A Heterogeneous Approach for Developing Applications with FIWARE GEs

Simone Di Cola[1], Cuong Tran[1], Kung-Kiu Lau[1], Antonio Celesti[2], and Maria Fazio[2](✉)

[1] School of Computer Science, The University of Manchester, Manchester M13 9PL, UK
{dicolas,ctran,kung-kiu}@cs.man.ac.uk

[2] Facoltá di Ingegneria Contrada di Dio, S. Agata, Universitá Degli Studi di Messina, 98166 Messina, Italy
{acelesti,mfazio}@unime.it

Abstract. The European Commission funded FIWARE project aims to support the development of a European cloud, and a rich catalogue of generic components called Generic Enablers (GEs). However, the lack of an efficient approach and tool for developing applications using GEs hinders their adoption. This paper tries to fill this gap by proposing an approach based on a component model, along with its related tool, that allows heterogeneous composition of GEs and non-GE components. The approach is validated with a case study where a content delivery application is developed.

Keywords: Cloud · FIWARE · Generic enabler · Component model · Heterogeneous composition

1 Introduction

FIWARE [3] is an initiative funded by the European Commission whose aim is to ease the development of smart applications by means of an open cloud-based infrastructure that offers a catalogue of ready-made components called Generic Enablers (GEs). Each GE offers a number of general-purpose functions through public and royalty-free APIs.

Developing an application using FIWARE GEs means constructing a software system by composing GEs with non-GEs software components [2]. In current state of the art, GEs can be used in a workflow, or composed in a GE bundle[1].

In a workflow, GE instances are orchestrated. Workflow activities invoke specific services on GE instances which are already deployed on some servers.

C. Tran—Research leading to these results has received funding from the EU ARTEMIS Joint Undertaking under grant agreement no. 621429 (project EMC2) and from the Technology Transfer Board (TSB) on behalf of the Department for Business, Innovation & Skills, UK.

[1] http://catalogue.fiware.org/bundles

S. Dustdar et al. (Eds.): ESOCC 2015, LNCS 9306, pp. 65–79, 2015.
DOI: 10.1007/978-3-319-24072-5_5

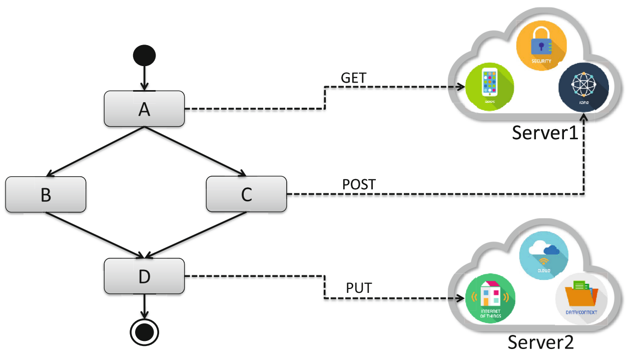


Fig. 1. A workflow using Generic Enablers.

For instance, Fig. 1 depicts a workflow where three activities `A`, `C`, and `D` invoke respectively *GET*, *POST* and *PUT* methods to GE instances on `Server1` and `Server2`.

A bundle, in contrast, is a template that dictates a possible composition of specific GEs directly interacting with one another. Such composition is described informally as configuration instructions to be manually performed when participating GEs are deployed. As such, composition is not concrete until deployment time. Moreover, complex control flows or data transformations usually need to be developed afresh as services sitting between GEs.

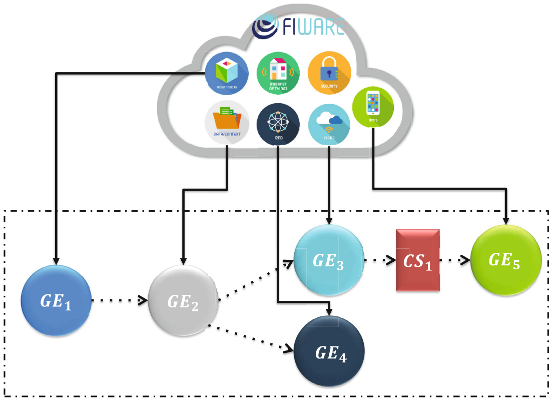


Fig. 2. A GE bundle.eps

Fig. 2 depicts an example of bundle which consists of five GEs. Those GEs can be configured so that GE_1 can call GE_2 which can call GE_3 and GE_4. GE_3 calls services in CS_1 which then calls GE_5.

We argue that both mechanisms are inefficient in developing GE-based applications. A workflow requires that all participant components expose WSDL or

RESTful services. This represents a heavy-weight solution [6], which is sometimes not even applicable (e.g. for a GUI component). A bundle composes a set of GEs, but it needs to be customized in order to result in an application. Furthermore, the composition is informally described, and has to be performed manually.

In this paper, we present a component model [8], and its related tool [1] that has been extended for constructing GE-based applications. Our solution supports heterogeneous systems because it allows composition of GE and non-GE software components. Such components are hierarchically composed by means of predefined (exogenous) composition connectors.

