service invocations in a flow (web services execution time). There is also $T_V$ which is a validation time of the composed workflow. Currently we don't use any validation technique, and fully rely on the consistency of the knowledge base during the automated workflow synthesis ($T_V = 0$). But there is a risk of the incorrect behavior of the composed workflows (like infinite loops) even if they were constructed without any inconsistency with the knowledge base, so we need formal modeling techniques. Petri nets [13] and process calculus [14] could be successfully used to model web service workflows and related tools will be further integrated to the system to improve its reliability.

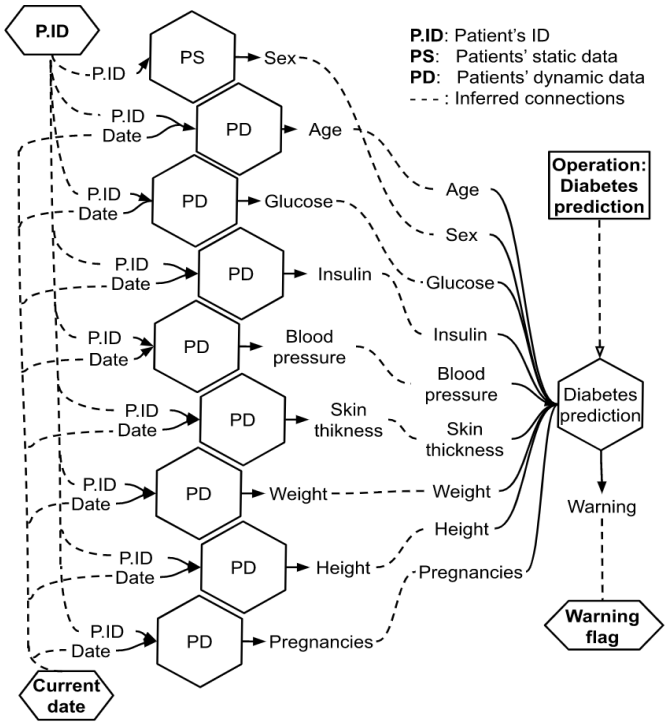


Figure 6: Diabetes prediction workflow (inputs and output are marked bold)

When comparing hard-coded scenario execution ($T_{exec}' = T_S$) with dynamic orchestration $T_{exec}$, the dynamic and flexible approach lasts longer by $T_{exec} - T_{exec}' = T_M + T_{WF}$, which was 3 to 5 seconds in case of small service registry (about 30 service operations) we used for diabetes prediction scenario. It should be noted that this time is seriously affected by the knowledge base facts quantity and quality (seriously impacts $T_M$).

In order to avoid same process of workflow synthesis each time a doctor needs a hint, the resulting workflow is cached in SR's knowledge base. The overhead is minimal and almost equal to ($T_{WF} + T_S$) because in this case $T_{exec}'' = T_C + T_{WF} + T_S$, where $T_C$ is a cache search time and $T_C << T_M$). But in case of any changes in the knowledge base the cache should be cleared. This is a trade-off we get instead of updating the clients: the knowledge base update leads to reasoning on the updated facts and repeating the service matchmaking again from scratch. But found these knowledge base related activities less time-consuming then updating, redeploying and QA of all client applications.

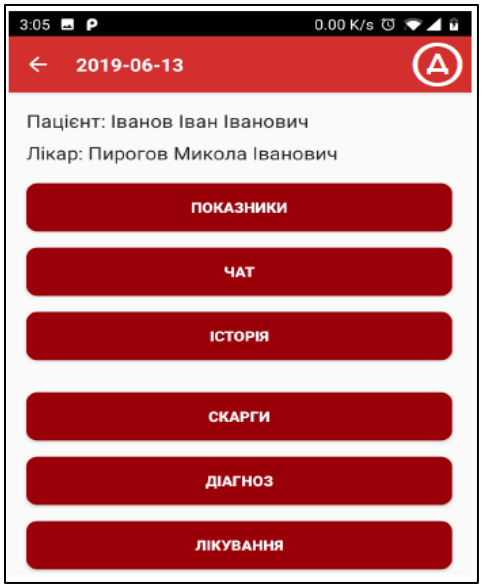


Figure 7: A 'patient menu' sample screen of the mobile e-Health application with diabetes warning icon at the top.


Figure 8: Chat screen of the mobile e-Health application, with diabetes warning icon at the top.

## 5. Conclusions and Future Work

This paper presented a description of a software architectural approach that aims to bring more flexibility to service-oriented systems and make service dependencies weaker. The main idea is to provide additional "abstract request layer" to the well-known service orchestration mechanism. The abstract request for functionality is being matched with the knowledge base of available web services to perform actual calculations.

The approach described has been successfully applied to the e-Health system, and it gives promising results for this application field, since e-Health applications typically have a lot of functions and data sources that could be accesses with the web service interfaces.

In our opinion this approach has the following benefits not only for e-Health domain but for other application fields as well:

- Service clients and web services are fully separated and could be developed independently by the teams of different development tools and background. The only dependency is the knowledge base (and service registry that uses it). Service ecosystem can be easily updated without changes to its clients. Only the knowledge base needs to be updated.
- It is possible to develop the UI for users to allow them easily construct the functionality requests and thus extend the functionality without help from software developers. This is one direction for our future work. It includes the development of workbench where users can test their requests and development of UI editor to control the displaying of requested results.
- In case of small service registries, the matchmaking process and workflow synthesis will not harm performance. This is true in case of avoiding usage of some existing heavyweight tools for orchestration like BPEL engines as the core for service orchestrator. The problem is that these tools like BPEL engines were basically designed for static workflows, not dynamic ones.

