Lub: a DSL for Dynamic Context Oriented Programming

Steven Costiou Mickaël Kerboeuf Glenn Cavarlé Alain Plantec
Univ. Bretagne-Occidentale, UMR CNRS 6285, Lab-STICC, F-29200 Brest, France
{steven.costiou, mickael.kerboeuf, glenn.cavarle, alain.plantec}@univ-brest.fr

Abstract

Embedded, interactive or reactive systems have to face unexpected events coming from their environment. Taking this kind of event into account at design time raises the challenging issue of the dynamic behavior adaptation at runtime. In this paper, we investigate a DSL approach to address this problem. This DSL, called Lub, is a context oriented programming language. It is defined as a featherlight adaptation of Pharo which enables to temporarily change the base of the method lookup when a message is sent to an object. This language is evaluated with a running example of a fleet of drones facing an unexpected problem of GPS loss.

Keywords Behavior adaptation, Context Oriented Programming, Dynamic Lookup

1. Introduction

The need for dynamic adaptation especially arises in live systems where the events coming from the running environment cannot be entirely taken into account. A typical example would be an autonomous drone flying on a preprogrammed mission. Its software relies on physical sensors to perform its task, like plotting a route using a GPS signal to fly from one point to another. This behavior is defined at compile time and it is usually not designed to be modified at runtime to face unexpected events. For instance, if the drone loses its GPS signal and if it is not programmed to operate without it, the original behavior cannot be executed and the drone is unable to fulfill its mission.

In this paper, we investigate behavior adaptation at runtime by means of a context-oriented programming language. We explore through our DSL the notion of *context*. This notion comes from the Context Oriented Programming paradigm. In this approach, a context is traditionally defined with specific language constructs allowing software modifications or adaptations. By using contexts, the developer avoids the use of multiple *if* blocks for each specific behavior. Our drone would then have a default behavior while using its GPS, and when the GPS is lost, it enters a given context which allows it to continue its navigation without a GPS, for example by using a camera instead.

While the definition of *context* can be subject to subtlety, we propose to reify it as a pure language classifier, namely a class. When the need for a software to adapt to a situation arises, any object from the system that would need adaptation could change its context, *i.e.* change its class (and its behavior accordingly) while retaining its identity, *i.e.* its state and relations with other objects. Any other object instance of the same class would not be affected. To this end, we introduce the notion of *lookup base*, which enables the extension of the original lookup mechanism by specifying the base class from where to start the method lookup.

The main contributions of this paper are:

- The definition of Lub, a featherlight domain specific language for dynamic context oriented programming
- An evaluation of the first Lub implementation with a case study of a drone adaptation simulation

The remaining of this paper is organized as follows: in section 2 we describe our motivation through a simple example. In section 3 we describe the context oriented programming approach and we present the Lub language. In section 4 we show an evaluation of Lub in two use case scenarios. Related works are discussed in section 5 and to conclude in section 6 we present the current and future work.

2. Motivating Example: the fleet of drones and the GPS loss

This section describes a simple example which aims at underlying the need for dynamic adaptation.

In this example a fleet of drones is composed of two drones moving close to each other in a given environment. Since the environment is not entirely known, it is not possible to predict everything that could happen during the flight.

Each drone relies on its own GPS module to plot its route and fly without any incident. They also use proximity sensors to control their flight formation and maintain their relative positions to each other while avoiding potential obstacles. During the flight, one of the drones loses its GPS signal. Therefore, it has now no capability to guide itself as its sensors are sending back erratic or null data. Both drones enter a standby mode waiting for the situation to be resolved: this is the predefined behavior. If this behavior is required to be modified to enable the guidance by other means, the mission has to be aborted and the software has to be updated offline.

3. The Context Oriented Approach

This section describes a possible solution to the drone problem using context oriented programming. This simple solution raises questions from which we introduce our proposition.

3.1 Dynamic context adaptation

Let us say that the loss of one of the two GPS was in fact perfectly anticipated. The software has to adapt to the situation. It could be programmed on a case-by-case basis, *i.e.* with as many *if* blocks as expected specific situations we think the system would meet. But then, behavior would be spread into the source code instead of being factored in specific objects. In addition to the fact that it is not an object oriented approach, it would make code more difficult to read, to understand and to maintain (Costanza and Hirschfeld 2005).