To show the effectiveness of our approach, we construct a simple case study for the provisioning of multimedia contents. It includes three GEs and two non-GE components, which are composed in a meaningful application.

The paper is organized as follows: Section 2 presents related work in the literature. Section 3 introduces our component model. Details on how to use our component model for heterogeneous composition of GEs and non-GE components are then presented in Section 4. A content delivery application is presented in Section 5 as a case study to evaluate our solution. Details of the implementation of the related workflow in X-MAN are discussed in Section 6. Section 7 summarizes the advantages and limitations of the presented solution, and identifies future work that we intend to pursue.

2 Related Work

There have been a number of efforts to exploit FIWARE GEs technology. In the ENVIROFI project [5], domain specific GEs were identified and developed for six domains. The work reported in [17] presents an interesting utilisation of FIWARE to handle IoTs, smart environment devices, data and services by using a semantic approach within the FIWARE core platform. The work described in [19] illustrates an application for sensor driven FI applications used as a motion sensor cloud service. A solution for healthcare developed exploiting FIWARE technologies is presented in [2]. In [20], an application for text mining based on the BigData GE is developed and presented.

In supporting developing GEs and GE-based applications, a tool set called FI-CoDE[2] was developed. Essentially, it consists of several Eclipse plug-ins for GE-based software projects. Apart from generic functionalities such as collaborative development, task management, version control and testing, it provides a Java code generator to yield GEs clients. There is no support for building GE bundles or composing GEs with non-GEs components.

In the area of service composition, workflow is the de facto standard approach. A workflow can be defined using a suitable language such as BPEL [21], BPMN [18], or JOpera Visual Composition Language (JVCL) [16]. A workflow can be turned into a service by giving it a WSDL, or RESTful interface. One representative work is BPEL for REST [14], which offers a heterogeneous mechanism

[2] http://catalogue.fiware.org/tools/fi-code-tools

to compose RESTful and WSDL services. For homogeneous RESTful services composition, JOpera [15] provides a visual modelling language for workflows. A JOpera workflow can be compiled into a RESTful service.

Following the same idea, the FIWARE catalogue contains a special GE called Ericsson Composition Engine (ECE) [13]. It consists of a composition editor for creating composed service skeletons, and a composition execution engine. Offering its own graphical language, the editor allows users to model event-driven service executions and data flows. A configure service skeleton can then be instantiated into a workflow and executed within the composition execution engine.

RESTful services can be aggregated in a web application with web widgets and data sources by means of mashup [11,22]. The mechanism to perform a mashup is still a workflow. However, mashup cannot be used on non-web software components and applications.

In component-based development, SCA [12] is a component model that allows us to create heterogeneous composition of various components which may be implemented as Java classes, RESTful, and WSDL services. SCA does not explicitly define control and data flows in a composition.

3 Our Component Model

Our approach is based on an extended version of the X-MAN component model [7], with three kinds of first-class entities: *components*, *connectors*, and *services*.

Components. There are two types of components: *atomic*, and *composite*. They are both fully encapsulated, i.e. they have no external functional dependencies.

An *atomic component* (Fig. 3a) is a unit of computation. Its `computation unit` (CU) contains the implementation of the `services` $(S_1, \ldots, S_m)$ it exposes via the `invocation connector` (IC).

Its behaviour can be specified in the language of state charts (Fig. 3b): when a service S_i is invoked, a transition occurs from the initial state to the state in which S_i is executed; when S_i's computation ends, the component reaches

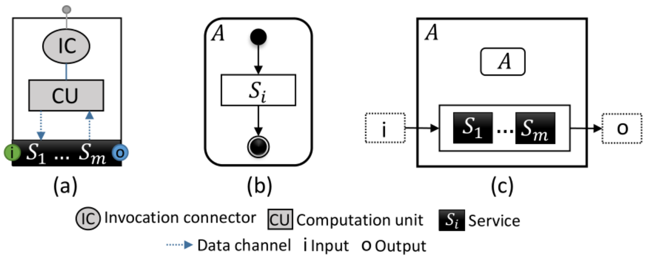


Fig. 3. An atomic component and its functional model.

its end state. Data to and from the CU is provided and retrieved via `service` parameters. The activity chart in Fig. 3c shows parameters as external activities, and services as internal ones. The latter are controlled by the control activity A defined by the state chart in Fig. 3b.

Atomic components are composed into *composite components* by means of *composition connectors.*

Connectors. *Composition connectors* are (exogenous) control structures that coordinate the execution of the components they compose. They are `Sequencer` (SEQ) and `Selector` (SEL), which provide sequencing and branching respectively.

