On the Role of Context in the Design of Mobile Mashups

Valerio Cassani, Stefano Gianelli, Maristella Matera, Riccardo Medana, Elisa Quintarelli, Letizia Tanca, and Vittorio Zaccaria

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
{valerio.cassani,stefano.gianelli}@mail.polimi.it, name.surname@polimi.it

Abstract. This paper presents a design methodology and an accompanying platform for the design and fast development of Context-Aware Mobile mashUpS (CAMUS). The approach is characterized by the role given to context as a first-class modeling dimension used to support *i)* the identification of the most adequate resources that can satisfy the users' situational needs and *ii)* the consequent tailoring at runtime of the provided data and functions. Context-based abstractions are exploited to generate models specifying how data returned by the selected services have to be merged and visualized by means of integrated views. Thanks to the adoption of Model-Driven Engineering (MDE) techniques, these models drive the flexible execution of the final mobile app on target mobile devices. A prototype of the platform, making use of novel and advanced Web and mobile technologies, is also illustrated.

Keywords: Mobile Mashups, Mashup Modeling, Context Modeling, Context-aware Mobile Applications, GraphQL

1 Introduction

The data deluge we are confronting today virtually drives people to continuously search and discover new information. The opportunity to access a large amount of information, however, does not always correspond to the growth of people knowledge. Many times, indeed, one does not know how to filter data "on-the-fly" to obtain the information that is the most suitable to the current context of use. This aspect is even more critical for mobile devices. Smart phones, for example, are often used to satisfy "quickly" very contingent information needs. Also their reduced screen size and battery power do not favor neither visualizing huge data sets nor executing multiple progressive queries to filter out irrelevant data.

Given this evidence, our research focuses on the definition of a methodology and related tools for the semi-automatic design and development of *Context-Aware Mobile mashUpS* (CAMUS)[1]. CAMUS leverage the results of two main research lines, related to the design of context-aware systems and mashups, with the aim to support developers in the creation of flexible apps that dynamically gather and combine data from heterogeneous data sources and filter and adapt

the integrated content to the users' situational needs. With respect to traditional applications, designed to satisfy predefined requirements, the CAMUS added-value is their intrinsic capability of identifying pertinent data sources, i.e., adequate with respect to the current users' needs, and pervasively presenting them to the final user in the form of context-aware integrated visualizations deployed as mobile apps. This application paradigm overcomes the limits posed by pre-packaged apps and offers to the users flexible and personalized applications, whose structure and content may even emerge at runtime based on the actual user needs and situation of use.

In this paper we show how CAMUS design and development can be concretely based on a set of high-level abstractions for context and mashup modeling. In particular, we will present a novel design methodology and related tools for fast prototyping of mobile mashups, where context becomes a first-class design dimension supporting: *i)* the identification of the most adequate resources that can satisfy the users' information needs and *ii)* the consequent tailoring at runtime of the provided data and functions. We start from two consolidated approaches for context modeling [2, 3] and mashup modeling [4] and show how the synergies of the two approaches can be amplified to define a new design methodology for the fast prototyping of flexible mobile apps.

This paper is organized as follows: Section 2 clarifies the motivations of our work and summarizes the main elements that characterize our design methodology by also comparing it with other similar approaches. Section 3 describes the main design steps based on the adoption of two consolidated approaches for context and mashup modeling, which are however integrated and somehow revisited or augmented to comply with each other's features. Section 4 illustrates the organization of the resulting framework, as well as the architecture and the implementation of the related platform supporting both the design of CAMUS apps, by means of visual design environments, and the context-aware execution of the generated mobile apps. Section 6 summarizes the main features of our design framework in relation to some classifying dimensions adopted during the Rapid Mashup Challenge. Section 7 then shortly describes the demo given during the challenge. Section 8 finally outlines our conclusions and describes our future work.

2 Rationale and Background

The CAMUS project merges techniques coming from two areas whose role is fundamental for the solution of problems related to the design of mobile systems. The first area deals with the issue of information overload by introducing *tailoring* techniques based on context-awareness, while the other one addresses the seamless integration of data and services. From different perspectives, both areas promote the creation of flexible mobile applications that dynamically gather and combine data from heterogeneous data sources, supporting the users' situational needs.

2.1 Context Awareness

The research on context made a significant step forward in the 90's, when the research community raised the problem of representing context-aware user and system activities [5]. While the community of computer science professionals initially perceived the context only as a matter of user location and time, this notion has been extended including, in the idea of context, other personalization aspects like current user interests, current role of the user in the system, the company the user keeps at the moment, and possibly other situational dimensions that may depend on the specific application at hand [6].

In CAMUS, the perspectives that characterize the different contextual situations in which the users can act in a given application scenario are modelled by means of the so-called *Context Dimension Model* [3], which provides the constructs to define at design-time the Universal Context Dimension Tree (*Universal CDT*). As represented in Figure 1(a), the Universal CDT is a hierarchical structure consisting of *i) context dimensions* (black nodes), modeling the different perspectives through which the user perceives the application domain (e.g., *time*, *interest topic*, *transport*), *ii)* the allowed *dimension values* (white nodes), i.e., the values used to tailor the context-aware information (e.g., "morning", "with car", "culture"), and *iii) variables* (e.g., "geographic coordinates" for a *location* dimension), that are either custom values supplied by the user at run-time or data acquired by device sensors (e.g., the current GPS coordinates of a given device). The dimension values and the variables are also called *context elements*. Note that the adoption of a hierarchical structure allows us to employ different abstraction levels to specify and represent contexts.

Any sub-tree of the Universal CDT with at most one element for each dimension represents *a possible user context*. Figure 1(b) shows a possible context for the Universal CDT of Figure 1(a).

The CDT was originally introduced to tailor, at design time, the contextual portions of a global database, in order to grant to the users run-time, context-aware access to huge datasets. In this paper we will describe how, when a certain context is detected at run-time by means of device sensors or using some information provided by the user, the context-relevant services are invoked to build a service mashup appropriate for the identified context.