Of course, this vision has its drawbacks:

- The e-Health software must be very robust. But it has to be noted that the absence of hard-coded reference to concrete services there is a non-zero probability that SR will find wrong services or does not find anything thus producing wrong results returned by SE. From this point of view the only way in assuring the correct behavior is continuous testing of functions used by clients. When client gets updated with new functionality request call, this call should be added to the test plan. The improvements in QA of the solution presented are planned for our further work. At the same time formal models like Petri nets or process calculus can help to test the behavior of the dynamically generated workflows automatically.
- Another option to make sure correct services are matched to a request is to make only strict logical assertions by SR. No partial matching possible, service or workflow is either matching the request or not. In this case the clients will

need to use only strict request constructs and only correct ontology terms it their requests, thus making the new programming language to express the functionality requests. This will make it harder to add the new requests and make them work by end users and even programmers.

- In case of large registries of services, the matchmaking process could be time-consuming, since a lot of services could be "paired" according to their inputs and outputs and their functionality. So this solution will not work as expected in a system with hundreds of service operations and third-party services. And there will be problems if there are too many fine-grained microservices in the ecosystem even if it does not include external third-party services. In this case the matchmaking process should be restricted of simplified since e-Health software should operate without lags.
- Another problematic issue is the correct management of the knowledge base. Each time a new service is developed and should be registered in the system, there are a lot of work to do in the knowledge base. In fact, the knowledge base is the most important component of such system and requires very accurate updates, permanent logging of changes and intensive QA. Together with service registry it forms the fragile bottleneck of the system.
- Such complex interaction patterns could bring additional problems in security and data privacy. This is very important for e-Health applications working with personal medical data. But this aspect is out of scope of this paper.

Other possible directions for the future work are: research on the specifics of orchestrating services that control IoT devices and moving orchestrator service to the cloud infrastructure.

## Acknowledgment

This paper describes partial results of the ongoing fulfillment of the project "The development of the modern service systems by the example of mobile medical system for a front-line settlements in the military conflict zone" by the National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute".

## References

[1] A.I. Petrenko, B.V.Bulakh, "Intelligent service orchestration as a service", Proc. of 2018 IEEE First International Conference on System Analysis and Intelligent Computing (SAIC), 8-12 October, 2018, Kyiv. - pp 201-205. https://doi.org/10.1109/saic.2018.8516723

[2] B. Ludäscher, M. Weske, T. McPhillips, S. Bowers, "Scientific Workflows: Business as Usual?", Lecture Notes in Computer Science. (2009) 31-47. https://doi.org/10.1007/978-3-642-03848-8_4.

[3] V. Ladogubets, B. Bulakh, V. Chekaliuk, O. Kramar, "Employing BPEL Engines for Engineering Calculations", The Experience of Designing and Application of CAD Systems in Microelectronics: 12-th Intern. Conf. «CADSM'2013», 19-23 February 2013, Polyana-Svalyava (Zakarpattya), Ukraine: proc. – Lviv, 2013. – P. 427–430. – ISBN 978-617- 607-393- 2

[4] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers et al. "The Taverna workflow suite: Designing and executing workflows of Web Services on the desktop, web or in the cloud". Nucleic Acids Research. 41 (Web Server issue): W557–W561. doi:10.1093/nar/gkt328. PMC 3692062. PMID 23640334.

[5] M. Pierce, S. Marru, L. Gunathilake, T. A. Kanewala et al., "Apache Airavata: Design and Directions of a Science Gateway Framework," 2014 6th International Workshop on Science Gateways, Dublin, 2014, pp. 48-54. doi: 10.1109/IWSG.2014.15

[6] J. Kranjc, R. Orač, V. Podpečan, N. Lavrac, M. Robnik-Sikonja.. "ClowdFlows: Online workflows for distributed big data mining". Future Generation Computer Systems 68, 2017. pp. 38-58. doi:10.1016/j.future.2016.07.018.

[7] R. Seiger, S. Huber, P. Heisig. "PROtEUS++: A Self-managed IoT Workflow Engine with Dynamic Service Discovery". 9th Central European Workshop on Services and their Composition (ZEUS). 2017. pp 90-92.

[8] O.O. Petrenko, A.I. Petrenko. "Cyber-Physical Medical Platform for Personal Health Monitoring", Journal of Scientific Achievements (JSA), Australia, vol. 2, issue 8, 2017, pp. 24-28. ISSN: 2207-4236

[9] OWL-S: Semantic Markup for Web Services. [online] Available https://www.w3.org/Submission/OWL-S/.

[10] Петренко І.А., Петренко О.О. "Автоматизовані методи пошуку і відкриття необхідних сервісів", Вісник Університету «Україна», Серія «Інформатика, обчислювальна техніка та кібернетика», №1(17), 2015, С. 55-64

[11] S. Rodriguez Loya, K. Kawamoto, C. Chatwin, and V. Huser, "Service Oriented Architecture for Clinical Decision Support: A Systematic Review and Future Directions". Journal of Medical Systems, December 2014, 38(12): 140. https://doi.org/10.1007/s10916-014-0140-z

[12] A. Celesti, M. Fazio, F. Galán Márquez, A. Glikson, H. Mauwa, A. Bagula, F. Celesti, and M. Villari. How to Develop IoT Cloud e-Health Systems Based on FIWARE: A Lesson Learnt. Journal of Sensor and Actuator Networks. 2019, 8(1), 7; https://doi.org/10.3390/jsan8010007

[13] W. van der Aalst, K.M. van Hee. Workflow management: models, methods, and systems. MIT press, 2004. 368 p.

[14] M. Weidlich, G. Decker, M. Weske. "Efficient Analysis of BPEL 2.0 Processes using pi-Calculus". Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC'07). – Japan, 2007. – P. 266-274.