The component Q in Fig. 4a is built by sequencing n atomic components $A_1 \dots A_n$. Similarly, the same atomic components composed by a selector results in the composite component B in Fig. 5a. The state chart for Q (Fig. 4b) is

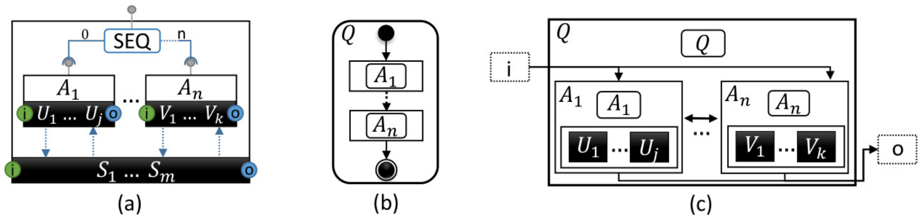


Fig. 4. A composite component with sequencer and its functional model.

composed from the state charts for $A_1 \dots A_n$ by sequencing them in the order specified in SEQ. Similarly, the state chart for B (Fig. 5b) is composed from the state charts for $A_1 \dots A_n$ by branching according to the condition in SEL.

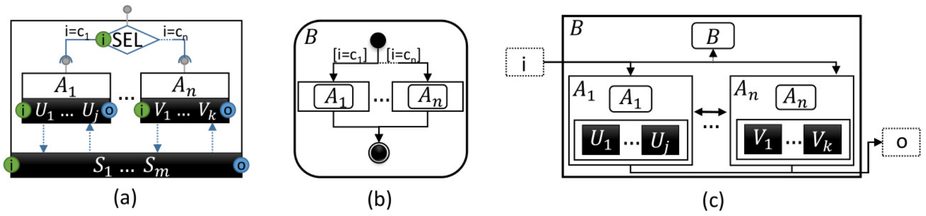


Fig. 5. A composite component with selector and its functional model.

The activity charts for Q (Fig. 4c) and B (Fig. 5c) are composed from those of $A_1 \dots A_n$. Data flow among activities mirrors the data flow among the corresponding services. The control activity B receives a control flow input needed to perform branching decisions.

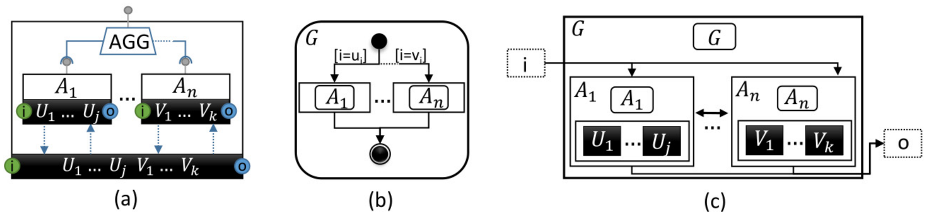


Fig. 6. A composite component with aggregator and its functional model.

Apart from composition connectors, X-MAN also defines an `aggregator` connector (AGG), which aggregates in a new composite component the services exposed by its sub-components. An aggregated component effectively provides a *façade* to the aggregated services. In Figure 6a, the component G is built by aggregating the services exposed by components $A_1 \dots A_n$. Like the composite component B, the state chart for G is composed by branching among the state charts for $A_1 \dots A_n$, but with a condition on the choice of service. Its activity chart is composed from the activity charts for $A_1 \dots A_n$.

Single components can be adapted by `adapters` such as `loop` (L) and `guard`. The former provides looping, while the latter gating (we omit its details for lack of space). Fig. 7a shows a component A adapted by L into R. The state chart

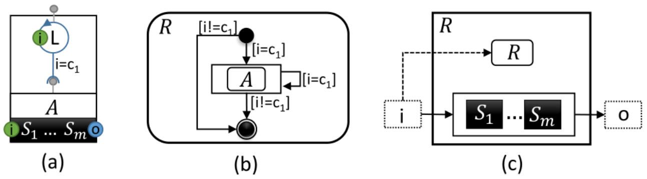


Fig. 7. A component with loop and its functional model.

for R (Fig. 7b) is composed from the state chart for A by looping the latter until condition i is verified; failing that, the end state is reached. Finally, in its activity chart (Fig. 7c), the loop condition is shown as a control flow coming from the external activity i.

Services. A *service* represents an operation exposed by a component. It contains two main entities: parameters, and service references. *Parameters* are inputs and outputs, while *service references* specify services in sub-components that contribute to the provided operation.

As already stated, an atomic component exposes services implemented by its CU. For example, the atomic component A in Fig. 3a offers m services named as $S_1 \dots S_m$.

On the other hand, a composite component, or an adapted one, exposes services resulting from the coordination of the ones exposed by its sub-components.

For instance, the composite component Q in Fig. 4 exposes m services ($S_1 \dots S_m$) resulting from sequencing services $U_1 \dots U_j$ and $V_1 \dots V_k$ from sub-components $A_1 \dots A_n$ respectively. Moreover, the adapted component R in Fig. 7a has its services realised by conditionally repeating invocation to $S_1 \dots S_m$ services of the original component A.

Clearly, any architecture in X-MAN is a service-oriented one. Moreover, we can hierarchically and compositionally build larger service-oriented architectures from existing ones. This leads to the next section where we detail our composition of GEs.

4 Heterogenous Composition

Our approach allows heterogeneous composition of GEs and X-MAN components to construct applications, i.e. composite components. To that end, a suitable mechanism needs to be devised.