2.2 Mobile Mashups

Mashups are "composite" applications constructed by integrating ready-to-use functions and content exposed by public or private services and Web APIs [7]. The mashup composition paradigm was initially exploited in the consumer Web for creating rapidly simple Web applications that reused programmable APIs and content scraped out from other Web pages. Soon the potential of this lightweight integration practice emerged in the other domains where the possibility to create rapidly new applications, also by laypeople, is an important requirement. In the last years many efforts have been devoted to the definition of usable and intuitive composition paradigms. This aspect is indeed considered a factor enabling the addition of significant new value with respect to other development

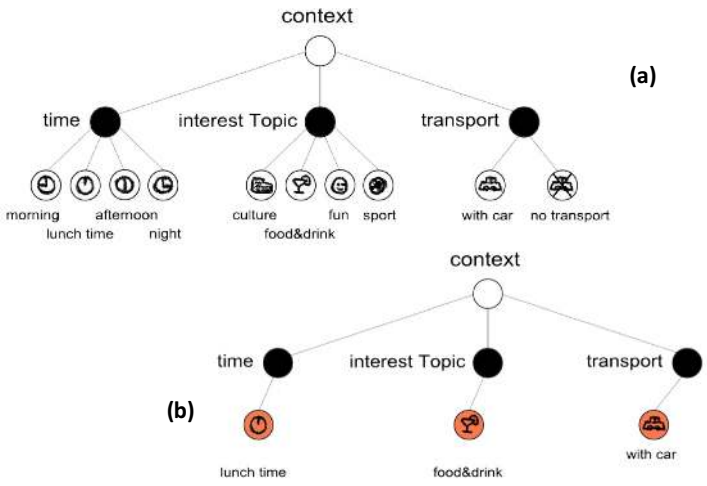


Fig. 1. Example of Universal CDT and a possible context.

practices. Intuitive notations and visual design environments can offer the advantage for designers, or even end users, to achieve effective applications that match exactly their needs and that can be created in a short time by simply reusing and customizing existing resources. Therefore, several approaches have proposed composition paradigms based on visual notations that abstract relevant mashup development aspects and reduce- or sometimes totally eliminate, the need of programming.

Among the proposed approaches for mashup design, very few specifically concentrate on mobile mashups. In [8] the authors illustrate a mobile generator system that aims to support fast prototyping as it is able to automatically generate a large part of the application code. However, this approach does not support content integration, while we believe this is a fundamental feature for the mobile usage context where integrated views can greatly improve the information access experience. Also, it proposes a domain specific language with abstractions that are very close to the ones of the Android execution platform. The approach indeed focuses exclusively on Android apps and does not exploit modeling as a means to abstract from specific technology and achieve multi-platform deployment.

Recently proposed services, like IFTTT (If This Than That - `https://ifttt.com/wtf`) and Atooma (`http://www.atooma.com`), enable users to synchronize the behavior of different apps through simple conditional statements. However, they do not support at all the integration of different data sets and of the corresponding UIs.

For the design of CAMUS apps, we adopt the approach presented in [4]. It is based on a UI-centric paradigm for data integration, as it requires acting

directly on the user interface of the mashup under construction, in a kind of live-programming paradigm where each composition action corresponds to a data integration operation that generates an immediate visual feedback on the artifact under construction [9, 10]. One of its distinguishing features is the capability of abstracting from the specific technologies of the target applications. In line with the Model-Driven Engineering (MDE) philosophy, it indeed leverages on the generation of application schemas, and on their interpretation in different execution platforms by means of engines supporting the generation of code for native application. This is a very relevant feature: recent studies on device and traffic share report on a generally observed attitude of users to access applications through different devices (desktop and mobile)[11].

2.3 Context-aware Mobile Mashups

The literature reports different experiences for the development of context-aware mobile applications, showing how applications can be extended to gather and use context at run-time (see for example [12]). However, these works consider context-awareness as an orthogonal dimension, to be programmed *ad-hoc* for any application, while they do not provide conceptual models and design frameworks.

In [13] the authors show how a mashup design environment may implicitly provide support for context-awareness, thanks to the introduction of mashup components in charge of managing context, i.e., capturing context events and activating related operations in other components of the mashup. Although effective, the approach does not provide any abstraction to model the context; the designer is in charge of configuring the context components (basically location and time) by means of parameter settings. We instead assign a fundamental role to usage-situation modeling, from which we then derive the logics for selecting services and dynamically build the final applications.

MyService is a mashup design framework that supports the creation of context-aware services based on rules [14]. It provides an Android design environment that allows end users to select pre-defined context-based recommendation rules on top of a service directory. Proper services are thus selected depending on the context gathered at runtime, and the code of a mashup is generated. This approach is in line with our idea to filter at runtime services by means of a context representation. However, MyService focusses especially on location-based adaptations, while we are able to cover any dimension that can filter content. The CDT model indeed is generic with respect to the specific domain, allowing for the representation of all possible perspectives that characterize context by means of the generic concept of *context dimension*. Also, MyService does not support data integration and it is not clear whether the generated code also covers the rendering of User Interface views. In the following sections we will show how we address this point – which is crucial especially when different execution platforms are addressed – by means of advanced technologies that instantiate views in the mobile app starting form an abstract schema of the integrated data set to be provided by the app.

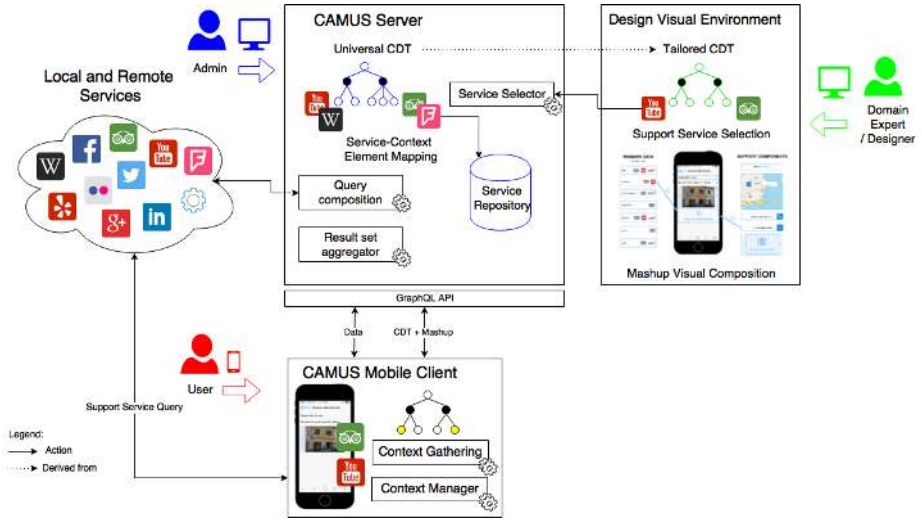


Fig. 2. Main system components and supported design and execution activities in CAMUS.