Context Oriented Programming (COP) languages, as described by Salvaneschi et al. (2012), bring concepts and mechanisms for behavior adaptation. For example, the concept of *layer* has been introduced in COP, allowing the developer to define and dynamically choose behaviors depending on a context (Costanza and Hirschfeld 2005). It is the most common construct for context specific behavior in COP languages (Salvaneschi et al. 2012). Here, our GPS-less drone could activate a layer allowing him to communicate with its mate drone to rely on its functioning GPS. The following pseudo-code illustrates a typical solution with this approach:

```
class PositionTracker {
  ...
  layer GPSPositionTrackerLayer {
    // find position using the GPS sensor
    pinpointPosition() {...}
  }

  layer RemoteGPSPositionTrackerLayer {
    // contact mate drone and use its GPS
    // to pinpoint relative position
    pinpointPosition() {...}
  }
}

public class main (){
  ...
  with(GPSPositionLayer)
    positionTracker.pinpointPosition();
}
```

In this code, the *with* keyword activates a specific layer before executing a block of code. Our problem here is that the choice of the layer to use is tied to the loss of the GPS. It is not possible to use a specific layer, *i.e.* activate or deactivate it, from another thread than the base program (Appeltauer et al. 2011). Typically, we cannot trigger a layer activation or deactivation based on an external event (our GPS loss) without keeping track of the current layer in a global variable or as many as needed in a complex system (Kamina et al. 2011). To adapt our drone in our base program, we would need to use this workaround as follows:

```
...
onEvent(GPSON) {
  currentLayer:= GPSPositionTrackerLayer;
}

onEvent(GPSOFF) {
  currentLayer:= RemoteGPSPositionTrackerLayer;
}
...
public class main (){
  ...
  with(currentLayer)
    positionTracker.pinpointPosition();
}
```

Behavior adaptation is easily done, and our drone could continue to function properly without interruption. However, as noted by Kamina et al. (2011), in most COP languages layers can only be activated within a block and are automatically turned off at the end of this block. That means that context specific behavior cannot be active outside a scope bounded by the block of code. Each time the *pinpointPosition()* function needs to be called, a layer activation using the *with* keyword must be done to trigger the proper behavior. Furthermore, as layers are class based, layer activation cannot be specific to only one instance of a class. We would have two PositionTracker instances, one for determining the drone position and one for the other drone it is flying with, but we could not factor the code into the same layer activation. Subtle layer management would be needed to decide which layers each object should use or not:

```
public class main (){
  ...
  // Computing current drone position
  // depending of the current layer
  with(currentLayer)
    positionTracker.pinpointPosition();
  ...
  // Computing the mate drone position
  // to fly in close formation
  with(RemoteGPSPositionTrackerLayer)
    anOtherPositionTracker.pinpointPosition();
}
```

The DSL we investigate to address these problems aims at:

- Adapting object behavior on a per-instance basis, so that two instances of the same class can behave differently within the same block of code.
- Enabling behavior adaptation at runtime without thread or scope limitations, *i.e.* once a particular object has entered a specific context, it keeps its new behavior until

future context change. The context change may be triggered from an outer thread.

3.2 A dynamic DSL for Context Oriented Programming

In this work, we address the problem of behavioral adaptation at runtime by means of a DSL called Lub. This DSL is a featherlight adaptation of Pharo (Black et al. 2009), a Smalltalk dialect. Lub is a dynamically typed object oriented language: it extends Pharo with the ability to temporarily change the base of the lookup when a message is sent to an object.

3.2.1 Benefits of a DSL approach

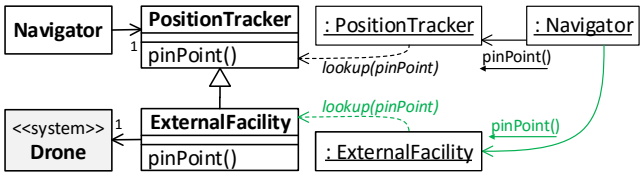

Figure 1. Updating the receiver to adapt message answers

This DSL-approach is a promising prospect since it can be efficiently implemented in an existing language, namely Smalltalk. As an illustration of this approach, we introduce in figure 1 an excerpt of the model used with the fleet of drones and the GPS loss case study. In this model, Navigator is in charge of the route. It relies on the pinPoint service provided by class PositionTracker, which is in charge of the GPS signal. At runtime, when an instance of Navigator sends a pinPoint message to its associated instance of PositionTracker, a lookup of method pinPoint is triggered from class PositionTracker.