A Generic Enabler (GE) is encapsulated, by definition. Its provided services are fully implemented and available via a RESTful interface. The interface can be formally specified in WADL [4], or informally as text.

Taking into account the aforementioned characteristics, it is sound to map a GE into a X-MAN atomic component, albeit a special one. In Fig.8, a GE atomic component is depicted as a white cube. The computation unit of a GE atomic component is always remote. Therefore, a GE atomic component needs to maintain an URL pointer to a GE instance. This pointer can be specified at design time or later when an instance of a GE atomic component is instantiated. Such a pointer is named as `Based_URI` in Fig. 9a.

In addition, the services of a GE need to be mapped to the ones of a GE atomic component. Usually, GEs expose RESTful services and are thus resource-oriented. There are four possible CRUD operations namely *POST*, *GET*, *PUT* and *DELETE*. As in Fig. 9(a) for each operation, the mapping yields a service-oriented counterpart in a GE atomic component. For instance, mapping for four RESTful services `GET /res1`, `PUT /res1`, `POST /res1`, and `DELETE /res1` produces four services called `sGetRes1`, `sPutRes1`, `sPostRes1`, `sDeleteRes1` respectively.

Furthermore in our mapping, a RESTful service usually has a dynamic URI which is constructed from a base URI. The dynamic part is influenced by the parameters for that service. Parameters can be either in the *path* or in the *query* part of a URI. For any situation, the mapping specifies those parameters in the resulting service as its inputs. For each input, the name, data type, order and an attribute indicating whether the parameter is path or query based is specified. Outputs for a RESTful service include status code and data. The status code indicates a provisional response of the service call, while the data is the result of service execution. Our mapping creates two outputs respectively for both of them.

In Fig. 9b, we give details of an example of our mapping. A RESTful service offers a GET method to access a resource called `res1`. From its API documentation, it accepts three mandatory parameters `param1`, `param2` and `param3`, and

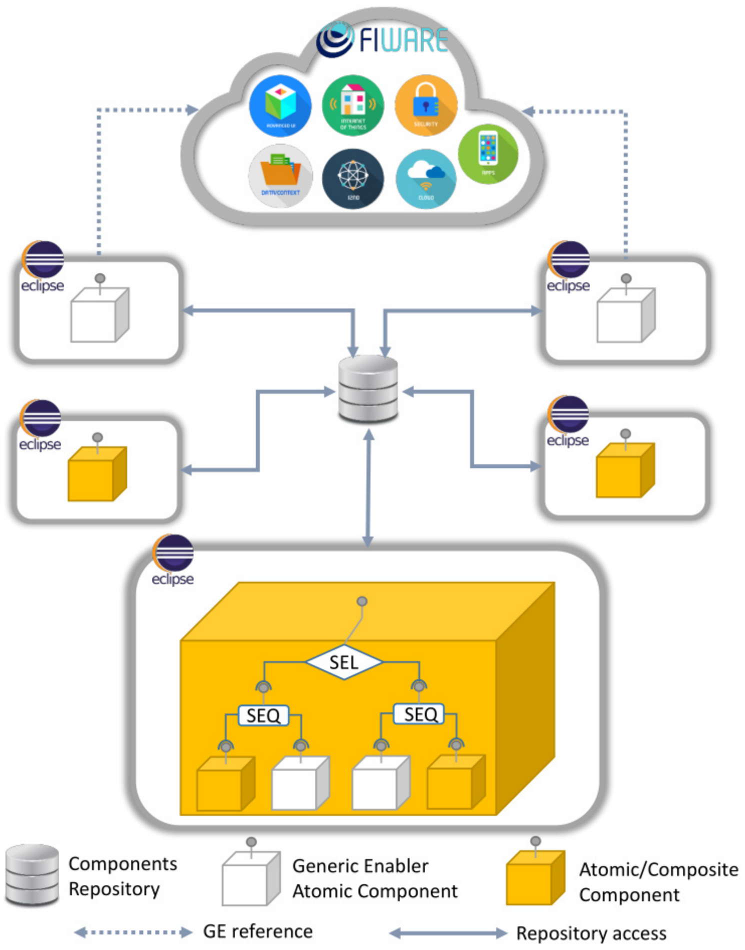
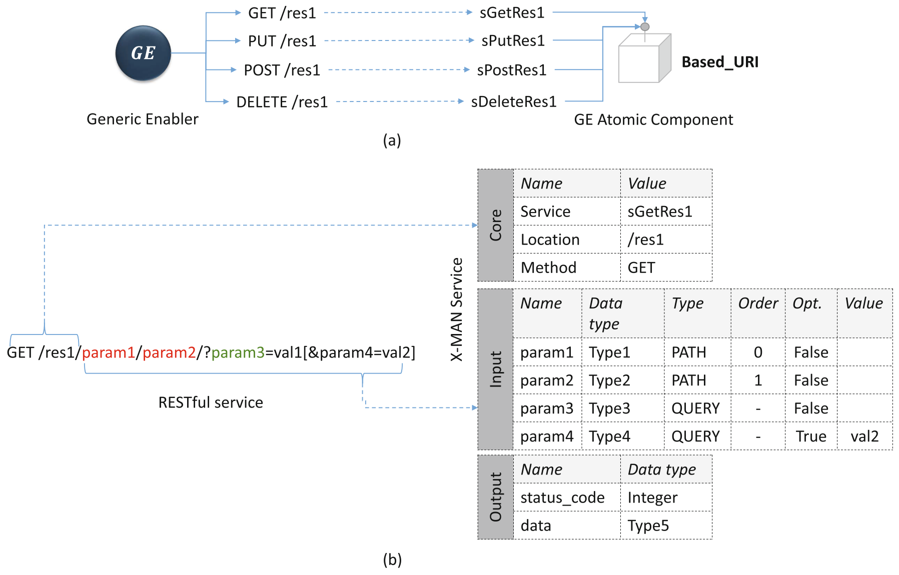