3 The CAMUS Methodology

We now present the design methodology for the creation of CAMUS apps starting from the specification of context requirements. Our approach is characterized by the adoption of design environments that, in line with recent approaches to visual programming of mashups, make intensive use of high-level visual abstractions [15, 4]. Visual paradigms indeed hide the complexity typical of service composition, data integration and the programming of context-aware mobile apps, and assist CAMUS designers (even if non-experts in these technologies) in the creation of multi-device personalized applications.

Figure 2 represents the general organization of the design framework and highlights the flow of the different activities and related artifacts that enable the transition from high-level modeling notations to running code. In the sequel, we will describe the activities performed by three main personae, the *administrator*, the *mashup designer* and the *app user*, who are the main actors interacting with the framework at different levels and with different goals. In order to exemplify how these activities are carried out, we will refer to a case study in the domain of tourism, characterized by: *i)* a tourism *service provider*, who sets up an ecosystem of tourism services and the platform for the delivery of CAMUS apps; *ii)* the *tourist*, i.e., the end user of a CAMUS app created on top of the available services; and *iii)* a *tour agent*, i.e., an intermediary player who assists the end user in the creation of the specific tour and, consequently, acts as mash-up designer customizing the CAMUS app according to preferences related to the specific trip and person - which might not be entirely captured by the Universal CDT.

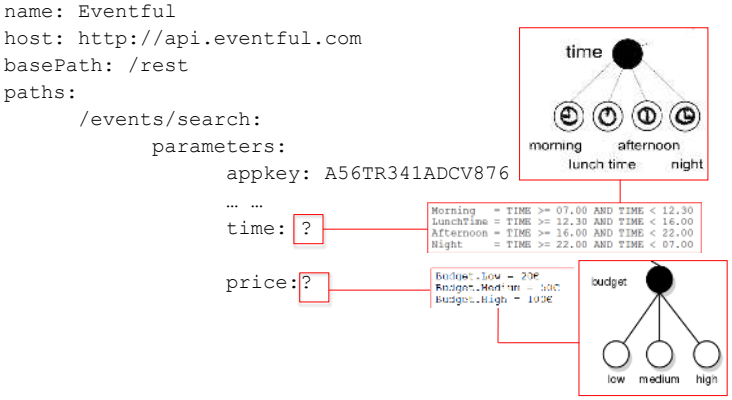


Fig. 3. An excerpt of a service descriptor specifying properties for service invocation.

3.1 Creation of the Service Ecosystem

The *administrator* is in charge of managing the CAMUS server and resources. One of the main roles is to create and maintain the *service repository*. S/He registers into the platform distributed resources (remote APIs or in-house services) that are pertinent with respect to a given domain, as resulting from the specific requirements and from the requests of the final users. For example, in the tourism domain the administrator will register services that provide information about hotel, restaurants, points of interest, and any other information useful for a trip. Service registration is taken by creating descriptors that specify:

- *How the resources are to be invoked*, e.g., the service endpoint, its operations and input parameters. In this phase, some parameters can be bound to wrappers that perform transformations from symbolic context values gathered at runtime to corresponding numerical service input. Figure 3 reports an excerpt of a descriptor for a service returning data on events. The input parameter `price` is associated with a wrapper that transforms symbolic terms, such as `low`, `medium` and `high` specified as user preferences, into specific price values, as expected by the service.
- *The schema of the responses of the returned service.* To ensure homogeneity of data formats, needed to merge the data that must be visualized by the final app, the response schema of each registered service is annotated with *terms* (e.g., *title*, *description*, *address*) indicating categories of attributes, according to a vocabulary that is defined and maintained in the service repository. These annotating terms have a double role: when the mashup is defined (see Section 3.3), they allow the designer to select service attributes by reasoning on abstract categories, instead of specific attributes resulting from service queries; at run time they assist the merging progress of different result sets, since it is easier to identify attributes that refer to the same entity properties.

For these annotations we currently assume the availability of a set of *ad-hoc* defined category tags, which the administrator explicitly associates to the attributes returned by registered services. Domain ontologies can of course be exploited as well to automatically associate service attributes to semantic terms.

3.2 Universal CDT augmentation

The *administrator* also specifies the Universal CDT, providing a representation of all the possible usage contexts. In order to support the context-aware selection of services at runtime, s/he augments the Universal CDT by defining *mappings* between the identified context elements and the services registered in the platform.

Services belong to two different categories. *Core services* provide the main data that contribute to forming the core content of the final app. As represented in Figure 4, they are associated with the so-called *primary dimensions*, i.e., dimensions for whose values some "primary' content" has to be provided in the final application. Such content is considered as primary as it is supposed to be the main object of the users' requests. For example, services providing data on restaurants are associated with the `food&drink` value of the `interest topic` dimension. Therefore, their selection at runtime occurs if the `food&drink` dimension is the emerging user interest in the identified context.

Support services supply auxiliary content (e.g., the meteo condition or the public transportation in a given location) or functionality (e.g., the localization on a map of a restaurant retrieved by some core service).

Do consider that, when the app is working, the available support services may vary depending on the usage context. This means that, during the Universal CDT augmentation, the association is operated *at the service category level*. For example, a "transport" service category is associated with a given context node (e.g., `food&drink`) to represent that, within the final mashups, transport services will be selected when the user's context is characterized by the `food&drink` interest topic. Then, at runtime a specific service belonging to this category will be selected and invoked, based on the identified geographical area. This requires specifying, within the service descriptor, the category the service belongs to and its characterization with respect to the context values its final selection depends on .So, during The Universal CDT augmentation, the administrator characterizes the role that every node in the Universal CDT plays in the runtime selection of services, by assigning two typologies: *filter* or *ranking*. By default, all the nodes have a *filter* role, meaning that reaching them while traversing the tree implies adding a corresponding filter for service selection. The administrator can characterize some nodes as *ranking* when they can provide a ranking criteria for sorting different candidate services. For example, *location* is a ranking node because its related context parameters might influence the selection of services that provide relevant data in a given geographical area(e.g., the information services provided by the local transportation company of a certain city are to be preferred to a generic transport information service).