Now we suppose that the GPS signal is lost. As a consequence, the instance of PositionTracker is not able to provide the expected pinPoint service. In this case, if another drone is nearby, it can provide its own pinPoint service through a dedicated point to point connection. In the model of figure 1, the other drone is depicted by class Drone. The services that this drone can provide through the dedicated connection are gathered in the facade class ExternalFacility.

At runtime, the receiver of the pinPoint message can be either an instance of PositionTracker or an instance of ExternalFacility. This ability to change the receiver of a same message relies on polymorphism and late binding. However, the responsibility of this change falls to the instance of Navigator. If the link between the two initial instances is required to remain unchanged, then the receiver of the message must have itself the ability to change *on demand* its answer to this message. This can be easily achieved by delegation, but this has to be provided by its interface. Moreover, this solution to the problem of GPS loss has to be foreseen, designed and implemented far before the execution.

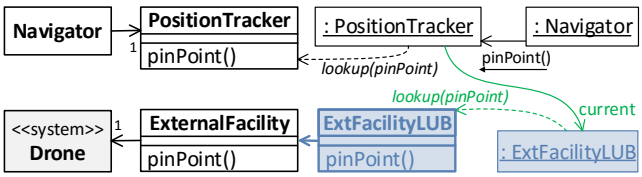

Figure 2. Update of message answer by Lub

Figure 2 depicts the specific adaptation abilities of Lub in this precise case. In this version, a *lookup base* named ExtFacilityLUB is associated with class ExternalFacility (which is no longer required to inherit from PositionTracker). Its instances can be *dynamically* associated with *any* object, without any constraint. They enable the lookup of method pinPoint from the objects they are associated with, even if they are not instances of PositionTracker or of one of its subclasses. As depicted by figure 2, the behavior of pinPoint depends on the presence of a link between the receiver of the message and a lookup base. As a consequence, the link between the instance of PositionTracker and the instance of Navigator remains unchanged whereas the behavior of pinPoint may vary. Moreover, this variation is not provided by the interface of PositionTracker and it does not rely on delegation. This variation is not even required to be designed before the execution. It can be implemented manually at runtime.

3.2.2 Language design

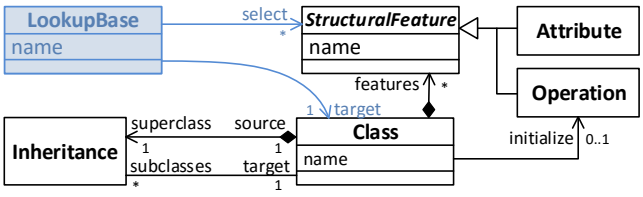

Figure 3. Lub Infrastructure

Figure 3 depicts the infrastructure of the DSL. A Lub specification is basically a set of named classes composed of named structural features, namely attributes and operations. A class can be derived from at most one class (simple inheritance). One of the operations associated with a class may play the role of *instance initializer*. So far, these language features are not original. The specifics of Lub lies on the lookup base depicted by class LookupBase in figure 3. It is associated with a target class and with a subset of its structural features. It is a named classifier and it can be instantiated.

The object runtime model of Lub is depicted by figure 4. An instance of a Lub class corresponds at runtime to an instance of class InstanceRecord. An instance of a Lub lookup base corresponds to an instance of class LUBRecord. In both

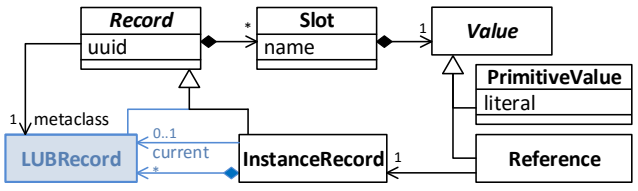

Figure 4. Lub Runtime Model

cases, they represent records that are referred by unique identifiers. These records contain named slots. A slot is a Lub class attribute, that can store either a primitive value or the reference of an instance. An instance can be associated with many lookup bases but only one can be activated at a time (the *current* one).

An instance record or a lub record is associated with one record which plays the role of meta-class instance. It corresponds to the starting point of a method lookup. More precisely, when an object receives a message, its lookup starts from the meta-class of the object itself if it is not associated with a lub record. This is the standard lookup mechanism. Otherwise, the lookup starts from the meta-class of the current lub record. If the lookup fails in the lookup base, it starts again in the original object meta-class which may trigger the method invocation or a *does-not-understand* exception. Notice that when an instance is associated with a lookup base, only this particular instance will perform its lookup in its lookup base. All other instances of the same class remain unaffected.