Fig. 8. Approach overview.

an optional `param4`. The former two parameters are path-based, separated by '/', while the latter two are query-based, following '?' and delimited by '&'. Our mapping yields a new service which is described in three parts. The core part specifies the service name, resource location relatively to the base URI and the method as `sGetRes1`, `/res1` and `GET` respectively. The input part has four inputs `param1`, `param2`, `param3` and `param4` matching the ones of the service. The data types of these inputs are identified from the API. From the same source, the order, optionality default values of inputs are identified. The output section consists of two outputs which are `status_code` and `data`. `status_code` is always an integer while `data` can be of a type matching one stated in the API.

Once GE atomic components are defined, they can be composed with other components by means of X-MAN composition connectors to yield a composite X-MAN component. Such a composite component is in fact a GE-based application.



Name	Value
Service	sGetRes1
Location	/res1
Method	GET

Name	Data type	Type	Order	Opt.	Value
param1	Type1	PATH	0	False	
param2	Type2	PATH	1	False	
param3	Type3	QUERY	-	False	
param4	Type4	QUERY	-	True	val2

Name	Data type
status_code	Integer
data	Type5

Fig. 9. Mapping of GE to GE atomic component.

It can be delivered, or stored in our repository (Fig. 8) in order to be further composed.

5 Case Study: A Content Delivery Application

Figs. 10 and 11 depict a simplified content delivery application. The application stores multimedia data in an object storage system. Whenever an authorised client asks for a content, the application stores his information, verifies its availability and delivers it if found. Moreover, for performance analysis, the response time is also returned.

To develop this application we compose three GE atomic components (`Cosmos`, `ObjectStorage` and `Kurento`), with two X-MAN components (`Logger`, `Adapter`):

- `Cosmos` offers cluster-based data persistence and functionality for processing vast amounts of data; we use it to store and search content meta-data.
- `ObjectStorage` provides robust, and scalable object storage functionality; we use it to store actual media content.
- `Kurento` implements an abstraction layer for multimedia capabilities; we use it to stream media contents to clients.

- `Logger` logs clients' credentials, and calculates the response time.
- `Adapter` analyses `Cosmos` result to extract relevant content meta-data.

As depicted in the top right corner of Figs. 10 and 11, the application offers two services: *StoreMedia*[3], which stores a media, along with its meta-data, in the FIWARE cloud, and *PlayMedia*, which streams a required media from it.

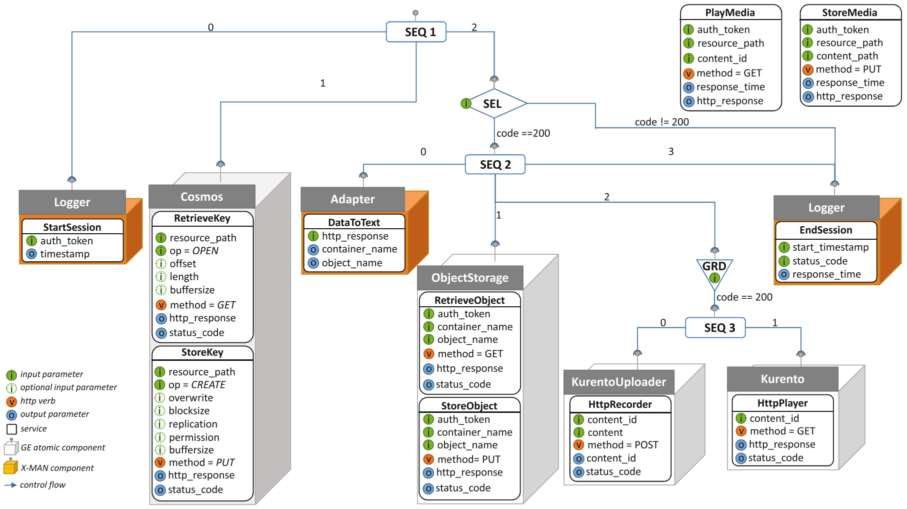


Fig. 10. A content delivery application (control flow)

To access a content, a client needs an authentication token, the path of the required media, and the media id. Following the order of the root connector (SEQ 1) in Fig. 10, the application first invokes the service *StartSession*, which logs information about the callee (extracted from the provided authorisation token), and returns the actual timestamp. The latter will be used by another instance of *Logger* to calculate the response time (Fig. 11).