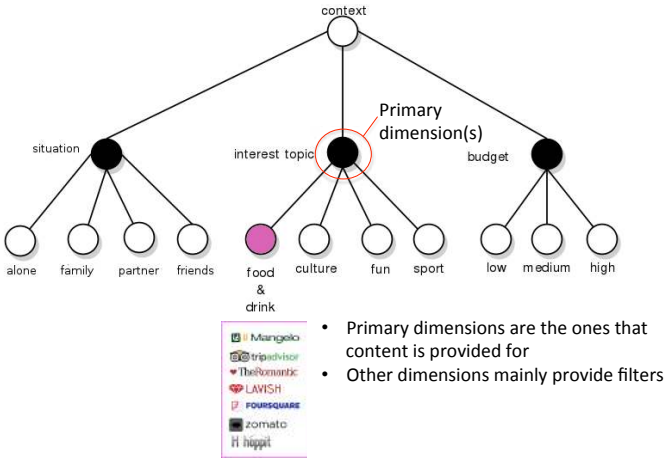


Fig. 4. An excerpt of a service descriptor specifying properties for service invocation.

3.3 Mashup Visual Design

The *mashUp designer* starts from the image of the available resources represented by the augmented Universal CDT and, using a *Design Visual Environment*, defines a *Tailored CDT* by further refining the selection of possible contexts and the mapping with services (both core and support). This is needed to fulfill the needs and preferences of specific users or user groups.

Given the services associated with a given context dimension (e.g., all the services providing data on restaurants associated with the `food&drink` context dimension) the designer can select the categories of attributes (i.e., the annotating terms specified at service-registration time) to be visualized in the mobile app. As schematically represented in Fig. 5, this selection is operated visually, according to a composition paradigm for mobile mashup creation already defined and implemented in the PEUDOM mashup tool [4]. The designer drags and drops the semantic terms associated with the attributes of the service response. A "virtual device" provides an immediate representation of how the final app will be shown on the client device. In addition, the designer can include *support services* that can provide additional information and enrich the user experience (e.g., provide transport indications to reach a restaurant, or extend the core content with descriptions of places taken from Wikipedia). Support services are also context dependent: for instance, if the user expresses that s/he is in a situation where s/he wants to use "transportation by car", the system provides route information; otherwise, if s/he selects "public transport" it suggests a bus line. This requires that also for support services the inclusion of data attributes be operated by exploiting annotating terms describing attribute categories exposed by classes of services, not the actual data attributes exposed by single, specific services. At runtime, the binding defined at service registration time between semantic terms and service attributes will be exploited to query the services actually selected.

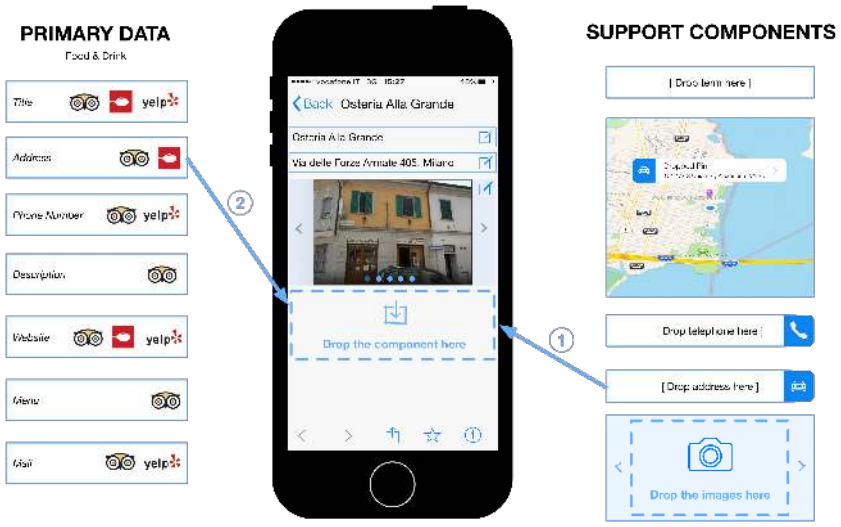


Fig. 5. Schematic representation of the visual mapping activities to associate service attribute classes to elements of the final app UI.

All the visual design actions are translated by the design environment in a JSON-based *mashup schema*, which specifies rules that at runtime guide the instantiation of the resulting app and the creation of its views.

It is worth noting that, in comparison to other approaches to mashup design [7], the composition activity and, more specifically, the selection of services are not exclusively driven by the functional characteristics of the available services or by the compatibility of their input and output parameters. Rather, the initial specification of context requirements enables first the progressive filtering of services and then the tailoring of service data to support the final situations of use.

3.4 App Execution

The *CAMUS (app) users* are the final recipients of the mobile app that offers a different bouquet of content and functions in each different situation of use. When the app is executed, the context elements that characterize the current situation, identified by means of a client-side *sensor wrapper* or explicitly selected by the user, are communicated to the server; this, in turn, chooses the pertinent services to be invoked and returns an integrated data set that includes the attributes corresponding to the semantic terms selected during the mashup design. The mashup schema created by the designer is thus interpreted locally on the device (by means of a *Schema Interpreter*), and the generated views are populated with the returned data as defined during the visual mapping step. The platform indeed exploits generative techniques: modeling abstractions guide the design of the final applications, while generative layers mediate between high-level visual

models and low-level engines that execute the final mashups. Execution engines, created as hybrid-native applications for different mobile devices, then make it possible the interpretation and pervasive execution of schemas.

4 Platform Organization and Implementation

In order to support the CAMUS methodology, we developed a proof-of-concept prototype; the demo given at the Rapid Mashup Challenge focused on it. Its architecture is server-centric, meaning that a *Server* manages the main functions for the execution of the mobile app, i.e.: *i)* analyzing of the user's context, detected through the mobile device, to select the services to be queried and, *ii)* querying the selected services and transforming their results into an integrated data set to be rendered by the mobile app.

The Server exposes several endpoints to enable the execution of service queries as well as CRUD operations on other system data, such as users' profiles, and the descriptors for the Universal CDT and the service repository. The framework used for its implementation is Node.js and the database is MongoDB. The main API invoked by the mobile client to access the server functionality is compliant with the GraphQL API specification [16]. GraphQL offers a layer that enforces a set of custom-defined typing rules on the data sent and received via HTTP. Besides, it provides a flexible way to specify the response format, by making it easier to support different generations of APIs.

