A Programming Language Approach for Context-Aware Mashups

Jorge Vallejos, Jianyi Huang, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt
Software Languages Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
{jvallejo,jihuang,pascal.costanza,wdmeuter,tjdhondt}@vub.ac.be

ABSTRACT

This paper focuses on mashup techniques as a solution for dynamic service composition in the Internet field. Existing mashups approaches provide simple means for efficient service interaction thanks to the use of asynchronous invocation schemes. However, the services are statically selected and wired to the mashups which significantly hinders the reusability. In addition, the asynchronous service invocations lead to fragmented mashup definitions obscuring the mashups' tasks. In this work we propose a programming language framework, called Dymac, that supports the development of context-aware mashups with the abilities of abstracting web services by type, performing dynamic service selection based on the context of use, and supporting sequential task-driven composition. With this framework web services become easier to maintain, web service selection is dynamic so that it is more adapted to the environment and the composition process of mashups is sequential and task explicit.

1. INTRODUCTION

The Internet is currently facing two evolutionary changes. In terms of software, the Internet has evolved from an information providing media to an interactive service platform, known as the Web 2.0. This platform enables the users to contribute with their own services to the Web, and more interestingly, to compose new services out of existing ones, called mashups. In terms of hardware, the computers connected to the Internet can now be integrated into everyday devices (such as mobile phones or home/office appliances), forming networks that dynamically change configuration as the users move about and the devices become (un)available. Software services in this setting are expected to react to such changes and adapt their behaviour accordingly. This dynamic condition entails a number of issues for today service composition techniques such as mashups, as they lack the appropriate support to deal with the effects of unreliable connections on the services. In these approaches, the unexpected disconnection of the devices hosting the services used in a mashup can lead to execution deadlocks, or in the best case, to exceptions. However, in the new hardware setting such disconnections can be just temporary, i.e. the devices can reconnect in a later point in time and resume the services [15]. Additionally, for the cases of permanent disconnections, the explicit reference to concrete services in mashup definitions prevent the disconnected services from being replaced with others providing the same functionality.

1.1 Event-driven Programming for Mashup Development

In this work, we investigate *event-driven* programming [14] as a programming model that can enable mashups to cope with the issues of dynamic networks described above. In event-driven programming, the interaction between the services, as well as the changes in their environment, are represented as events which are asynchronously exchanged and processed. Programs in this model are written in a reactive style, requiring the developers to perform some sort of continuation management to handle the events (e.g. callback functions, listener objects, etc.). An example of event-driven programming in the Web 2.0 are AJAX asynchronous requests [2][1]. Event-driven programming enables the mashup developers to manually deal with changes in the availability of the services at the program level. However, this model suffers from the problem of *inversion of control* [4]: the control flow of the programs is broken up according to non-blocking calls into several callbacks. Such fragmented control flow is particularly awkward for mashups as it obscures the tasks they accomplish.

To reconcile event-driven programs execution with the sequential task definition style required for mashups, we propose our ongoing work on an object-oriented programming language framework for dynamic mashups development, called Dymac. Dymac enables the mashups to become context-aware, i.e. to be able to dynamically adapt their tasks to the dynamics of the context of use, by promoting three design principles:

- To define mashups in terms of *service types* which group the services providing the same functionality, and *generic functions* that abstract away implementation details of the interaction with concrete services, making the task of the mashups more readable.

[1]Although their original raison d'être is rather related to increasing responsiveness.

- To dynamically select the services used in mashups based on context conditions such as service availability or user preferences.
- To feature a sequential programming style while internally executing the programs in event-driven manner. This allows structuring the mashups in a way does not require to explicitly deal with continuation-passing schemes (like callbacks).

We illustrate the use of Dymac by presenting the development of a simple yet representative mashup in the prototypical implementation of this framework in our Lisp-based research language platform, called Lambic [13]. At the end of this paper, we discuss how the properties of Dymac can be reused in other mashup development approaches.

2. A CONTEXT-AWARE MASHUP

Consider the scenario in which a user wants to reach certain destination and needs to know where to take the public transport. The mashup defined for this case, called `get-me-there` in this paper, requires the current user's location to pass it as parameter to the public transport service and get the location and timetables of the nearby stops, which are finally displayed in a map of a map service. Figure 1 shows an implementation of this mashup using Dymac (the used syntax is further explained in Section 4):

```
; GET-ME-THERE MASHUP
(defun get-me-there (destination)
 (let ((my-location (get-location location-service)))
  (if my-location
   (let ((stops (get-stops transport-service
                           from: my-location
                           to: destination)))
    (if stops
     (display-map map-service
                  center: my-location
                  markers: (list destination stops)))))))
```

Figure 1: Definition of the `get-me-there` mashup in Dymac.