Once the request is logged, the service *RetrieveKey* of the GE atomic component `Cosmos` is invoked. The mapped API (Fig. 12) requires the resource path (optional parameters have been omitted for simplicity), and returns its content. If the resource is found, then the *DataToText* service parses its content, and returns two parameters, i.e. *container_name*, and the *object_name*. The former are used by the service *RetrieveObject*, which maps to the counterpart API in Fig. 12, to return the object content in the response body. If the return code is 200, the *http_response* is redirected to the service *HttpRecorder*. The latter, using the corresponding API in Fig. 12, uploads the object content to the Kurento Media Server. Finally, the *content_ID* is passed to the Kurento *HTTPPlayer* service (Fig. 11), which returns a JSON object containing the content URL.

[3] We omit its details to simplify our discussion. Indeed, other X-MAN components are needed to upload content to `Cosmos`, and `ObjectStorage`.

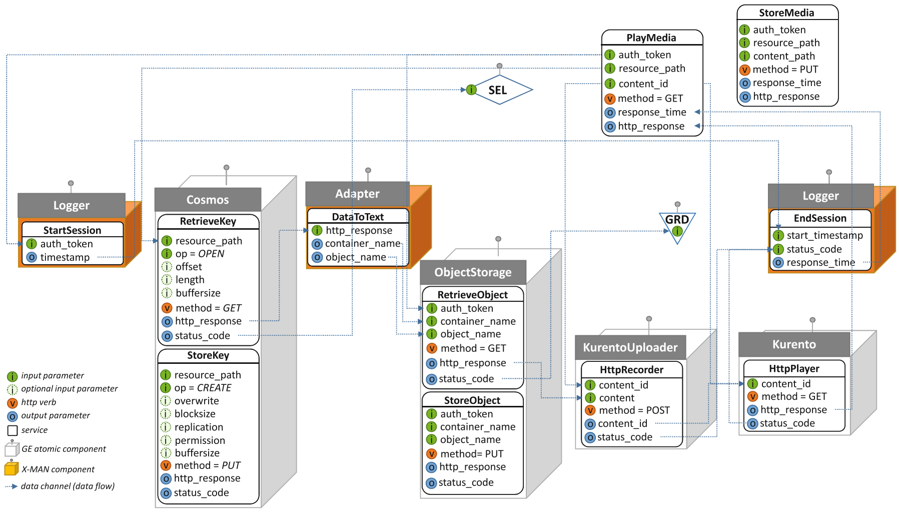
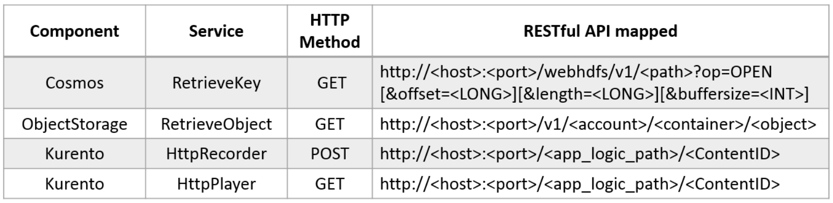


Fig. 11. A content delivery application (data flow)



Component	Service	HTTP Method	RESTful API mapped
Cosmos	RetrieveKey	GET	http://<host>:<port>/webhdfs/v1/<path>?op=OPEN [&offset=<LONG>][&length=<LONG>][&buffersize=<INT>]
ObjectStorage	RetrieveObject	GET	http://<host>:<port>/v1/<account>/<container>/<object>
Kurento	HttpRecorder	POST	http://<host>:<port>/<app_logic_path>/<ContentID>
Kurento	HttpPlayer	GET	http://<host>:<port>/<app_logic_path>/<ContentID>

Fig. 12. RESTful APIs mapped

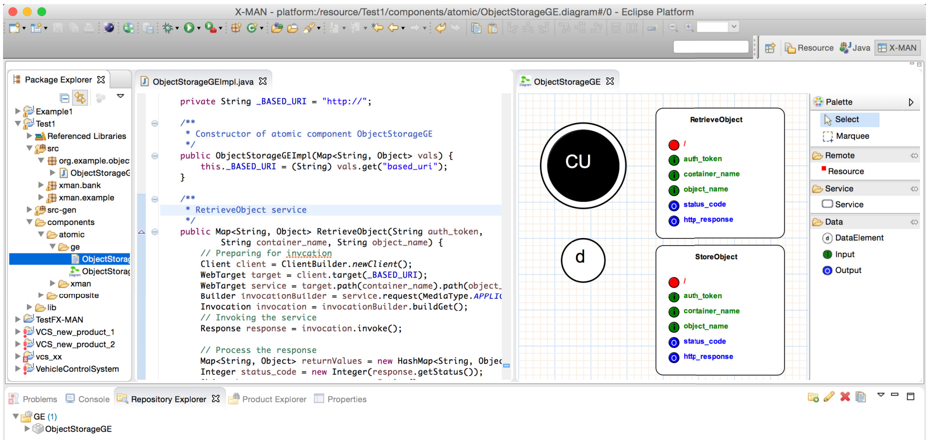
6 Implementation

Based on the new X-MAN tool [1], two extensions are developed. The first extension is to implement GE atomic components while the second one is to support heterogeneous composition. To that end, the meta-model is extended to capture the mapping results, i.e. core, inputs and outputs.