The *Visual Design Environment* consists of a suite of Web applications to: *i)* easily register new services to the system, *ii)* specify visually (and automatically generate an internal representation of) the CDTs and the associations of services with pertinent nodes, and *iii)* design visually the final mashups and automatically generate their schema.

The *Client App* is the front-end enabling the interaction of the end-user with the whole system. During its initialization, the app loads the user CDT and the JSON-based specification of mashup schemas to be rendered from the server. The schema specifies the structure of the app views and drives their instantiation. The transformation of the schema into concrete views exploits React Native [17], a framework recently introduced by Facebook to streamline the production of cross-platform mobile apps. The app logic is written in Javascript and, for the most part, is agnostic with respect to the target platform. React enforces a pseudo-functional/reactive approach that involves a central state (which holds the *model* of the application) and a number of pure functions that render the view. The view elements, in turn, can produce actions that act on the state through a dispatcher, while network responses represent another source of actions that can change the state. The state of the app serves the rendering of the views and their data: it is mainly composed of the mashup data, the current interest topic, the CDT and the result of the current context-based query.

A typical *request* from the client is composed of a JSON payload that describes the *context* and a specification of the format of the data that is expected by the client. As represented in Figure 6, the request is thus processed through the following steps:

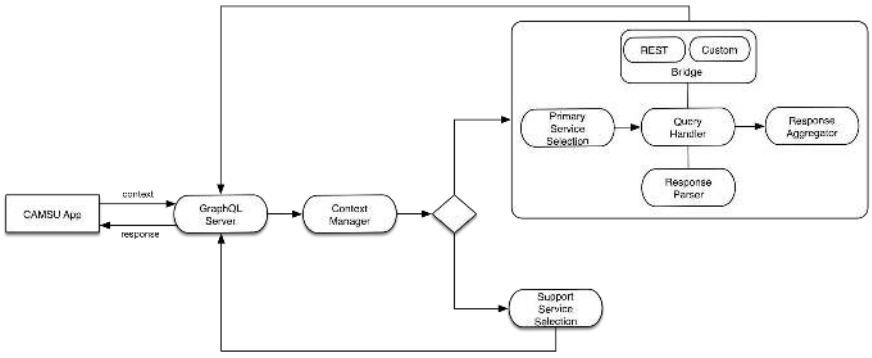


Fig. 6. Server request flow.

- The *Context Manager* parses the context and "decorates" it with all the Augmented UCDT information (services, ranks, etc.) related to its elements.
- Based on the analyzed context, the *Primary Service Selection* component selects the services to be queried.
- The *Query Handler* queries the selected services by using service-specific bridges that wrap the retrieved result sets and transform them into a common internal representation that complies with the semantic terms associated to the different service attributes. This internal representation enables merging the different data sets based on attributes associated with the same terms.
- Finally, the activation of support services – if any –, is bound to the selection of specific attributes in the integrated result set, as defined by the mashup designer when creating the mashup.

The rest of this section will illustrate in more details the steps involved in the request flow.

Primary service selection. As mentioned above, the *Context Manager* takes care of decorating the user's current context with the information coming from the Augmented Universal CDT. Service selection is thus operated by interpreting the request context as a key-value query, and using this representation to "navigate" through the Universal CDT specification. The result of this navigation is the set of *Service Associations* (*SA*) found in the different visited nodes, that suggest the use of service that can be pertinent with respect to the current context. Each association is composed by the priority that characterizes the service for the reached node and the node weight. These values are set when the Universal CDT is modeled: the priority is an increasing integer starting from 1, and weights are predefined values assigned with the constraint that a filter

node's weight must be less than that of a ranking node. The final relevance value for each service s is thus computed from the weights w_i and node priorities p_i as:

$$R_s = \sum_{i \in SA(s)} \frac{w_i}{p_i} \tag{1}$$

The obtained value is used to rank and filter the N top relevant services for the query.

Query handling. The *Query Handler* is in charge of identifying the queries to be posed to the selected services; a number of bridges than actually invoke the services by formulating the queries in accordance with the protocol exposed by the service APIs. We supply a default bridge for REST-type services plus an abstract class that can be extended for implementing new bridges covering further service types.

A bridge receives the service descriptor provided by the Query Handler and builds the URL where the service should be queried. During this composition, the bridge uses the context to retrieve the list of parameter nodes which, in turn, store the values that are needed to perform the query. When all the necessary queries are completed, it sends the responses obtained back to the Query Handler.

Response aggregation. The *Response Aggregator* executes two main tasks: *i)* merging items from different services that refer to a same instance and *ii)* scoring each instance. In fact, two or more services might return data referring to a same instance, thus duplicate identification is needed to discover equal or similar instances and fuse them into a unique object. The fusion then might produce a richer set of attributes for an instance, as one service can provide attributes not supplied by the others.

Merging is computationally intensive, since it requires pairwise comparison of all the instances in any of the service result set. To reduce this complexity we devised some optimizations: first, each instance item is classified on the basis of the phonetic code of its key attribute (for example, the title), using some phonetic string matching metrics[1]. Then, inside each class, pairwise comparison of the common attributes is used to compute a similarity index. If this value is greater than a predefined threshold, the two items are considered similar and they are fused together. The complexity of this comparison strategy is $O(n)$ (i.e., linear in the number of analyzed instances).

Support service selection. The selection of support services is similar to the one operated for primary services. However, a support service is selected and included in the mashup if and only if all the bindings defined between the mashup core data and the operations exposed by the support service, as defined by the

[1] Our current prototype uses the Chapman's Soundex metrics[18].

mashup designer, are satisfied. This avoids runtime invocation of services that are not applicable in a particular context, for instance because the needed input parameters are not provided by the integrated result set or by the usage context. The result of the support service selection is a set of service endpoints that are communicated to the client within the mashup schema, so that the mobile app can directly invoke the services to retrieve and visualize the auxiliary data.

App life cycle. At the application startup, the user chooses the current interest topic. The context selection page supports the user in editing the current context, and also probes the hardware for sensors data. When the user finalizes the context input, a GraphQL query is built and sent to the server. The request specifies the structure the incoming data should have in order to be rendered in the results page. An important difference with respect to a more traditional approach like REST is that different clients can request different data formats from the same end point. Once received, the data is stored in the main application state and the app view is re-rendered by hydrating a React Native template.