3.2.3 Language expressions

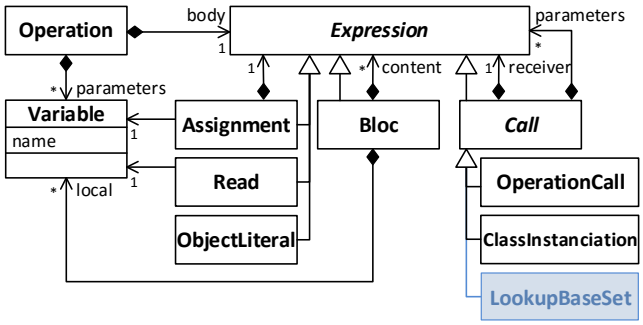

Figure 5. Lub Programming Constructs

Figure 5 depicts the body of Lub operations, which extends the Lub infrastructure depicted by figure 3. A Lub operation is made of one body and a sequence of formal parameters. A body is an expression corresponding to an assignment, a variable read-access, an object literal, a block of sub-expressions and a *call*. A call can be a method call or a class instantiation. Lub also enables the use of a specific operator specified by LookupBaseSet. Because of this operator, it is possible to add, change or remove the lookup base currently associated with an object as shown in figure 2.

3.2.4 First implementation

Our implementation includes a parser and a compiler, for which we rely on Pharo tooling (Bergel et al. 2013). The compiler is rather simple and is able to generate code for classes and methods defined with Lub.

Lookup mechanics is actually done using proxies and message interception with Ghost (Peck et al. 2015). We use what is called a virus in Ghost which *infects* a single object. Once infected, the object, if inspected, does not lose its identity. It is the same object with the same state from both the user and the Pharo system point of view. However, each message received by the object is intercepted by its virus where we can route the lookup as described earlier. The lookup base associated class is instantiated and when we need to perform the lookup in it instead of the receiver object, we forward the message to this instance.

4. Evaluation

In the following examples, we explore and evaluate Lub. We first go through the lookup mechanism and then we address our drone example with Lub. The Lub implementation with full examples and results are available in a Pharo one click image at `http://kloum.io/lub`.

4.1 Lookup through lookup base updates

In this example, we will show how the lookup is impacted by lookup base changes. Following a methodology inspired by Beugnard (2002), we declare three classes (X, J and K) with four methods (m1, m2, m3 and m4) using our DSL:

```
class J {
  attributes {}
  operations {
    m1
      ^'J.m1()'.

    m2
      ^'J.m2()'.}}


class K {
  superclass := J.
  attributes {}
  operations {
    m3
      ^'K.m3()'.}}


class X {
  attributes {}
  operations {
    m1
      ^'X.m1()'.
    m4
      ^'X.m4()'.}}
```

K inherits from J and X is apart. We now want to define two lookup bases, one associated with J and one associated with K. Each method prints in a *Java-like* way the name of the class where it has been found followed by its own name (method signature). For convenience purposes, when an object does not respond to a message, an *error* string is printed. Lookup bases are defined as followed:

```
LookupBase LUBJ {
  class := J.
}

LookupBase LUBK {
  class := K.
  without { m2. }
}
```

We can see that LUBJ does not restrict operation access to J and LUBK explicitly excludes m2 from K and its inheritance chain. In other words, an object using the context LUBK will not be allowed to access m2. Now, each of these methods will be successively called on an instance of x and the results will be logged. This will be our first test sequence. After this first sequence, a lookup base change will be triggered to use LUBJ, and the same test sequence will be executed. The same protocol will be repeated again for a change of lookup base to LUBK:

```
  "First test"
  x := X new.
  x m1; m2; m3; m4.

  "Second test"
  x lookupBase: LUBJ.
  x m1; m2; m3; m4.

  "Third test"
  x lookupBase: LUBK.
  x m1; m2; m3; m4.
```

A simple illustration of context change is depicted in figure 6 where we can see how the lookup base changes the lookup. Results are shown in table 1. We can see that for m1 and m3, the lookup behaves normally as if x was successively an instance of X, J and K. As method m2 is filtered in lookup base LUBK, the lookup behaves normally for x as X (*does-not-understand* exception) and for x as J. For x as K, x is not allowed to reach the method definition despite the fact that the method is compiled in K, so a *does-not-understand* exception is raised. The method m4 is a particular case, for which the lookup cannot be resolved in J and K, as m4 is not defined in any of these two classes. When the lookup fails in J and K, it starts again in the original class hierarchy of the object x and is able to find the method in X - therefore x is able to respond to m4.

