Science of Computer Programming 161 (2018) 149–171

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Lub: A pattern for fine grained behavior adaptation at runtime

Steven Costiou *, Mickaël Kerboeuf, Glenn Cavarlé, Alain Plantec

Univ. Bretagne-Occidentale, UMR CNRS 6285, Lab-STICC, F-29200 Brest, France

ARTICLE INFO

Article history:
Received 14 January 2017
Received in revised form 19 June 2017
Accepted 19 September 2017
Available online 27 September 2017

Keywords:
Unanticipated adaptation
Dynamic behavior
Dynamic lookup
Runtime evolution

ABSTRACT

Autonomous systems have to evolve in complex environments and their software must adapt to various situations. Although it is common to anticipate adaptations at design time, it becomes a more complex issue when facing unpredictable contexts at runtime, especially if applications cannot be stopped. We introduce Lub, a pattern designed to extend object oriented languages with fine grained unanticipated adaptations. Lub is based on dynamic instrumentation of the lookup, and allows objects to acquire behaviors from another class than their own. A Pharo Smalltalk implementation of Lub is evaluated through a performance analysis and a running example of a fleet of drones facing unexpected GPS problems. Lub is then discussed from the unanticipated software adaptation perspective.

1. Introduction

The need for dynamic behavior adaptation especially arises in applications that need to run continuously, or embedded in autonomous systems like drones or robots. Adaptations may not be easy to predict, for example because of unexpected events coming from a complex environment in which the system is running, or more simply because of bugs. Sometimes these systems cannot be stopped, or the cost of turning them off for an update or for debugging is too high.

An example could be an autonomous fleet of drones flying on a programmed mission. Their software relies on physical sensors, *e.g.* a GPS, to recover and share the fleet's position. This behavior is defined at compile time and it is usually not designed to be modified at runtime. For instance, if one of the drones loses its ability to process the GPS signal and if it is not programmed to operate without it, the whole fleet will be affected and will be unable to fulfill its mission. Debugging the system would be a very interesting option as recalling the drones means canceling the mission. First we would like to see what is happening in the software. A very basic debug technique is to trace on a log what the system does, if this option is available in the running system. If not, this logging behavior has to be defined, added to the running system (without interruption) and removed once it is not needed anymore. Second, once we figured out what is wrong we would like to adapt the behavior of the software, and *try–cancel–retry* new adaptations through the debugging process. These kinds of changes cannot be anticipated.

Unanticipated software evolution [1] makes possible to dynamically adapt a running software. Applications can be instrumented by touching both functional and non-functional behaviors. This opens perspectives in the debugging activity to understand what is happening in the application and to experiment behavioral variations before applying a patch. Al-

* Corresponding author.
E-mail addresses: steven.costiou@univ-brest.fr (S. Costiou), mickael.kerboeuf@univ-brest.fr (M. Kerboeuf), glenn.cavarle@univ-brest.fr (G. Cavarlé), alain.plantec@univ-brest.fr (A. Plantec).

https://doi.org/10.1016/j.scico.2017.09.006
0167-6423/

though there exist adaptive systems, not many of them cope easily with unanticipated adaptation [2]. A key component in these systems is the adaptation mechanism, which must provide enough flexibility and precision to address the problems of unanticipated behavior adaptation [3].

In this paper, we explore and discuss a pattern named *Lub*. We use it to build an adaptation mechanism designed to answer the needs and constraints of unanticipated behavior adaptation in object-oriented software. With Lub, the developer can specify adaptations to extend or modify the protocol of a given object. Adaptations target classes so that any object can individually acquire behaviors from another class than its own. Lub is designed to be a pattern, therefore it can be implemented in any technology provided the host language has the necessary capabilities discussed in this paper.

The main contributions of this paper are:

- The definition of Lub, an object oriented pattern for unanticipated behavior adaptation. The pattern is described in detail, and we specify how it could extend an object-oriented language.
- A performance analysis of a featherlight implementation of Lub with the Pharo language, to analyze its usability in terms of execution speed, adaptation time and memory overhead.
- The evaluation of this implementation through the case study of a drone simulation. The evaluation shows how adaptations can be specified and how they impact the behavior of the running software.
- The discussion of the Lub pattern with regard to the literature.

The remainder of this paper is organized as follows. In section 2 we describe our motivation for unanticipated adaptation, which we illustrate through a simple example with drones. In section 3, we discuss the difficulties of unanticipated dynamic adaptation. Section 4 defines the Lub pattern and section 5 briefly describes the current implementation. Section 6 shows a performance analysis of Lub and section 7 shows an evaluation of Lub in two use case scenarios. Lub and related works are discussed in section 8. Future works are described in section 9 and we conclude this paper in section 10.

2. Motivating example: unanticipated behavior adaptation in a fleet of drones

This section illustrates the need for unanticipated dynamic adaptation through a simple example. Two drones are moving together, close to each other. The first one (*gps-drone*) has an active GPS while the other one (*follower-drone*) uses a communication channel with *gps-drone* to recover its own relative position. The shared GPS of *gps-drone* is used for navigation by the fleet.

2.1. The GPS loss scenario

Since the environment is highly dynamic and therefore not entirely known, it is not possible to predict everything that could happen during the flight. Our scenario is illustrated in Fig. 1. The fleet enters a secured area where all communications must be encrypted. However both drones enter a stand-by mode because *follower-drone* fails to recover its GPS position from the first drone. But *gps-drone* GPS' signal is fine and the communication between the two drones seems alright. The context change was effective in *gps-drone* but not in *follower-drone*, therefore when *follower-drone* asks *gps-drone* its GPS position, *gps-drone* answers encrypted data that *follower-drone* is unable to decipher. The drones are waiting for the situation to be solved: this is a predefined behavior. To change this behavior and to enable adaptation for this particular context, a simple way is to abort the mission and to update the software offline.

This example may happen in a simulation when designing the software, as the use-case we will show in section 7.2. It can also occur in a real mission with autonomous drones. In both cases, restarting the mission can be very expensive: a simulation may have run for a long time, autonomous drones may have flight over a great distance. Stopping a simulation or recalling the drones each time there is a new context that was not anticipated at design time is a problem. In addition, the drones accumulated dynamic states and data until the problem happened: they *lived* in an unpredictable and complex environment. So there are no guarantees one could reproduce the problem in development mode due to the impossibility to recreate or simulate the exact context and conditions under which it originally happened. This use-case can be classified as a bug and as such it remains unforeseen at design time.