The view schema provides a very flexible mashup design. As reported in Figure 7, every page is associated with the corresponding key in the file (e.g.: `results`, `details`, ...) and, at render time, the view builder loads the schema dedicated to the rendering of data for the current topic (tag `topics`); potentially, the app is able to render a different view for each possible topic. The tag `contents` specifies the view elements; thanks to the `style` attribute it is possible to pass directly to the app CSS-like style attributes used in React. The elements within the `contents` tag are defined recursively, thus enabling a very customizable design of the app: in principle, any single view element can be defined in this way and then dynamically instantiated.

5 Evaluation

In this section we provide a preliminary characterization of the performance of the system. Since the application is still under active development, the numbers shown here are to be considered with care. However, we think that they provide some interesting insights on the feasibility of context-aware strategies for service selection and querying, as the ones illustrated in the previous sections.

System and workload model. To model the system, we use a basic M/G/1 queue [19]. In fact our system behaves as:

- M/*/*: a service node where request arrival follows a *markovian* process, i.e. requests arrive continuously and independently at a constant average rate λ. We will use this assumption in the characterization of the response time.
- */G/*: the service rate distribution is not yet known, so we assume it being a general distribution with fixed mean and variance.
- */*/1: a single process (Node.js) serves incoming requests.

The system used for workload evaluation is characterized by an Intel Core i5-5257U CPU, with 2 cores and a 3GHz frequency, a 8 GB DDR3 RAM, and an SSD disk of 128GB.

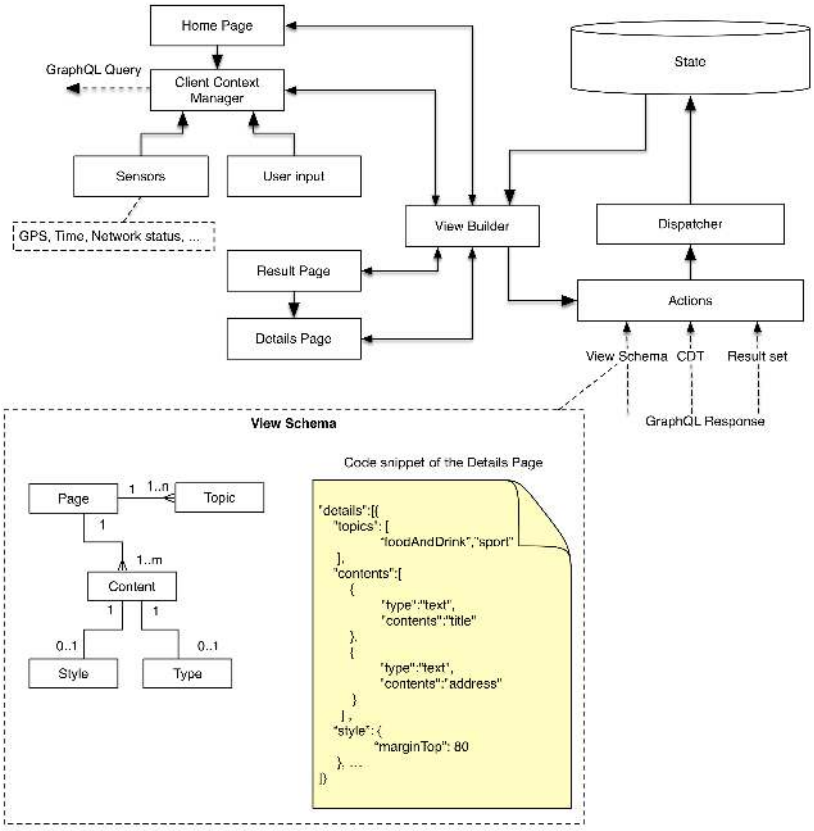


Fig. 7. App data flow.

Service time. The service time is the time it takes for a single request to be served. To better understand the distribution of the service time (which has been assumed as *general* in the previous paragraph), we use a sequence of 500 back-to-back requests, where each request is sent once the previous one has been served. Requests are served by the system with a first-come/first served (FCFS) policy. We stubbed the *query handler* in such a way as to measure just the internal delays of the system components.

Figure 8 shows the histogram of the measured response time. To a first inspection, the shape of the distribution seems to agree with a log-normal distribution whose parameters are $\mu = 202(ms), \sigma = 6.4(ms)$. This suggests an ability to sustain almost 5 requests per second. We use this information to generate a workload of independent requests.

Response time. When receiving independent requests (which can arrive before the current one is effectively served), the system can show a delay due to

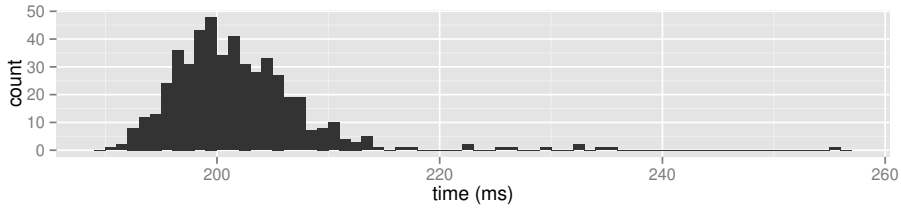


Fig. 8. Distribution of the service time.

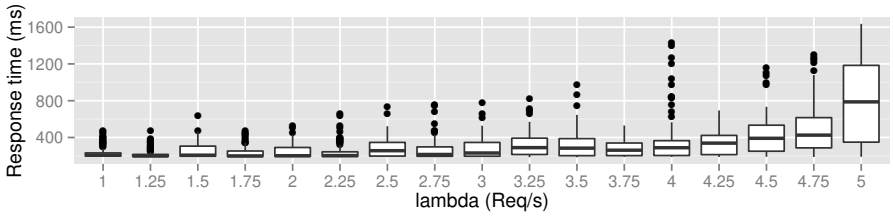


Fig. 9. Distribution of the response time under varying workload.

requests queuing up. To characterize the behavior under this type of workload, we generate a sequence of requests using an exponential arrival-rate distribution. The exponential distribution is in fact congruent with the markovian arrival-rate assumption made above:

$$f(x;\lambda) = \lambda e^{-\lambda x}, \text{where } x \geq 0$$

where λ characterizes the rate of generation of independent requests and x is the time between one request and the next.