In the above figure, the `get-me-there` mashup is represented as a function with the user's destination as argument. Its body mainly consists of the invocations of the `get-location`, `get-stops`, and `display-map` functions. The services indicated as parameters in these functions (`location-service`, `transport-service` and `map-service`) are not coupled to particular services but indicate the type of service that can handle such invocations. The parameters and return values of services of the same type are treated indistinctly. As such, the user's location can, for instance, be obtained from a GeoIP web service (e.g. MaxMind [9]), or from a GPS available in the user's environment. The result of the invocation to either of these services is wrapped in a location object which can be passed as parameter to the `get-stops` and `display-map` functions.

The concrete services used in the `get-me-there` mashup are dynamically selected at execution time according to the user preferences and the services' availability. For instance, in the case of the location service, the user may prefer the GPS over the GeoIP service as the former is more precise. However, if the GPS is not available, the GeoIP service is invoked instead.

Finally, each of the invocations to the services in the `get-me-there` mashup is asynchronously processed. Still, no callbacks are required to handle the results. This is implicitly done by Dymac enabling the mashups to be defined using a standard sequential programming style.

In the remainder of this paper, we explain how each of these features are supported by Dymac.

3. THE DYMAC FRAMEWORK

To support the development of context-aware mashups, Dymac promotes three design principles: type-based service composition, dynamic service selection and sequential event-driven execution.

3.1 Type-based Service Composition

In Dymac, services providing the same functionality are grouped and classified with the same type. A type is a first-class entity (e.g. an object) that can be directly used in mashups, acting as generic handler for the requests to the services the type represents. Using types, mashups are less vulnerable to changes in the services' availability. For this to work, all the services of a type, and the type itself, are provided with a similar interface. Then, the type and its services are structured in a way similar to the chain-of-responsibility pattern [1], where the type handles a request by successively delegating it to the services until the result is obtained. The parameters and return values of the functions are also abstracted away in types as a way to deal with the current diversity of data formats.

Mediating Functions

Type-based service composition also requires the definition of *mediating functions*[2] which take care of converting the generic requests and values into concrete invocations and data used by the services. With this, we introduce a stratification in the definition of mashups, where concrete service invocations and data are handled in a different level than the one of the composition.

3.2 Dynamic Service Selection

In Dymac, the concrete services used by a mashup are selected according to its execution context. Thus far, this context is represented by the availability of the services and the user preferences. While the former is associated to the changes in the connection of the services' hosting devices (as explained in Section 1), the latter determines the order in which the type's generic handler invokes the member services.

Dealing with Disconnections

In Dymac, there are two ways to deal with service disconnections – at the discovery and communication levels – which are inherited from the Lambic language [13] and originally proposed in the AmbientTalk language [15]. At the level of

[2]In a similar aim as the *data mediation* activity described by Maximillien et al in [8].

the discovery, Dymac provides a number of callbacks to specify actions upon the discovery, disconnection and reconnection of services. Such callbacks are used to keep the types' list of available services up to date (by adding or removing services).[3] At the level of the communication, Dymac relies on a time-based network failure handling mechanism. Invocations can be defined with a timeout that delimits the period of time to receive the response. This time interval is respected even if disconnections occur in between. A disconnection is considered permanent only after the timeout is reached, in which case an exception is raised. By default, Dymac handles timeout exceptions by delegating the invocation to next service in the list of a type.

User-driven Service Ordering

Dymac lets the mashup developers to determine the order of the services of a type by providing a priority algorithm for each function. By default, this algorithm is executed when services are added or removed, but also each time the algorithm is modified, enabling the priorities to be changed for particular invocations.

3.3 Sequential Event-driven Execution

Dymac features a sequential event-driven execution process based on the Lambic adaptation of the communicating event loops model for concurrency and distribution of AmbientTalk [15] and E [11]. We briefly highlight the features of this adaptation that are relevant for Dymac and refer the reader to dedicated references [13] for further details.

Asynchronous Remote Invocations and Return Values

In Dymac, a function invocation is synchronously processed only if it occurs within the device that owns the requested service. Inter-device computations are possible by means of asynchronous remote invocations. The return values of such invocations are handled by means of *futures* (also known as *promises* [7]). A future is an object created at the device from where the remote service is invoked, acting as placeholder for the result of the invocation. Once the return value is computed, it is communicated to the future; the future is then said to be resolved with the value. Originally this asynchronous communication scheme uses a reactive programming style, with explicit callbacks in the programs to express actions that depend on the results received by the futures. However, in Dymac we have extended this model to combine event-driven execution with a sequential programming style. For the sake of space and focus, in this paper we only explain the extension that concerns the invocation of web services.