2.2. The unanticipated adaptations

The fleet is stalled because one of the drones failed to adapt to the changing conditions (the entering of the unsecured area). We know exactly what happens here, when we describe our use-case in this paper, but it is not obvious that a human operator would easily understand the problem when it happens. From his point of view, he is facing an unexpected problem with no obvious reason, and a restricted set of options. Either the drones are recalled and their software and data analyzed offline, either the operator has to dynamically update their behavior. The software itself cannot adapt, as we made the assumption in section 2.1 that this particular case was unforeseen at design time.

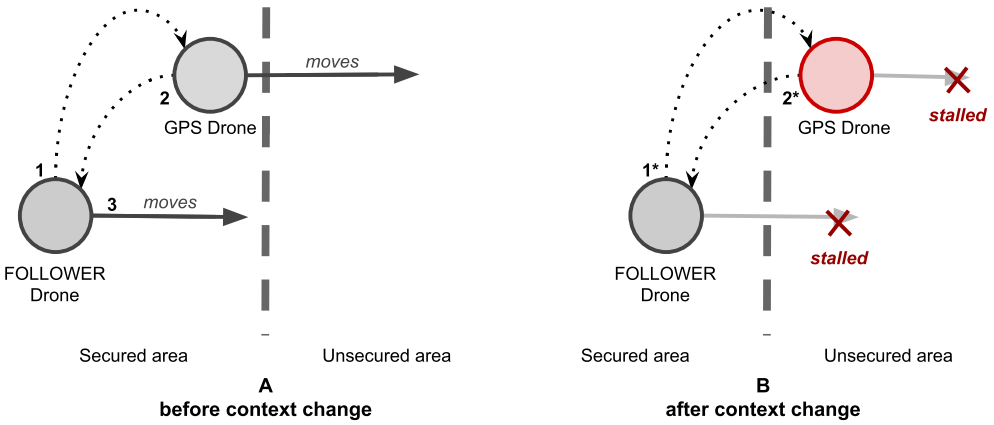

Fig. 1. The fleet is moving in a secured area (A). *Follower Drone* requests GPS position (1) from *GPS Drones* which answers (2) and the fleet can move (3). When *GPS Drone* enters the unsecured area (B), it starts ciphering its communication. When *Follower Drone* sends requests (1*), *GPS Drone* answers with encrypted data (2*). *Follower Drone* was not aware of the context change and cannot interpret the ciphered data. The fleet is stalled.

A self-adaptive system [4,5] would allow such program to correct itself, but it would unlikely be able to fix every kind of bug. For example a user intervention would be required to change the semantics of a given piece of code, or to interpret data and find the cause of the erratic behavior. Dynamically updating the software (DSU) is then a possibility [6,7]. DSU can be very intrusive, as it can deeply modify an application base code (from removing states to completely change a class hierarchy). As such it doesn't allow easy experimentation of new behaviors. Previous states could be lost because the application has migrated, unless there is dedicated management to ease reverting to a previous version but also to previous states.

Instead, the human operator could prefer to experiment changes: probe the system, find the problem, design and implement a solution. These steps could be reproduced until a reasonable solution is found by the user, which would lead to a context adaptation. These changes would then be dynamically integrated to the runtime. This is a very similar use-case as in [8], and almost identical regarding the adaptation methodology (*i.e.* probing, understanding, adapting).

Keeney [9] defines *completely unanticipated adaptation* as a dynamic adaptation for which every aspect (what it does, where, when and how it is applied) remain unknown until runtime. He argues that this kind of adaptation is needed when one does not know the system will need adaptation before runtime and when stopping-updating-restarting the system is not acceptable, typically when dynamic state has been acquired and should not be lost during the adaptation process. We believe the adaptations in our use-case to be completely unanticipated, first because probing the system by adding behavior is not part of the original design. As a means to temporarily observe the running software and figure out what is wrong with it, this behavior should also be removed after the adaptation is done. Second, because probing the system to build a consistent adaptation means that the needed behavior is clearly unknown until the log is analyzed and the problem understood.

For example, the operator could decide to add specific logging behavior to the entity receiving the GPS data in *follower-drone* to understand what is wrong. The log function is arbitrary: at some point we need to understand the whole context, therefore it is necessary to build custom views of the system's internals. Then the operator would define and apply a new behavior consistent with the current context and validate the adaptation using the view he previously built to inspect the system. Once the adaptation is validated, the probing behavior can be removed from the system because it is no longer needed. Then the drones can resume their mission. These two steps, namely investigating and fixing the running system, are complementary and can be done concurrently while debugging the software. Ultimately, adaptations designed to inspect the system are removed while the behavior fixes will remain, thus solving the problem.

3. Difficulties and requirements for unanticipated adaptation

In this section, we put the emphasis on specific difficulties of dynamic unanticipated adaptation. Adapting running software with unexpected behaviors leads to many problems, which have been described by Ebraert et al. [3]. From these difficulties we define constraints for an unanticipated adaptation mechanism.

3.1. Fine grained adaptations

Adaptations of objects should be very accurate, for example when building a specific view of a running system. If we add logging behavior, we may want to adapt only a given object so that all objects from the same class would not be affected. That would allow in this case to build a meaningful view and not having a lot of noise from unrelated objects when logging. This marks a difference in the needs with [8]: adaptations may target very specific objects and not all instances from the same class.

3.2. Preservation of adapted objects

It is difficult, yet imperative to understand the impact of an adaptation in terms of states and of dependencies. The grain of adaptations should be as fine as possible, to minimize and to better control the effects of adapted behaviors. States consistency is also a complicated problem, as it has to remain stable between two adaptations. For example, while adapting an object, we must ensure that its states have been preserved from one version to another because these states have been accumulated throughout the execution to form the current context. It is also important to preserve the original behavior in our drone example if the adaptations are unsuccessful. We must have the possibility to revert to the original behavior (*i.e.* the stand-by mode) and to recall the drones (and not definitely lose them). So while being adapted, an object must preserve its identity: structural connections, references and states. It avoids overly complex management, because structural reconfiguration implies maintaining coherent states and relations between objects.