Figure 9 shows the box-plot charts for a varying request rate, from 1 to 5 requests per seconds (saturation threshold). As can be seen, the system exhibits a robust response up to $\lambda < 4$. After that point, both variance and mean of the response time exponentially diverge, approaching the saturation point individuated in the previous paragraph.

Discussion. The above analysis brings us to an interesting insight which we are going to investigate further in our work: the service time is log-normally distributed. This type of distribution is characteristic of a process which is a product of many independent random variables. Our conjecture is that this could be due to the way in which the response elaboration has been split across the components, thus the software composition might play a role in the performance of the system. This is however a preliminary observation that needs to be corroborated by means of wider and deeper investigation.

6 Features and Level of Maturity

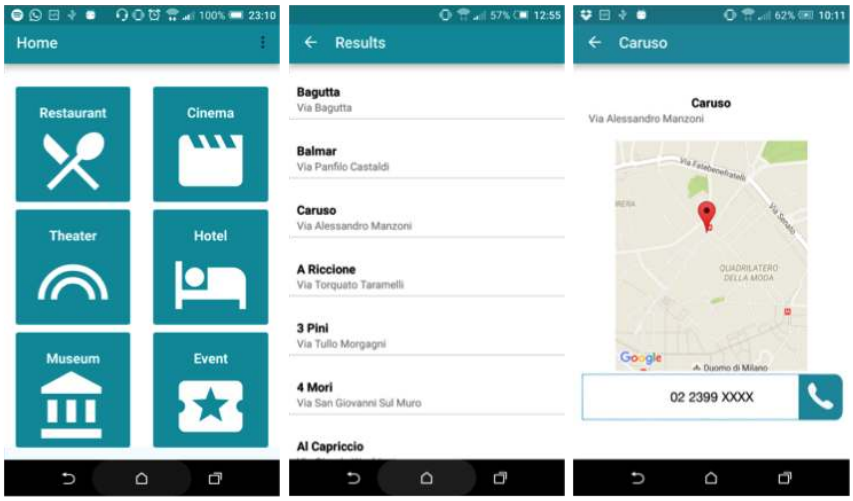
We now summarize the main characteristics of our design approach in relation to some dimensions for the classification of mashup paradigms used at the Rapid Mashup Challenge. Our approach mainly supports the generation of *data mashups*. The platform backend indeed exploits data fusion techniques to combine the data sets extracted from the selected core services. This process involves mainly the invocation of *data components*. However, the generated mashups also include mechanisms for *UI synchronization*: an event-driven logic at the UI level is used to activate the generation and synchronization of different views in the app. For example, when an item is selected in a view displaying core data, a new view is instantiated to display related support data.

The execution of the generated mashup is *distributed between the mobile app and the backend server*. Besides capturing the context parameters and composing the requests to the backend, the app is also in charge of interpreting the mashup schema and instantiating corresponding views. The identification of pertinent services and the generation of the mashup schema is however executed by the server. The integration logics is based on an *orchestration of the involved services*: decoupled components are indeed invoked by the Query Handler, according to a schema determined by the selection of the primary and support services. The resulting mashups are *short-living* as the involved services and the way they are integrated in the app depend on the current context, which is reconsidered at any new requests generated by the app. However, the app allows the user to materialize on the server some data, like reservation data and tickets, that can be useful for future sessions.

The tool assisting the design of CAMUS apps targets *both developers and non-programmers*: the former (e.g., the platform administrators having experience with service registration and requirements modeling) are supposed to prepare the instruments (e.g., the CDT specification) that the latter can visually refine according to a *WYISIWIG visual language*, to better capture the requirements of the specific usage domain. The design tool then offers a *full automation* for the generation of the final app starting from the schemas visually defined by the mashup designers.

Any change to the schemas requires a *re-initialization of the final app*. Our composition approach, indeed, supports live programming during the app design, as it offers a preview of the final app in a virtual device where each composition action is "materialized" into a change visible in the app UI. However, once an application schema is downloaded on a mobile device, the local execution engine keeps instantiating the app according to that schema. Possible changes will be applied only if a new execution of the app starts (and a new schema is downloaded on the device).

While we have a stable implementation of the platform backend and of the app runtime, we still need efforts to achieve an integrated visual design environment. At the time of the demo at the Rapid Mashup Challenge, all the JSON descriptors (service descriptors and CDT representation) had to be written manually. We have now started developing visual editors for service registration and for the CDT specification. The PEUDOM visual editor [4] assists the visual map-



(a) View for the Interest Topic selection

(b) View displaying search results

(c) View for data and functions offered by support services

Fig. 10. App screenshots from the demonstration.

ping activity for the design of the app views; however, we still need to reconcile the syntactic format of the mashup schema generated by the design tool with the one required by the current app execution engine.

7 Demo at the Rapid Mashup Challenge

As explained above, we still do not have an integrated visual design environment, since our research so far has especially focused on proving the feasibility of the approach for the context-driven, dynamic selection of services. Therefore, at the Rapid Mashup Challenge we illustrated the main features of the design methodology: we emphasized the role that context modeling plays in the dynamic selection of services and showed that the context-driven, dynamic construction of mobile mashups is feasible. The dynamic generation of CAMUS apps starting from the representation of possible usage situations is indeed the most characterizing feature of our approach.

During the presentation we illustrated the steps needed to set up the service ecosystem, as well as the mechanisms that, starting from the context captured at run-time and the representation of the augmented CDT, enable *i)* the selection of pertinent services, *ii)* the production of a result set integrating data extracted from the single services, and *iii)* the dynamic generation of the mobile app views for data visualization.

Figure 10 shows the sequence of app views that were shown during the challenge to illustrate such mechanisms. The first view (Figure 10.a) allows the user to select an interest topic; this selection plus other parameters characterizing

the usage situations (e.g., time and geographical position) are sent to the platform back-end and trigger the selection of pertinent services. In case of multiple selected services, data fusion procedures are also executed; thus the composed result set is sent back to the client, where it is displayed through the view illustrated in Figure 10.b. Finally, the view in Figure 10.c shows further details made available by support services (in the example GoogleMaps and the device-local dialing service) for an item selected in the previous view.