The Sequential Event-driven Execution Process

The sequential event-driven execution process of Dymac relies on two principles: every invocation returns a future as result and every invocation can receive futures as parameters. The future returned by an invocation is implicitly handled which means that function definitions can be oblivious to it. The use of futures as parameters of an invocation causes the creation of an observer (also a future) that is notified when all the parameter futures are resolved with a value. Then, the futures are replaced with the values and the invocation is executed. Finally, the result of the execution resolves the future created for the invocation. No threads are blocked or created during this process, execution is entirely event-driven, where "events" are either incoming remote invocations or replies to earlier invocations containing the results for unresolved futures.

For web service invocations, we define a *send* operation that performs an AJAX asyncrhonous request and returns a future which is resolved with the result of the AJAX request. Therefore, web service invocations can be seamlessly integrated to the sequential event-driven execution process described above.

[3]This automatic mechanism does not prevent the developers or users from adding or removing services manually.

4. BUILDING CONTEXT-AWARE MASHUPS IN DYMAC

We now briefly discuss some details of the current implementation of the Dymac framework Lambic. Lambic is an extension to the Common Lisp Object System (CLOS). In CLOS, object-oriented programs are written in terms of function invocations rather than messages exchanged between objects (like in Java). Yet, both approaches result in the invocation of a method (or a chain of methods indicated in class inheritance relationships). For the sake of simplicity, in this paper we assume that the first argument of a function corresponds to the receiver of the invocation.

In this implementation, we use the Hunchentoot [16] Common Lisp web server as our mashup server. Thus far, Dymac supports the interaction with web services only using REST APIs. In this section, we illustrate the use of Dymac by building incrementally one of the functions used in `get-me-there` mashup introduced in Figure 1. In its current version, this mashup combines the services of the MaxMind [9] GeoIP service, a prototypical version of a public transport service developed at our lab, and Google Maps [3].

4.1 Building a Function

The first step a developer should do in order to add a new functionality to the Dymac framework is to define the a service type (if it is not already defined) and the corresponding function. For instance, the code below shows a simplified version of the implementation of the `get-location` function for the `location-service` type, used in the `get-me-there` mashup.

```
; LOCATION-SERVICE TYPE DEFINITION
(defclass location-service (service) ())

; GET-LOCATION FOR LOCATION-SERVICE TYPE
(defmethod get-location ((this location-service))
  (foreach (service (get-services this))
    (let ((result (try-catch (get-location service)
                    (timeout-exception () nil))))
      (if result
        (return result)))))
```

Figure 2: Definition of the `get-location` function for the `location-service` type.

This code shows the definition of the `location-service` type as a class (with the Dymac `service` class as superclass),

and the `get-location` function as a method specialised on the `location-service` class.[4] This method iterates over the services of the type invoking `get-location` on each service. This invocation is put inside a `try-catch` expression so that timeout exceptions thrown by the time-based network failure handling mechanism (described in Section 3.2) can be captured. In this example we assume a standard timeout stored in a global variable.

The structure of the body of the methods defined for service types is likely to be the same in all the cases (only varying in the method they invoke), and as such it can be automatically generated.

4.2 Defining the Proxy to the GeoIP service

Once the function is defined at the service type level, the next step is to add the concrete services and their mediating functions. In cases like web services, this also means to define proxies at the mashup server. The code below shows an implementation of the proxy to the GeoIP web service.

```
; PROXY TO GEOIP SERVICE
(defclass geoip-proxy (location-service)
  ((cached-locations initial-value: (make-hash-table)
                     getter: get-cached-locations)))

; ORIGINAL METHOD (mediating function)
(defmethod get-location ((this geoip-proxy))
  (send ... complete AJAX request ...))

; AROUND METHOD
(defmethod get-location :around ((this geoip-proxy))
  (let ((location (get (get-cached-locations this)
                       system-ip)))
    (if (not location)
      (begin
        (setq location (call-next-method)))
        (put (get-cached-locations this)
             system-ip location))))
    location))

; ADDING GEOIP PROXY TO SERVICE TYPE
(let ((geoip-service (make-instance 'geoip-proxy)))
  (add-service location-service geoip-service))
```

Figure 3: Definition the proxy for a GeoIP service.