3.3. Flexibility of the adaptation design

In [2], Da et al. describe middleware components in adaptive systems. Each middleware brings specifics features required for adaptation, *e.g.* the adaptation trigger system. However as adaptations are unanticipated, the adaptation strategy (*i.e.* how and when adaptations are triggered) possibly remains unknown until it is needed [9]. The choice of this adaptation strategy must then remain free and flexible. A way to select which behavior to adapt, and to express this selection, is also a concern, as we must be able to adapt part of an object's behavior and not necessarily its whole interface. While the adapted entities would be objects, the grain of adaptation would in fact be at the method level.

3.4. Modularity support

A module can be defined as a piece of an individual program which contains everything it needs to perform what it is designed to do, without requiring structural constraints from the software in which it is embedded. In our case, it is the ability for the adaptation mechanism to be loaded at runtime as an independent entity which does not require anything from the software's design to be plugged-in. To be completely unanticipated, one must be able to load the adaptation mechanism during runtime. It is possible that the base program does not carry the mechanism as a native capability. In that case to perform unanticipated adaptation one must inject this capability at runtime. Although it implies that the possibility of loading new libraries must be anticipated, the adaptation mechanism itself must be (un)loadable at any time while the program is running.

3.5. Reflection support

Ebraert et al. [3] analyze that further support for reflection is needed to address these problems and advise to use dynamic languages like Smalltalk [10]. Da et al. [2] also emphasize the need for reflection, as they describe systems where a single middleware is in charge of providing reflection mechanisms. Having highly reflective capabilities is a mandatory requirement to address unanticipated adaptation from a language perspective.

3.6. Requirements for unanticipated adaptation

We introduced the problem of dynamic unanticipated adaptation through our use-case. The fleet of drones needs unexpected adaptations because it meets new situations for which it's software was not designed. Adapting the system, especially if it was not foreseen, is an interesting option against canceling the whole mission. The adaptation mechanism at the language level is a key functionality to achieve such adaptation, but subject to various known difficulties.

In the scope of this paper we propose to investigate the adaptation mechanism, that should meet specific requirements to overcome difficulties of dynamic unanticipated adaptation:

- **Granularity** It is the smallest language construct or entity to which an adaptation can be applied (*e.g.* an instance, a set of instances or a class).
- **Rollback** The ability for an adaptation to be canceled, and to revert to the original behavior of the adapted entity.
- **Dynamicity** The ability for an adaptation to be designed and applied at runtime.
- **Identity Preservation** The ability for an adaptation to preserve the identity of the adapted entities.
- **Flexibility** The adaptation capability doesn't put constraints on the software's design, *e.g.* glue code, extensive use of non-functional code or the use of a particular framework.

In addition, we also need the mechanism to be practical. A user should be able to easily define and apply adaptations at runtime while still having full benefits of his tools and developing environment.

- **Modularity** The ability for the adaptation mechanism to be loaded at runtime as an independent entity.
- **Offline composability** The ability for multiple adaptations targeting the same entity to be combined offline to produce a unique runtime adaptation.

- **Non-intrusive adaptations** Adapting runtime entities do not introduce adaptation-related code or instrumentations into the base code of the software. More specifically it does not break nor pollute tools (*e.g.* debuggers).
- **No paradigm shifts** Using the adaptation mechanism must not induce a change of paradigm (*e.g.* object oriented), so that the developer does not have to switch between different programming models (*e.g.* one for the base application and one for the adaptations).

4. Lub: a simple pattern for dynamic adaptation in object-oriented languages

To cope with unanticipated adaptation of programs at runtime, we investigate a solution which implies a minimal adaptation of programming languages and environments. More precisely, we do not aim at defining a new programming paradigm, or even a new programming language. We focus instead on the principles of a featherlight adaptation of existing programming languages.

For that purpose, we defined Lub. It stands for a minimal object-oriented language that is typically used to define the kind of programs that we aim at adapting. Lub introduces a specific construct for dynamic adaptation. Besides this construct, Lub only gathers the elements that are required to enable dynamic adaptation. Hence, Lub is not statically typed and it supports only simple inheritance.

Some languages like Smalltalk [11] or Python [12] meet these minimal requirements. As Lub is intended to be an abstraction of these kind of programming languages, we present Lub as a *language pattern* for dynamic adaptation in object-oriented languages. It is not supposed to be a fully operational language.

This pattern-based approach is a promising prospect since it can be implemented in any existing language as far as it meets the minimal requirements exposed below. It mainly relies on well-known basic object-oriented features and its specific construct for dynamic adaptation relies on a new simple infrastructure entity and a new statement.

In this section, we detail the infrastructure of Lub, its statements and its operational semantics. The benefits of this approach are illustrated by the GPS loss case study.

4.1. Benefits of Lub

Fig. 2 is an excerpt of the model used with the fleet of drones and the GPS loss case study. In this model, ComPort is in charge of the communication between the drone and either the operator or the other drone. In both cases, a received message can be read by method readMessage. ComPortLog introduces a specialized version of readMessage which consists in keeping the communication traces in a dedicated log file. Thus, at runtime, when an instance of Drone sends readMessage to one of its associated instances, a lookup of method readMessage is triggered either from class ComPort or from class ComPortLog.

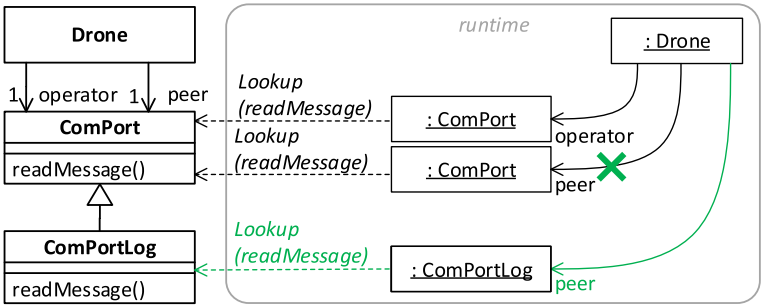

Fig. 2. Updating the receiver to adapt message answers.