The steps undertaken by the server that correspond to the previous interaction flow, and especially the actions to manage the context and to instantiate the app views, can be seen in the video available at: `https://www.dropbox.com/s/nitnsehsv38x5co/demo\%20camus.mov?dl=0`.

8 Conclusions

This paper illustrates the CAMUS methodology and its related platform, whose aim is to empower non-expert developers to create context-aware, mobile apps by integrating multiple and heterogeneous APIs acting on situational needs. It discusses in particular the major role that the CAMUS design approach gives to context modeling. The specification of the Universal CDT is the central design activity; around it the construction of the mashup is performed. At design time the designer defines mashup schemas by reasoning at a high level of abstraction on possible context dimensions and associated service categories; at execution time, specific services are dynamically selected and integrated by taking into account the actual user context. The paper also discusses how taking into account context elements in the automatic instantiation of the final app, and especially selecting on the fly the services to be queried, is feasible and does not affect the performance of the server components.

This paper does not discuss the usability of the design methodology (i.e., how the methodology is perceived by designers) and the usability of the generated apps (i.e., if they are considered useful and usable by the end users). However, since the CAMUS framework still exploits the composition paradigms and the final app organization that we already defined in our previous work, we capitalize on the large body of data and user feedback collected in the last years through families of user studies (see for example [15, 4] for an extensive discussion on the conducted evaluations). Our current work is devoted to refining the implementation of the platform, and especially to defining a tight integration among the different visual design environments that we developed so far. Other efforts are being devoted to the formal characterization of the operations for service selection and composition based on the CDT representation.

Acknowledgments

This research is partially supported by the research grants FluidCAMUS, funded by Aliday S.p.A., and SHELL (CTN01 00128 111357), funded by the Italian Ministry for University and Research - MIUR. We like to thank the large group of students of Politecnico di Milano who enthusiastically contributed to the design

and implementation of the first CAMUS prototype. They allowed us to assess the feasibility of revising mashup composition practices through the introduction of context modeling concepts.

References

1. Corvetta, F., Matera, M., Medana, R., Quintarelli, E., Rizzo, V., Tanca, L.: Designing and developing context-aware mobile mashups: The CAMUS approach. In Cimiano, P., Frasincar, F., Houben, G., Schwabe, D., eds.: Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings. Volume 9114 of Lecture Notes in Computer Science., Springer (2015) 651–654
2. Bolchini, C., Orsi, G., Quintarelli, E., Schreiber, F.A., Tanca, L.: Context modeling and context awareness: steps forward in the context-addict project. IEEE Data Eng. Bull. **34**(2) (2011) 47–54
3. Bolchini, C., Curino, C., Orsi, G., Quintarelli, E., Rossato, R., Schreiber, F.A., Tanca, L.: And what can context do for data? Commun. ACM **52**(11) (2009) 136–140
4. Cappiello, C., Matera, M., Picozzi, M.: A ui-centric approach for the end-user development of multidevice mashups. TWEB **9**(3) (2015) 11
5. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a better understanding of context and context-awareness. In Gellersen, H., ed.: Handheld and Ubiquitous Computing, First International Symposium, HUC'99, Karlsruhe, Germany, September 27-29, 1999, Proceedings. Volume 1707 of Lecture Notes in Computer Science., Springer (1999) 304–307
6. Bolchini, C., Curino, C., Quintarelli, E., Schreiber, F.A., Tanca, L.: A data-oriented survey of context models. SIGMOD Record **36**(4) (2007) 19–26
7. Daniel, F., Matera, M.: Mashups - Concepts, Models and Architectures. Data-Centric Systems and Applications. Springer (2014)
8. Chaisatien, P., Prutsachainimmit, K., Tokuda, T.: Mobile mashup generator system for cooperative applications of different mobile devices. In: Proc. of Web Engineering - 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011. Volume 6757 of LNCS., Springer (2011) 182–197
9. Cappiello, C., Matera, M., Picozzi, M., Caio, A., Guevara, M.T.: Mobimash: end user development for mobile mashups. In: Proceedings of the 21st World Wide Web Conference, WWW 2012, Lyon, France, April 16-20, 2012 (Companion Volume), ACM (2012) 473–474
10. Cappiello, C., Matera, M., Picozzi, M.: End-user development of mobile mashups. In Marcus, A., ed.: Design, User Experience, and Usability. Web, Mobile, and Product Design - Second International Conference, DUXU 2013, Held as Part of HCI International 2013, Las Vegas, NV, USA, July 21-26, 2013, Proceedings, Part IV. Volume 8015 of Lecture Notes in Computer Science., Springer (2013) 641–650
11. Lella, A., Lipsman, A., Martin, B.: The 2015 U.S. Mobile App Report. White Paper, ComScore, `http://www.comscore.com/Insights/Presentations-and-Whitepapers/2015/The-2015-US-Mobile-App-Report` (September 2015)
12. Schaller, R.: Mobile tourist guides: Bridging the gap between automation and users retaining control of their itineraries. In: Proceedings of the 5th Information Interaction in Context Symposium. IIiX '14, New York, NY, USA, ACM (2014) 320–323

13. Daniel, F., Matera, M.: Mashing up context-aware web applications: A component-based development approach. In Bailey, J., Maier, D., Schewe, K., Thalheim, B., Wang, X.S., eds.: Web Information Systems Engineering - WISE 2008, 9th International Conference, Auckland, New Zealand, September 1-3, 2008. Proceedings. Volume 5175 of Lecture Notes in Computer Science., Springer (2008) 250–263
14. Lee, E., Joo, H.J.: Developing lightweight context-aware service mashup applications. In: Advanced Communication Technology (ICACT), 2013 15th International Conference on. (Jan 2013) 1060–1064
15. Ardito, C., Costabile, M.F., Desolda, G., Lanzilotti, R., Matera, M., Piccinno, A., Picozzi, M.: User-driven visual composition of service-based interactive spaces. J. Vis. Lang. Comput. **25**(4) (2014) 278–296
16. : GraphQL. Draft RFC Specification, Facebook, `https://facebook.github.io/graphql` (2015)
17. : React Native. React Native official page, Facebook, `https://facebook.github.io/react-native` (2015)
18. Zobel, J., Dart, P.: Phonetic string matching: Lessons from information retrieval. In: Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR '96, New York, NY, USA, ACM (1996) 166–172
19. Sundarapandian, V.: Probability, statistics and queuing theory. PHI Learning, New Delhi (2009)