Our code generator is then extended to support the new extensions. The generated code for GE atomic components is essentially to perform invocations to GEs' RESTful services. For those invocations implementation, we use the Jersey library.[4]

In Fig. 13 we illustrate the design of the `ObjectStorage` GE atomic component in our tool. It offers two services *RetrieveObject* and *StoreObject*, each of which has three input and two output parameters. The base URI is captured by the data element (circle with letter d), specified at deployment time.

[4] https://jersey.java.net/



```
    private String _BASED_URI = "http://";

    /**
     * Constructor of atomic component ObjectStorageGE
     */
    public ObjectStorageGEImpl(Map<String, Object> vals) {
        this._BASED_URI = (String) vals.get("based_uri");
    }

    /**
     * RetrieveObject service
     */
    public Map<String, Object> RetrieveObject(String auth_token,
            String container_name, String object_name) {
        // Preparing for invcation
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(_BASED_URI);
        WebTarget service = target.path(container_name).path(object_
        Builder invocationBuilder = service.request(MediaType.APPLI(
        Invocation invocation = invocationBuilder.buildGet();
        // Invoking the service
        Response response = invocation.invoke();

        // Process the response
        Map<String, Object> returnValues = new HashMap<String, Objec
        Integer status_code = new Integer(response.getStatus());
```

Fig. 13. A GE atomic component.eps

The generated source code shows how the RESTful service is invoked with the specified inputs.

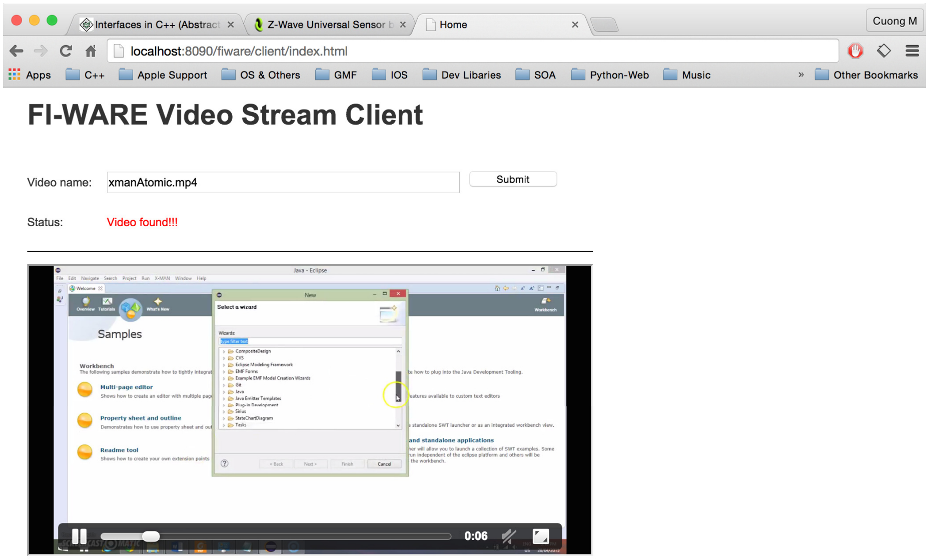
Once designed, components are deposited in a repository (bottom Fig. 13), and later retrieved to be composed into the application as in Figs. 10 and 11.

The source code of the application is then generated by our tool. As part of code generation, we specify the application to be packaged as a 'war' file, which is then deployed on a Tomcat[5] server. In order to test our application, we developed a simple web page (hosted by the same server) which provides an interface to take end-users' requests and pass them to the application. The media if found and returned from the application is then played via an embedded video player. The client is depicted in Fig. 14.

7 Discussion and Conclusion

We have defined and implemented an approach to developing GE-based applications based on an extended version of the X-MAN component model. Our approach presents novel heterogeneous composition mechanisms that exploit the power of the FIWARE ecosystem. Unlike heterogeneous composition in BPEL for REST, we compose software components with RESTful services. In comparison with JOpera, our applications are not limited to the middle tier of an MVC architecture. For instance, we can compose the application in our case study with an X-MAN component implementing a GUI that replaces the Web client. On another note, SCA does provide a component model for heterogeneous composition. However, unlike our approach, it does not provide explicit control flow. In addition, it requires a "glue" component to be developed afresh to act as coordinator.

[5] http://tomcat.apache.org/



FI-WARE Video Stream Client

Video name: xmanAtomic.mp4

Status: Video found!!!

Fig. 14. A client of the content delivery application.eps

Currently, applications can make use of existing FIWARE bundles which already composes certain GEs. The composition in these bundles is carried out by an ‘active’ GE such as `Orion` which implements the *Observer* pattern [10]. Orion is called by a ‘publisher’ GE and it then actively notifies ‘subscriber’ GEs. X-MAN composition connectors are Turing complete. It implies that we can use them to construct any complex control logics including the *Observer* pattern.