Now, we suppose that an instance of Drone is associated with instances of ComPort at the beginning of the execution. Then, later, there is a need for logging the communications between this drone and the other one. This requirement can be easily met by means of polymorphism and late binding. As depicted by Fig. 2, the instance of ComPort playing the role of peer can be replaced by an instance of ComPortLog so that the following lookup of readMessage triggers the logging of communications.

However, this modification has to be anticipated. Moreover, the responsibility of this modification falls to the instance of Drone whereas it is related to a method of ComPort. This may be a problem if the link between the two initial instances is required to remain unchanged. Fortunately, the receiver of readMessage can be designed to be able to change *on demand* its answer to this message. This can be easily achieved by delegation, but this feature has to be provided by its interface. In other words, this adaptation has to be foreseen, designed and implemented far before the execution.

Fig. 3 depicts the specific adaptation abilities of Lub in this precise case. In this version, the initial model of Fig. 2 has been extended with AdaptComPort, an instance of a specific Lub construct named *Adaptation*. This construct is a classifier associated with an existing class. It specifies a subset of attributes and/or methods from the associated class that have to be

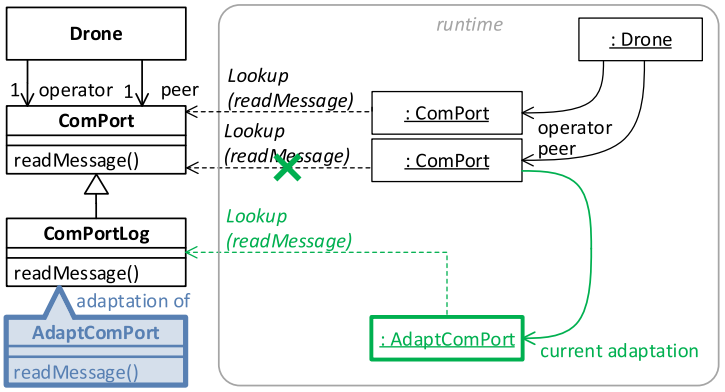

Fig. 3. Update of message answer by Lub.

adapted. In this example, the whole interface of ComPortLog, namely method readMessage, is adapted. At runtime, instead of modifying the link between the instance of Drone and the instance of ComPortLog playing the role of peer, a new instance of AdaptComPort is associated with this instance of ComPortLog. Because of this association, a call of readMessage triggers a lookup from the class associated with AdaptComPort, namely class ComPortLog.

As a consequence, the link between the instance of Drone and the instance of ComPort remains unchanged whereas the behavior of readMessage can vary. Moreover, this variation is not provided by the interface of ComPort and it does not rely on delegation. This variation is not even required to be designed before the execution: it can be implemented and triggered manually at runtime.

4.2. Lub infrastructure

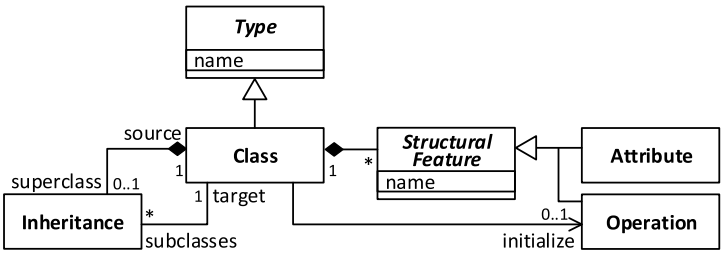

Fig. 4. Lub infrastructure pattern.

Fig. 4 depicts the infrastructure of the Lub pattern. It relies on the main notion of *class* which is basically the only required *named type* in the language. A Lub class is composed of several 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*.

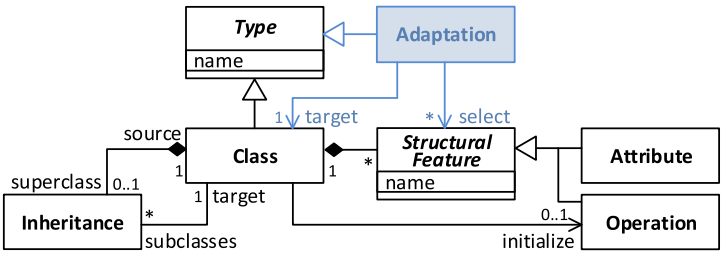

Fig. 5. Lub infrastructure extended with adaptation.

So far, these language features are not original. The specifics of Lub relies on the notion of *adaptation* depicted by class Adaptation in Fig. 5. It is associated with a target class and it selects a subset of its structural features. It is a named classifier and it can be instantiated. An adaptation can be associated with any object *o*. It specifies for *o* an alternate starting point for a method lookup or for an attribute access.

4.3. Lub operations

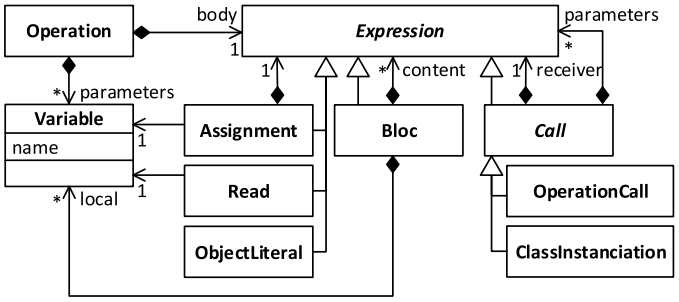

Fig. 6. Lub programming constructs.

Fig. 6 completes Fig. 4 with the details of Lub operations. 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.

These well-known notions are extended with a specific Lub operator depicted by AdaptWith in Fig. 7. As a call expression, this operator is associated with a receiver and a parameter. The parameter is an expression whose evaluation at runtime must provide an instance of class Adaptation. The receiver can be any object whose behavior is intended to be adapted. Using this operator, it is possible to add, change or remove the starting point of a method lookup as shown in Fig. 3.

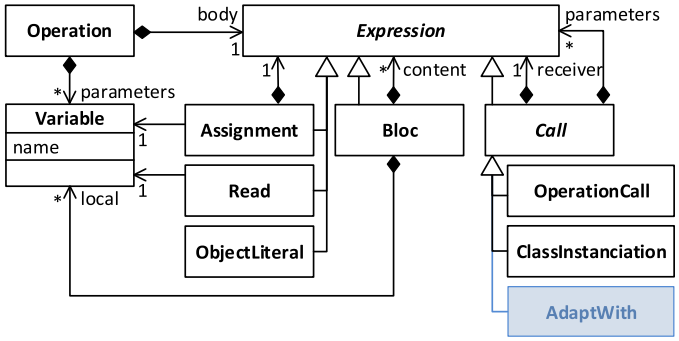

Fig. 7. Lub programming constructs extended with adaptation.