call/context	x as X	x as J	x as K
m1()	X.m1()	J.m1()	J.m1()
m2()	ERROR	J.m2()	ERROR
m3()	ERROR	ERROR	K.m3()
m4()	X.m4()	X.m4()	X.m4()

Table 1. Lookup results through lookup base updates

So when the lookup base is unset for a given object, the lookup remains consistent with what we could expect from a smalltalk system. When the lookup base is changed, the lookup starts in the base class pointed by the lookup base specification. It can be caught and routed back to the original class of the object if it fails. However, once started (or rerouted) the standard system lookup is performed.

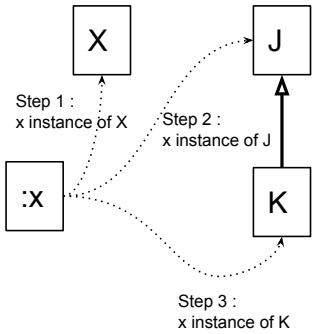

Figure 6. When the lookup base is changed, the object x behaves as an instance of another class.

4.2 GPS loss in a fleet of drones

We implemented with Lub a simple simulation of our fleet of two drones *dr1* and *dr2* from section 2. A simple event system allows external events to interact with the system, including the user. Drones log their actions into a transcript console. The original behavior of the drones is to reach a target on a grid using a GPS. They also have the capability to communicate between each other and to know the exact distance to each other. The simulated time is relative. At a given time in the simulation, one of the drones will lose its GPS and it will need to adapt.

We use two bases of code, the first one containing simulation classes with simulation utilities and simulated drone code. We focus here on the drone position tracking feature. The position tracker object is the object we are going to adapt. When the drone needs to compute its position, the interface *pinPoint: aDrone* of the tracker is always called:

```
positionTracker: anObject
  positionTracker := anObject

position
  ^self positionTracker
      ifNotNil: [:tracker |
        "Printing unique tracker id before pinpointing
        the drone position"
        Transcript show: 'Accessing␣'.
        Transcript show: tracker printString.
        Transcript space.
        tracker pinPoint: self ]
```

The second base of code is the Lub script, in which we are free to use any system class. We model two classes: *PositionTracker* which uses its drone GPS to find its coordinates, and *PeerPositionTracker* which uses its drone mate GPS to pinpoint its coordinates. *PositionTracker* is the default position tracker while *PeerPositionTracker* holds the behavior we will later need for adaptation. Each position tracker object is initialized with a unique ID: two different instances cannot have the same ID. A lookup base *PeerTrackerLookupBase* is declared, allowing any Lub object to use the class *PeerPositionTracker* as a lookup base. Note that

for the sake of readability, logging instructions have been removed from the following code:

```
class PositionTracker {
  attributes { id. }
  operations {
    pinPoint: drone
      |gps|
      gps := drone gps.
      ^gps ifNotNil:[gps positionFor: drone].

    printString
      ^'Tracker␣', self id printString.
  }
}

class PeerPositionTracker {
  attributes {  }
  operations {
    pinPoint: drone
      | peerDronePosition peerDroneDistance |

      "Asking peer drone GPS coordinates"
      peerDronePosition := drone peerDrone position.

      "Asking peer drone distance from
       the current drone"
      peerDroneDistance :=
        peerDronePosition distanceFrom: drone.

      "Computing the current drone coordinates"
      ^ peerDronePosition - peerDroneDistance.
  }
}

LookupBase PeerTrackerLookupBase {
  class:= PeerPositionTracker.
}
```

Two events are scheduled in the simulation, in the form of Smalltalk blocks. At relative time t=10 we simulate *dr2* GPS loss and at t=15 we update its lookup base with *PeerTrackerLookupBase*:

```
"Simulation events"
gpsDisablingEvent := [:simulation|
  (simulation agentNamed: 'dr2') disableGPS].
simulation addEvent: gpsDisablingEvent atTime: 10.

contextChangeEvent := [:simulation|
  |tracker|
  tracker:=
    (simulation agentNamed: 'dr2') positionTracker.
  tracker lookupBase: PeerTrackerLookupBase].
simulation addEvent: contextChangeEvent atTime: 15.
```