Our approach however has some limitations. The mapping we presented, although defined to be applied automatically, requires human effort to be performed. It is because currently GEs’ interfaces are described informally by textual documentation.Whilst this may be adequate for human consumption, the lack of a machine processable format such as WADL hinders our mapping. When this limitation is removed in the future, our approach can be improved accordingly.

GE catalogue currently supports programmatically access through a RESTful API. However, the results seem to be out of sync with the actual GEs. This poses a challenge for development tools like ours to integrate the GE catalogue with our component repository. Such integration when possible will allow seamless application development.

As future work, we plan to investigate an integration with FIA Project Management Plugin[6] to provide a complete environment for FIWARE users and developers. Finally, we intend to extend the X-MAN component model to support concurrency [9].

[6] http://catalogue.fiware.org/enablers/fia-project-management-plugin

References

1. Di Cola, S., Tran, C.M., Lau, K.K.: A graphical tool for model-driven development using components and services. In: Proceedings of SEAA 2015 - MOCS Track (2015)
2. Fazio, M., Celesti, A., Marquez, F.G., Glikson, A., Villari, M.: Exploiting the fiware cloud platform to develop a remote patient monitoring system. In: IEEE Symposium on Computers and Communications (ISCC). IEEE Computer Society, Larnaca, June 2015
3. Glikson, A.: Fi-ware: Core platform for future internet applications. In: Proceedings of the 4th Annual International Conference on Systems and Storage (2011)
4. Hadley, J.: Wadl (web application description language). GlassFish, WADL (2009)
5. Havlik, D., Soriano, J., Granell, C., Middleton, S.E., van der Schaaf, H., Berre, A.J., Pielorz, J.: Future internet enablers for vgi applications. In: Page, B., Fleischer, A.G., Göbel, J., Wohlgemuth, V. (eds.) EnviroInfo, pp. 622–630. Berichte aus der Umweltinformatik, Shaker (2013)
6. He, K.: Integration and orchestration of heterogeneous services. In: 2009 Joint Conferences on Pervasive Computing (JCPC), pp. 467–470. IEEE (2009)
7. He, N., Kroening, D., Wahl, T., Lau, K.K., Taweel, F., Tran, C., Rümmer, P., Sharma, S.: Component-based design and verification in X-MAN. In: Proc. Embedded Real Time Software and Systems (2012)
8. Lau, K.-K.: Software component models: Past, present and future. In: Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering, pp. 185–186. ACM (2014)
9. Lau, K.K., Ntalamagkas, I.: Component-based construction of concurrent systems with active components. In: Proc. 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2009), pp. 497–502. IEEE (2009)
10. Lau, K.-K., Ntalamagkas, I., Tran, C.M., Rana, T.: (Behavioural) design patterns as composition operators. In: Grunske, L., Reussner, R., Plasil, F. (eds.) CBSE 2010. LNCS, vol. 6092, pp. 232–251. Springer, Heidelberg (2010)
11. Liu, X., Hui, Y., Sun, W., Liang, H.: Towards service composition based on mashup. In: 2007 IEEE Congress on Services, pp. 332–339. IEEE (2007)
12. Marino, J., Rowley, M.: Understanding sca (2009)
13. Niemöller, J., Fikouras, I., de Rooij, F., Klostermann, L., Stringer, U., Olsson, U.: Ericsson composition engine-next-generation in. Ericsson Review **2**, 22–27 (2009)
14. Pautasso, C.: BPEL for REST. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 278–293. Springer, Heidelberg (2008)
15. Pautasso, C.: Composing RESTful services with JOpera. In: Bergel, A., Fabry, J. (eds.) SC 2009. LNCS, vol. 5634, pp. 142–159. Springer, Heidelberg (2009)
16. Pautasso, C., Alonso, G.: The jopera visual composition language. J. Vis. Lang. Comput. **16**(1–2), 119–152 (2005). doi:10.1016/j.jvlc.2004.08.004
17. Ramparany, F., Galan Marquez, F., Soriano, J., Elsaleh, T.: Handling smart environment devices, data and services at the semantic level with the fi-ware core platform. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 14–20, October 2014
18. Silver, B.: BPMN method and style, vol. 2. Cody-Cassidy Press Aptos (2009)

19. Stravoskoufos, K., Sotiriadis, S., Preventis, A., Petrakis, E.: Motion sensor driven gesture recognition for future internet application development. In: The 5th International Conference on Information, Intelligence, Systems and Applications, IISA 2014, pp. 372–377, July 2014
20. Villaseñor, E., Estrada, H.: Informetric mapping of "big data" in fi-ware. In: Proceedings of the 15th Annual International Conference on Digital Government Research, dg.o 2014, pp. 348–349. ACM, New York (2014). http://doi.acm.org/10.1145/2612733.2619954
21. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL. WS-reliable messaging and more. Prentice Hall PTR (2005)
22. Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding mashup development. IEEE Internet Computing **12**(5), 44–52 (2008)