4.4. Operational semantics

The operational semantics of Lub relies on a runtime model depicted by Fig. 8. An instance of a Lub class (*i.e.* an instance of Class in Fig. 4) refers to an instance of class InstanceRecord at runtime. This instance basically stands for a *record* that is referred by a unique identifier (the attribute uuid inherited from class ObjectRecord). A record contains named slots. A slot stores a value conforming to a Lub class attribute (*i.e.* an instance of Attribute in Fig. 4). It can be either a primitive value or a reference to another instance of InstanceRecord.

A record is always associated with exactly one instance of InstanceRecord which plays the role of meta-class instance. It defines 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. This is the standard lookup mechanism as implemented in languages like Smalltalk or Python.

Since Lub introduces a specific construct named Adaptation in order to enable dynamic adaptation, the runtime model of Fig. 8 has to be updated accordingly. The resulting runtime model is depicted by Fig. 9. In this version, an instance of a Lub adaptation (*i.e.* an instance of Adaptation in Fig. 5) refers to an instance of class LUBRecord, which is a new kind of *record*.

An instance of InstanceRecord can be *adapted*, in which case it is associated with an instance of LUBRecord. It can be associated with several adaptations, but only one among them can be active at a time (the *current* one).

The role of meta-class instance is played by an instance of LUBRecord instead of InstanceRecord. As a consequence, meta-objects are out of the scope of adaptation. The meta-class instance still defines the starting point of a method lookup. But now, this starting point can be *modified* by another Lub record. More precisely, when an object receives a message, its lookup starts as previously from the meta-class of the object itself if it is not associated with another lub record. Otherwise, the lookup starts from the meta-class of its current lub record (instance of LUBRecord playing the role of current in Fig. 9).

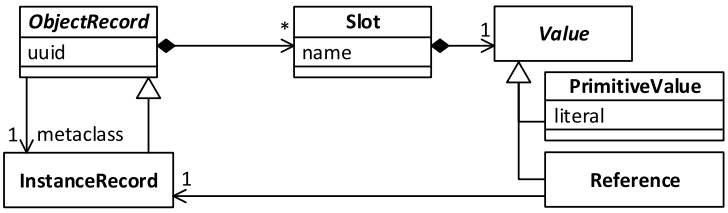

Fig. 8. Lub runtime model.

If the lookup fails in this case, then the lookup starts again from the original meta-class. This second lookup may finally either trigger the target method or raise a *does-not-understand* exception. Notice that when an instance is associated with a specific Lub record, only this particular instance is adapted. All other instances of the same class remain unaffected.

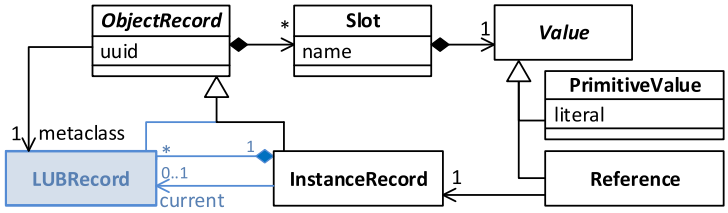

Fig. 9. Lub runtime model extended with adaptation.

As an illustration, we consider again the example of Fig. 3. In this example, an instance of ComPort playing the role of peer is initially associated with an instance of Drone. This state can be represented by a runtime model conforming to Fig. 9. In this model, the instance of ComPort is represented by an instance of InstanceRecord associated with an instance of LUBRecord playing the role of metaclass. In other words, this instance of LUBRecord represents ComPort. In this state, the receiving of readMessage by the instance of ComPort triggers its lookup from its metaclass, *i.e.* from class ComPort.

Now we consider the adaptation depicted by Fig. 3. In this adaptation, the instance of ComPort is associated with an instance of AdaptComPort, which is an adaptation of ComPortLog. In the corresponding runtime model, the initial instance of InstanceRecord is now associated with a second LUBRecord playing the role of the current adaptation. In this state, the reception of readMessage by the instance of ComPort triggers its lookup from the metaclass of its current adaptation, namely class ComPortLog.

5. Implementation

We implemented Lub in Python [12] and Pharo [11], which both provide similar reflection capabilities. Pharo and Python have been extended with the ability to temporarily change the base of the lookup when a message is received by an object. These implementations work on the same test cases, but the Python version is demonstrated on a reduced version of the usecase developed in the next sections to focus specifically on the mechanism in action. Instructions and examples are available online.[1]

The implementation is based on a lookup instrumentation, similar to the Talent solution [13]. The idea is that every message received by an object is intercepted and the lookup is reified and controlled to answer the message, as described in section 4. We chose to implement this lookup control by interposing an anonymous class between the adapted instance and its original class. Only this particular object is affected, and this mutation is transparent to most of the environment tools (to all of them, to the best of our knowledge). In addition, this technique offers good performances against other implementations based on reflection [14].

An adapted object keeps its identity in the sense that no manual copy of it is made during its class mutation. There are no automated migration management implemented to perform this operation. If there is any, it is handled by the virtual machine or the interpreter. From the application's or the user's point of view, the object did not change. From the object point of view, the *self* reference is preserved.

6. Performance analysis

In this section a performance analysis of Lub is made. Its results are compared at runtime against Pharo native mechanisms performances. More precisely, we compare lookup speeds, the impact of the adaptation process and the memory overhead.

[1] http://kloum.io/costiou/lub.

All benchmarks were made on a 64 bits Intel Core i7-4900MQ CPU @ 2.80 GHz ×8 processor with 32 GB RAM. The operating system was Fedora 22 – with Gnome 3.16.2. All tests were performed with Pharo 6.0 64 bits.