In this implementation we enable the GeoIP proxy to cache geolocations to reduce the number of requests to the GeoIP web service. The service is accessed only if the there is no cached geolocation that corresponds to the requested IP. Note that in this case the location can either be obtained remotely (using an asynchronous AJAX request) or locally (using a standard synchronous access to the cache). While this can be particularly challenging for languages featuring different models for local and remote interactions, as it may lead to inconsistencies (e.g. if the wrong communication model is used), the implementation of this case in Dymac does not represent any special hazard. We include the caching functionality inside a (standard Common Lisp) *around* method for `get-location`. The `:around` annotation in the second definition ensures that this method is executed before the original `get-location` method presented in the previous example. In this definition, the geolocation is first looked up in the cache of the GeoIP proxy (represented by the hash table stored in the `cached-locations` field of the `geoip-proxy` class). Only if this location is not found, the original `get-location` method is invoked by means of the Common Lisp `call-next-method` function (which corresponds to a super call).

[4]In Dymac, as in Lisp, classes and methods are defined separately. A method can represent the behaviour of a class by indicating one of its arguments with the type of the class. In the method signature of the example, read `(this location-service)` as "let the `this` (receiver) argument be of class `location-service`".

5. DISCUSSION AND RELATED WORK

In this paper, we identify a number of issues for today mashup techniques caused by the new hardware phenomena. Although existing approaches present simple means to compose services, they lack the support to enable the mashups to properly react to changes in their environment (such as permanent or transient disconnections). We then introduce our ongoing work, called Dymac, which supports the development of context-aware mashups by providing type-based service composition, dynamic service selection and sequential event-driven execution. Currently, there are several programming language frameworks that acknowledge the effects of dynamic environments on mashups (like HOP [12], Orc [5], Flapjax [10] and Ubiquity[6]). However, none of these approaches feature the three properties proposed in Dymac. Still, a further study of the state of the art for dynamic mashup development is an important part of our present work.

Ubiquity[6] is a javascript-based add-on in Firefox that takes the browser as platform of dynamic mashups which are invoked by means of natural-language commands. Ubiquity lets the users dynamically determine the place (a web page) where the result of the mashup is presented. However, it still requires for the mashup definitions to make references to concrete services. HOP [12] is a programming language for building dynamic web services which also provides an abstraction layer for service invocation. Requests to web services are possible using the `with-hop` construct and the result is handled inside a lambda that acts as callback. Orc [5] is a programming language with explicit support for service orchestration. The mashup definition in this language is simple as the invocations are abstracted and represented by keywords, and service composition is specified with a small set of *combinators*. Flapjax [10] is a programming language that proposes a unified event-driven programming style for local and remote computations. Callbacks are avoided by adopting a functional reactive programming model based on data flows. This model also helps to achieve a separation of concerns as the one promoted by Dymac.

Acknowledgements

This work has been supported by the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB), and the MoVES project of the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy.

6. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable*

Object-Oriented Software. Addison-Wesley, 1995.

[2] J. J. Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005. [Online; Stand 18.03.2008].

[3] Google Inc. Google Maps. http://maps.google.com.

[4] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[5] D. Kitchin, A. Quark, W. Cook, and J. Misra. The Orc programming language. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal techniques for Distributed Systems; Proceedings of FMOODS/FORTE*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[6] M. Labs. Ubiquity. http://labs.mozilla.com/projects/ubiquity, 2009.

[7] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[8] E. M. Maximilien, A. Ranabahu, and K. Gomadam. An online platform for web apis and service mashups. *IEEE Internet Computing*, 12(5):32–43, 2008.

[9] MaxMind. MaxMind GeoIP. http://www.maxmind.com/app/ip-location, 2009.

[10] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. Technical report, Brown University, April 2009.

[11] M. Miller, D. E. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[12] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 975–985, New York, NY, USA, 2006. ACM.

[13] J. Vallejos, P. Costanza, T. Van Cutsem, and W. De Meuter. Reconciling Generic Functions with Actors. In *ACM SIGPLAN International Lisp Conference, Cambridge, Massachusetts*, 2009.

[14] T. Van Cutsem, S. Mostinckx, Elisa Gonzalez Boix, J. Dedecker, and W. De Meuter. AmbientTalk: Object-Oriented Event-driven Programming in Mobile Ad hoc Networks. In *XXVI International Conference of the Chilean Computer Science Society (SCCC)*. IEEE Computer Society, 2007.

[15] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, pages 3–12. IEEE Computer Society, 2007.

[16] E. Weitz. Hunchentoot Common Lisp web server, 2005 - 2009.