We can follow the code execution through the following log traces. The simulation starts and both drones use their default simulation tracker, which is an instance of *PositionTracker*. Each drone logs its simulation tracker access and its new position. Drones are moving towards an arbitrary target position fixed at point 50@50:

```
t = 6
Accessing Tracker 1
 [dr1: GPSMobileDrone] this is dr1 at (86@43)
Accessing Tracker 2
 [dr2: GPSMobileDrone] this is dr2 at (5@5)
t = 7
Accessing Tracker 1
 [dr1: GPSMobileDrone] this is dr1 at (85@44)
Accessing Tracker 2
 [dr2: GPSMobileDrone] this is dr2 at (6@6)
t = 8
Accessing Tracker 1
 [dr1: GPSMobileDrone] this is dr1 at (84@45)
Accessing Tracker 2
 [dr2: GPSMobileDrone] this is dr2 at (7@7)
```

At relative time t = 10, the second drone stops moving. It is not able to access its GPS anymore:

```
t = 10
Accessing Tracker 1
  [dr1: GPSMobileDrone] this is dr1 at (82@47)
Accessing Tracker 2
  [dr2: GPSMobileDrone] No pinpoint device available
t = 11
Accessing Tracker 1
  [dr1: GPSMobileDrone] this is dr1 at (81@48)
Accessing Tracker 2
  [dr2: GPSMobileDrone] No pinpoint device available
```

At t = 15, the lookup base update event occurs and changes the *Tracker 2* instance's lookup base. When drone *dr2* accesses its position tracker and calls its interface *pinPoint:*, the lookup is performed in the *PeerTrackerLookupBase* lookup base. The *pinPoint:* method of *PeerPositionTracker* is then called instead of the one from *PositionTracker* and the behavior of *dr2* is adapted accordingly:

```
t = 15
Tracker 2 updating lookup base with:
 Lookup Base for: PeerPositionTracker
Accessing Tracker 1
  [dr1: GPSMobileDrone] this is dr1 at (77@50)
Accessing Tracker 2
 (dr2 requesting dr1 position: Accessing Tracker 1 )
 [dr2: GPSMobileDrone] this is dr2 at (9@9)
t = 16
Accessing Tracker 1
  [dr1: GPSMobileDrone] this is dr1 at (76@50)'
Accessing Tracker 2
  (dr2 requesting dr1 position: Accessing Tracker 1 )
  [dr2: GPSMobileDrone] this is dr2 at (10@10)
t = 17
Accessing Tracker 1
  [dr1: GPSMobileDrone] this is dr1 at (75@50)
Accessing Tracker 2
  (dr2 requesting dr1 position: Accessing Tracker 1 )
  [dr2: GPSMobileDrone] this is dr2 at (11@11)
```

In addition to the behavior adaptation, we can see that state of the position trackers is still the same: trackers' unique IDs are unchanged. The position tracker instance has retained its identity despite the lookup base update, *i.e.* the object itself has not been exchanged with another object. Furthermore, only one of the *PositionTracker* instances has been impacted. We can see two distinct objects of the same class, *Tracker 1* and *Tracker 2*, behaving differently. Their unique IDs ensure they are separate instances and the log shows their difference of behavior.

Note that the *printString* method is not defined in *PeerPositionTracker*. When the *PeerPositionTracker* position message is sent to drone *dr2*, the position tracker is asked to print itself on the log output. The lookup is performed in *PeerTrackerLookupBase* but, as it does not find any *printString* method, it starts again in the original object class hierarchy (*PositionTracker*). Thus, the executed print code is the one defined in the original position tracker.

5. Related Work

In the following section we discuss other approaches and compare them to Lub.

5.1 Context oriented programming

COP languages have been studied and compared by Salvaneschi et al. (2012). We focus on the most common approaches and complete them with more recent ideas as a base of comparison with Lub.

5.1.1 Layer based languages

Many context oriented languages have been implemented over the years, in the form of language extensions, like, among others, ContextL for CLOS (Costanza and Hirschfeld 2005), ContextS for Smalltalk (Hirschfeld et al. 2008) or ContextJ for Java (Appeltauer et al. 2011). They address the particular need of behavior adaptation depending on contexts. Contexts are usually defined as *layers*, which can be (de)activated and composed at runtime using specific language constructs. Partial methods are implemented in layers and can be invoked within a specific layer activation scope (i.e. the context). The scope is typically a block of code and outside this block, the layer is never active. Layers cannot usually be activated on a per-instance basis. Furthermore, as this activation depends on the use of (nested) blocks of code confined in a specific thread, fine management of layer activation on external event initiated from other threads is difficult.