6.1. Comparison criteria

The performance evaluation is based on two versions of a base test code implementing some methods. The first version is an extended base code with adapted behaviors as if they were anticipated and thus included in the original design of the application. It is done by conventional means, *i.e.* adding a subclass, specializing some methods and using this new class to instantiate or migrate our objects. The second version is composed of the base code and of Lub adaptations altering the behavior of this base code with the same flavors. It represents a software for which unanticipated adaptations were necessary and performed dynamically. The objective is to evaluate the performances of a program dynamically adapted by Lub against the same program statically adapted by object oriented conventional means.

6.2. What is not measured

We do not measure the time spent to design adaptations nor the time necessary to inject these adaptations into the running system. First, this is not what is proposed with Lub, which only allows to extend an existing language to provide unanticipated adaptations. Second, it depends on tools and techniques, *e.g.* remote debuggers [15]. The choice of these tools and techniques will affect the performances of adaptations injection at runtime, but it is completely independent of the overhead produced by the adaptation mechanism.

We do not measure the cost of iterating over the entities to adapt, *i.e.* the time taken by loops. Again it could be something tool related, and in any case the same time would be spent by loops when iterating to adapt instances in a native Pharo application or in a version of Pharo extended by Lub.

6.2.1. Lookup speed evaluation

As adaptations are made by lookup indirection, the speed of the lookup execution is of critical importance. To measure the impact of Lub on the lookup speed, we measure the difference of execution between two methods implementing the same code. One of this method is statically defined in the class of the adapted object. The other method is acquired with Lub by means of dynamic adaptation. The lookup speed is evaluated for the scenarios described in Table 1.

Table 1
Lookup evaluation scenarios.

Method	Lookup speed evaluation criterion	Source code
m	Call to a non-adapted method	100 printString
m1	Call to an adapted method	100 printString
m21	Call from a non-adapted method to an adapted method	self m1
m22	Call from a new method to an adapted method	self m1
m31	Read/write instance variable from an unadapted method	x := x + 1
m32	Read/write an acquired instance variable from a new method	y := y + 1
m4	Call to an acquired method	100 printString

The time measured is the time to execute 1 000 000 times the same method. This measure is taken 10 000 times in a row for the Pharo lookup and for the Lub lookup extension. To minimize the impact of the garbage collector on the measured speed, a garbage collection is forced before each measure. The two sets of measures are then compared by computing the mean for each version. A 95% confidence interval is given for the mean of the execution time overhead. The *Welch Two Sample t-test* method is used to compute these statistics.

Results of the lookup performances evaluation are shown in Table 2. We can see that the Lub lookup for adapted objects is around 4–6% slower than the Pharo lookup for adapted, non-adapted and new methods (*m*, *m1*, *m21*, *m22* and *m4*). Execution of methods reading and writing previously existing and newly acquired instance variables is slightly faster, but it could be related to the difference in the executed code (see Table 1). However, we can see than accessing original and new states through accessors, respectively methods *m31* and *m32*, induces similar overheads than the other scenarios, respectively around 0.5–1.8% and around 3.6–5%.

To provide more accurate statistics about instance variable access and separate its measure from the lookup execution time, we provide two tests dedicated to the read and write operations. Results are shown in Table 3. We can see that for already existing instance variables, Lub has no performances overhead on simple read/write operations. However, it is a little bit slower than Pharo native mechanisms when performing these operations on newly acquired instance variables.

One drawback when measuring such execution time is that the executed code speed which is supposed to be constant between the different tested methods is itself subject to very small variations. When variations are accumulated a lot of time like in our microbenchmarks, it can make a difference in the overall results. In fact, some tests results showed that Lub is faster than Pharo in specific cases depending on the test scenario (*i.e.* the methods called from Table 1). It is also very common that two sets of measures of the Pharo lookup following the same scenarios presents similar differences in speed

Table 2
Means of lookup execution speed over 10 000 measures of 10^6 executions (in milliseconds).

Lookup	Pharo	Lub	Overhead[a]
m	68.52	72.47	5.44%–6.08%
m1	68.35	72.81	6.43%–6.61%
m21	71.61	74.65	4.18%–4.31%
m22	70.83	74.37	4.93%–5.08%
m31	28.86	29.18	0.47%–1.78%
m32	29.00	30.24	3.61%–4.96%
m4	67.16	70.89	5.50%–5.61%

[a] 95% confidence interval.

Table 3
Means of state access execution speed over 1000 measures of 10^7 executions (in milliseconds).

Inst. var.	Profiled code	Pharo	Lub	Overhead[a]
Existing	$x := x + 1$	278.54	274.53	−3.75%–0.88%
Acquired	$y := y + 1$	273.621	279.991	−0.04%–4.70%

[a] 95% confidence interval.

as in the results presented above. We choose to present these results in particular because they show one of the highest gap in performances from our tests, and therefore represent a worst case scenario of speed loss when using Lub.

6.2.2. Objects adaptation time evaluation

When an object is adapted, it is mutated to a modified version of itself: it can acquire or modify behaviors and states. Performing this mutation at runtime has a cost, as the system has to wait until all concerned objects are adapted.

A first evaluation studies the cost of the migration of 1000 objects to one specialized version of themselves (one adaptation). A second evaluation studies the cost of migrating all these objects to a specialized version of each instance, which means that there would exist at runtime 1000 adaptations. Results are shown in Table 4. The same two evaluations are made to study the impact of reverting adaptations at runtime and are shown in Table 5. All objects are originally instances of a class *C0* with two methods *m1* and *m2* and an instance variable *x* initialized to 0. They are adapted with an adaptation which adds two new methods *m3* and *m4*, overrides *m1* and *m2*, and adds a new instance variable *y* which should be initialized to 1000000. We measure specifically the following items:

- The time necessary to migrate objects to their adapted version. Migration includes the modification of the instance and the initialization of its new instance variables.
- The time spent in the garbage collector. This time is an additional constraint at runtime because it is induced by the migration operations.
- The time spent in Lub mechanics, used to optimize the migration speed.
- The time spent to compile modified or new methods and states in the new versions of the objects.

Table 4
Impact of adapting objects in the runtime system (in milliseconds).

1000 objects	Migration	GC	Memory	Optimization	Compilation	Total
1 adaptation•	846	1	867 KB	2	8	905
1000 adaptations*	1162	148	2.7 MB	914	4882	7106

• 1000 objects are adapted by the same adaptation.
* 1000 objects are each adapted by one different adaptation.

Table 5
Impact of reverting adaptations in the runtime system (in milliseconds).

1000 objects	Migration	GC	Memory	Optimization	Compilation	Total
1 adaptation•	6	0	394 KB	0	0	6
1000 adaptations*	16	0	417 KB	0	0	16

• 1000 objects are adapted by the same adaptation.
* 1000 objects are each adapted by one different adaptation.

Migrating objects (Table 4) can have different effects on the system. If all objects are adapted with the same adaptation then the execution time is considerably shorter than when mutating each object with a specific adaptation. Migration times

Table 6
Memory overhead of Lub adapted object against Pharo objects (in bytes).

Instances	10•	100•	1000•	10⋆	100⋆	1000⋆
Pharo	402	1842	16242	2580	25800	258000
Lub	446	1886	16286	3020	30200	302000
Overhead	11%	2.4%	0.3%	17%	17%	17%

• One adaptation is provided for all objects.
⋆ One adaptation is provided for each object.

are not significantly different compared to the compilation time which depends on the number of individual adaptations. Optimization operations facilitate the mutation of a lot of objects by a few adaptations, and greatly reduces the compilation time when there is not a lot of different behavioral variations for instances of the same class. However, this advantage reduces with the variability of objects to adapt and adaptations. The more the objects are of different kind and the more they are specialized, the more the migration cost is high because of compilation and optimization operations. We can also see the impact in terms of memory: this value represents the size of objects in memory which live during the migration and which should be removed afterwards. In our case these objects have been garbage collected.

Compilation and optimization times could be reduced to zero if they were performed offline. All necessary information could be retrieved from the system and adapted code generated remotely before being injected in the system. The cost would be reduced to migration, memory overhead (although reduced) and the execution and memory overhead produced by the tooling environment used to perform these operations. Current implementations of Lub do not provide such tools.

The cost in execution speed of reverting back objects to their original behavior (Table 5), *i.e.* canceling adaptations, is very low. In both cases (1 or 1000 adaptations), reverting 1000 objects to their original version is very fast compared to the migration time. The induced memory overhead is not negligible, but it is equivalent independently of the number of adaptations and the kind of objects to rollback.

6.2.3. Memory overhead evaluation

To evaluate the impact on memory of unanticipated adaptation with Lub, we compare it with an hypothetical adapted application in which objects are adapted by a dynamic software update. This update provides, in the most extreme case, a specialized version for each of the concerned instances. We are interested in the memory overhead induced by Lub against this hypothetical adapted version of the application. This memory overhead is measured when all objects of interest are effectively mutated and does not consider migration times.

In this scenarios, an original class *C0* has two methods *m1* and *m2* and an instance variable *x* which is initialized to 0. These two methods just execute the code *100 printString*. 1000 instances of this class exist at runtime and they need to be adapted. A simple way of adapting these instances with specialized behavior would be to subclass *C0*.

Scenario 1• Each object needs to be adapted with the same behavioral variations. *C1* is a subclass of *C0* which implements this adapted behaviors. For the sake of simplicity and to ease the comparison with Lub, this subclass implements two new methods *m3* and *m4* and also overrides *m1* and *m2*. All these methods implement the same code as the original methods from *C0*. A new state, *y*, is also added to the subclasses and is initialized to 1000000. Once the thousand object are migrated to their new class, the space they occupy in memory is measured. Then in an other test, a Lub adaptation will be applied to the same amount of *C0* instances and will bring the same behavioral variations (*i.e. m1, m2, m3, m4* and the new *y* instance variable). The measure is repeated again.

Scenario 2⋆ To provide really specific specialization to each one of these objects with object oriented mechanisms, we would need to create a thousand subclasses of *C0*. Each object is then migrated to its own specialized class. To ease the comparison with the first scenario and Lub adaptations, each of the subclasses implements the same behaviors and states mutations as *C1*. Similarly, a thousand adaptations are designed on the same model and applied to one target instance each. Measurements are repeated.

Results of memory overhead measurements are shown in Table 6. We can see that for the first scenario, using Lub induces a memory overhead of 11% for 10 adapted instances. The difference is proportionally significant, but the absolute difference between Lub and Pharo is not (44 bytes out of 446 bytes). If more instances are adapted, this overhead tends to decrease. The cost of adapted behavior was paid when the first adaptations were made. Since the entities providing these behavior variations are already in the system, the only cost left is the memory occupied by the instances themselves, which is equivalent to the cost of regular objects with the same definition (methods, instance variables).

However, as we can see for the second scenario, when adapted entities are of different kind or are adapted with different adaptations (which is the case here), the cost in memory is higher and remains constant whatever the number of adapted instances. This cost is due to the meta entities created or injected to provide runtime adaptations for one single specification. For a given adaptation, *e.g.* acquiring a new method, the cost in memory is fixed. Providing many different adaptations will add as many times this memory cost to the system. Adapting a large number of objects will then have a significant memory impact on the system.

Notice that these numbers are valid for the described example which specifies simple and limited modifications of behaviors and states. The 17% memory overhead could significantly increase if, for example, a lot of objects with a few methods were extended by deep modifications and complex additions. However once all the adaptations are in place, the overhead either tends to decrease with the number of adapted objects or tends to remain constant if adaptations are heterogeneous.

7. Evaluation

In the following examples, we explore and evaluate Lub. We first go through the lookup mechanism (section 7.1) and then we address the drone example with Lub (section 7.2). We chose to demonstrate Lub with a Pharo implementation, because it provides a very rich integrated tooling environment [16] from which we can fully benefit (debuggers, inspectors, etc.). This implementation is available in a dedicated Pharo image and includes all the examples depicted in this section. A step-by-step tutorial is available[2] to help reproduce the results presented in this paper.

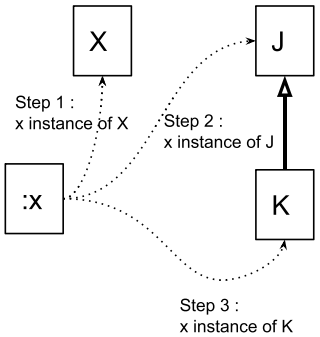

Fig. 10. When an adaptation is applied to the object x, this instance behaves partially as an instance of another class.