EventCJ (Kamina et al. 2011) is a DSL that brings a modular control of layer activation, making it possible to manage layer transitions based on events. Behavior adaptation is made on a per-instance basis. A layer transition can then be triggered from outside the base program thread and affect a single object. EventCJ is, in terms of objectives, the closest to our approach. However, in Lub, since the behavior adaptation activation strategy is left open, the developer can choose between *Lookup Base* updates based on a of mechanics event or a manually scoped activation (or both).

Von Löwis et al. (2007) propose two mechanisms that extend layers. They implemented implicit layer activation, allowing a layer to be declared as *active*. Layered method composition is made dynamically by checking all active layers upon a method invocation. Therefore, (de)activation of layers is made only when needed, and specific code to scope layer activation (*with* keyword) is no longer necessary. They also introduced dynamic variables to hold context-specific state. A dynamic variable is accessible from anywhere in a running program, especially from other threads. It makes possible for layer management to rely on external states and events. Efficiency of using such constructs has not been taken into consideration in their study.

L is an ongoing work (Hirschfeld et al. 2013a,b; Igarashi and Felgentreff 2015) relying only on layers instead of a combination of classes and layers. Behavior and state are defined and can be shared in layers, with an access control to use them through layer composition. This access control brings verbosity to the code, and the authors are still working on a simpler syntax. Behavior adaptation is made by activating (nested) layers like in other layer-based languages. As there is no more classes, lookup is exclusively done through layers, taking into account restriction access to methods (via a *Smalltalk-inspired* lookup). Layers activation is still bounded to an execution scope and adaptation cannot simply be performed on a single object.

As COP languages are based on layer activation and deactivation throughout the execution of the program, the impact of repeatedly switching layers on and off on performances may become a concern. Costanza et al. (2006), while raising this question, show that it is possible to write an efficient program with layer constructs.

5.1.2 Role based languages

The Epsilon model (Tamai et al. 2005) proposes to use *roles* as a mechanism for object to adapt their behavior to their environment, which they call *Context*. The EpsilonJ language, based on Epsilon model, allows the definition of a Role which implements context-specific behavior. A role instance can bind to exactly one object, which in return can be bound to multiple roles instances, assuming role composition.

NextEJ (Kamina and Tamai 2009) brings type safety and role scoping to the Epsilon model. As for layers, roles in NextEJ are only active within a scope. Furthermore, roles states are preserved upon deactivation, allowing its bound object to reactivate it later and recover its context related state. Both NextEJ and EpsilonJ provide behavior adaptation on a per-instance basis, but it is not obvious how to simply manage role (de)activation from outer threads events.

5.2 Other work

5.2.1 Talents

Talents (Ressia et al. 2014) are objects, called *units of reuse*. A talent models behavior that a specific object can acquire or lose dynamically at runtime. To extend an object's features, one must instantiate a new Talent object, model the features (e.g. define new methods) and ask the object to acquire the talent object. The lookup is preferably performed in the talent object instead of in the object that acquired the talent. For example, if an invoked method is both defined in the talent and in the object using the talent, the talent method will always be called instead of the original object's method. This allows a per-instance basis behavior adaptation and no conflict resolution is needed. Talents are also composable, and cannot be composed if any conflict between the talents exists. These conflicts have to be resolved manually.

Talents feature a very effective solution, allowing dynamic behavior adaptation at runtime on a per-instance ba-

sis. However, there is no abstraction nor control structure to properly handle a COP mechanism, which we want to provide with our lookup bases. Also, composition conflicts and behavior removal in Talents can be a source of difficulties when trying to assess if an adaptation can cause failures.

Conceptually, Lub could rely on Talents to provide the lookup base mechanisms. Although Lub has been implemented with other means, a comparison with a new implementation based on Talents could be interesting.

5.2.2 Reclassification

$Fickle_{II}$ (Drossopoulou et al. 2002) is a strongly typed object oriented language providing a reclassification operation: an object can change its class membership at runtime. $Fickle_{II}$ ensures that an object subject to reclassification will never attempt to access non existing members, thus avoiding *does-not-understand* exceptions. However, an object of a given class can be reclassified to a target class only if both classes share a common *root* superclass. A *root* class is a specific kind of class introduced in $Fickle_{II}$, therefore an object cannot be reclassified to a class outside the scope of its *root* class hierarchy. Lub allows an object to change its lookup base to any class of the system, but it is then possible for a message to not be understood by its receiver.

5.2.3 Aspect oriented approaches

Tanter et al. (2006) propose an aspect oriented system with context-specific behavior management, in the form of an open framework. While putting aside performance and efficiency concerns, they describe and analyze the needs and implications of *context-awareness* of aspects. As they state, aspects and COP do not follow the same paradigms and as such are not easy to compare. However, we find some of their implemented features interesting for behavior adaptation, such as keeping track of each defined context and their state.

5.2.4 Framework based approaches

Context aware systems (Baldauf et al. 2007) aim at adapting their behavior to their current context. These systems are implemented as frameworks, with tools and user level services. Contexts are usually reified as real world entities, *i.e.* as models. Other frameworks, like Fractal (Mantilla 2011), a component based architecture, allow dynamic software evolution at execution time.

These works target software adaptation and/or evolution at an architecture level. Frameworks provide system entities that allow the software to perform its adaptation to a context. With Lub, we propose a meta-level approach, where the adaptation is part of the language semantics.

6. Conclusion and future work

We have introduced Lub, a new DSL for dynamic context oriented programming. We described the *Lookup Base*, the new abstraction the language brings to COP and illustrated how it impacts the lookup mechanics in Lub.

For now, all the features we described have not been implemented yet. We are currently working on this implementation and on more detailed examples. Lub does not implement safety checks yet and we deliberately put performances concerns aside. The next step after a more complete implementation would be to compare the language to other approaches in terms of safety and efficiency, for example a Lub solution based on Talents. We also think of going further in the Lub's integration into Pharo, in the form of an object interface instead of an internal compiled language.

For example, we will investigate message interception at the meta-level instead of using a *virus-proxy*. We will also investigate a real lookup instead of forwarding the message we want to re-route.

We also plan to experiment on a physical target device. We plan to use an embedded version of Pharo to evaluate Lub on a *home made* robot built around an electronic card. This would allow us to study a Lub program evolving through a real case of behavior adaptation instead of a pure simulated example.

References

M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, 6(2):399–419, 2011. doi: 10.11185/imt.6.399.

M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo*. Lulu. com, 2013.

A. Beugnard. Oo languages late-binding signature. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*. Citeseer, 2002.

A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: An overview of contextl. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM. doi: 10.1145/1146841.1146842. URL `http://doi.acm.org/10.1145/1146841.1146842`.

P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient layer activation for switching context-dependent behavior. In *Modular Programming Languages*, pages 84–103. Springer, 2006.

S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle &par. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

R. Hirschfeld, P. Costanza, and M. Haupt. An introduction to context-oriented programming with contexts. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008.

R. Hirschfeld, H. Masuhara, and A. Igarashi. L: Context-oriented programming with only layers. In *Proceedings of the 5th International Workshop on Context-Oriented Programming*, page 4. ACM, 2013a.

R. Hirschfeld, H. Masuhara, and A. Igarashi. Layer and object refinement for context-oriented programming in l. In *Proceedings of 95th IPSJ Workshop on Programming, IPSJ*, 2013b.

R. H. H. M. A. Igarashi and T. Felgentreff. Visibility of context-oriented behavior and state in l. , 32(3):149–158, 2015.

T. Kamina and T. Tamai. Towards safe and flexible object adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM, 2009.

T. Kamina, T. Aotani, and H. Masuhara. Eventcj: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0605-8. doi: 10.1145/1960275.1960305. URL `http://doi.acm.org/10.1145/1960275.1960305`.

J. N. Mantilla. *Une infrastructure pour l'optimisation de systèmes embarqués évoluables à base de composants logiciels*. PhD thesis, Univ. Bretagne-Occidentale, 2011.

M. M. Peck, N. Bouraqadi, L. Fabresse, M. Denker, and C. Teruel. Ghost: A uniform and general-purpose proxy implementation. *Science of Computer Programming*, 98:339–359, 2015.

J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: an environment for dynamically composing units of reuse. *Software: Practice and Experience*, 44(4):413–432, 2014.

G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

T. Tamai, N. Ubayashi, and R. Ichiyama. An adaptive object model with dynamic role binding. In *Proceedings of the 27th international conference on Software engineering*, pages 166–175. ACM, 2005.

É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *Software Composition*, pages 227–242. Springer, 2006.

M. Von Löwis, M. Denker, and O. Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.