7.1. Lookup through object's adaptations

In this example, we will illustrate the effects of the Lub lookup, and its impact on the Smalltalk lookup. Following a methodology inspired by Beugnard [17], we declare three classes (X, J and K) with four methods (m1, m2, m3 and m4):

```
Object subclass: #J
  m1 ^'J.m1()'
  m2 ^'J.m2()'

J subclass: #K
  m3 ^'K.m3()'

Object subclass: #X
  m1 ^'X.m1()'
  m4 ^'X.m4()'
```

K inherits from J and X is apart. We now want to define two adaptations, 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 understand a message, an *error* string is printed. Adaptations are defined as followed:

```
1  lubJ := Adaptation targetClass: J.
2  lubK := Adaptation targetClass: 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 extended by the adaptation *lubK* will not be allowed to access m2. Now, each of these methods will be successively called on an instance of X and results will be logged. This will be our first test sequence. After this first sequence, the *lubJ* adaptation will be applied to the same object, and the same test sequence will be executed. The same protocol will be repeated again for an adaptation with *lubK*:

[2] http://kloum.io/costiou/lub.

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

"Second test"
x adaptWith: lubJ.
x m1; m2; m3; m4.

"Third test"
x adaptWith: lubK.
x m1; m2; m3; m4.
```

A simple illustration of these adaptations is depicted in Fig. 10 where we can see how adaptations change the lookup semantics. Results are shown in Table 7. We can see that for m1 and m3, the lookup behaves normally as if x was successively an instance of X, J and K. Calls to method m2 for x as X raise *does-not-understand* exceptions, because it is not part of its behavior. For x as J the method is found by the lookup in class J. However for x as K, the adaptation does not allow lookup access to the m2 method in K: the lookup is directly started in X and also raises a *does-not-understand* exception. 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 always able to respond to m4 which is part of its original behavior.

Table 7
Lookup results through adaptations updates.

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()

The lookup extension based on Lub does not impact the original lookup, except if it is specified by an adaptation in which case it behaves as described by Lub. Adaptations can be defined to target and adapt specific behaviors. When an object has not been modified by adaptations or for behaviors that are excluded from adaptations, the lookup remains consistent with what we would expect from a Smalltalk system. When behavior adaptation has been specified with an adaptation, the lookup starts in the class pointed by the specification. It can be caught and routed back to the original class of the object if it fails or if it redirects to a behavior that is not meant to be adapted. The *self* reference of the object is always preserved: the lookup for messages sent to *self* behaves the same as in Table 7.

7.2. The GPS loss: dynamic unanticipated behavior adaptation in a fleet of drones

The following evaluation presents a scenario of unanticipated adaptation in a fleet of drones. It is not meant to prove Lub is usable on a real case of drone adaptation. It is meant to show how code and behaviors could be adapted at runtime from a pure software perspective. Therefore it is a simulation that we designed to emphasis how a running software is investigated and modified through unanticipated dynamic adaptations.

7.2.1. Studied scenario

We implemented the scenario as a simple simulation of our fleet of two drones, *gps-drone* and *follower-drone*, from section 2.1. The drones log their actions into a transcript console. They also have the capability to communicate with each other and to know the exact distance to each other. The original behavior of the drones is to move on the *y* axis of a virtual grid. This grid is composed of areas that can be secured or unsecured. In the latter case, the drones must cipher their communications. The simulated time is relative. The *follower-drone* must rely on the GPS of *gps-drone* to position itself. Otherwise, it cannot move and the whole fleet enters a *stand-by* mode.

In this section we play a small scenario where the *gps-drone* enters an unsecured area and starts ciphering the messages it sends to the other drone. However the *follower-drone* did not enter the area yet and therefore is not using the encryption mode: it cannot understand messages from *gps-drone* anymore. The mission is monitored by an hypothetical human operator to whom the situation is unexpected: he has to understand what happens and to adapt the behavior of the fleet.

We consider this scenario as a simulation only, which is a first step before experimenting on a complete and fully operational system. We suppose that there are no loss nor interruptions of communication with the drones and between the drones. If that were to happen, it could be managed by tools that we do not provide to ensure seamless communication with the user. We focus on the study of the adaptation mechanism implemented following the Lub language pattern. Research about tools to solve such problems is out of scope of this paper.

Complete loss of communication with one of the drones could not be easily fixable with unanticipated adaptations. However in that case, the use of any dynamic solution requiring interaction with the user seems compromised, as there would be no means of communicating with the remote devices.

7.2.2. Step-by-step adaptations

We focus on a small part of the model that we will adapt, which is the drone-to-drone communication feature. First, they use what we call *communication channels* modeled by a class named *CommunicationPort*. Each drone has one drone-to-drone communication channel and one drone-to-operator channel that logs information about the simulation in a user side console, as depicted in Fig. 2 from section 4.1. The instances of *CommunicationPort* in charge of drone-to-drone communication can send messages to each other. For example *follower-drone* will use its channel named *peerDroneComPort*, instance of *CommunicationPort*, to send messages asking GPS coordinates to *gps-drone*. These messages are received by its mirror instance variable in *gps-drone* which can answer following the same process. Behaviors to send and receive messages are defined as methods *sendMessage:* and *receiveMessage:* in class *CommunicationPort*. Once received in a communication channel, the message can be decrypted and forwarded to its drone that will later process all the messages waiting in the message queue. This processing occurs in the class *LSDrone* in the method *proceedMessageQueue*. The GPS behavior is implemented in class *GpsDrone* from which *gps-drone* is an instance while *follower-drone* is an instance of *FollowerDrone* which implements the behavior to follow a drone with a GPS. Excerpts of the model are presented below:

```
"Base class for the simulation drones"
Object subclass: #LSDrone
    instanceVariableNames: 'name operatorComPort peerDroneComPort messageQueue safeMode y x'

"Instances of GpsDrone class can use a GPS to pinpoint their position"
LSDrone subclass: #GpsDrone
    instanceVariableNames: 'gps'

    "The proceedMessageQueue method in GpsDrone process incoming messages depending on their type.
    Messages are received and sent from an instance of FollowerDrone."
    proceedMessageQueue
        [... ]
        msg := messageQueue next.
        "If the message requests the gps position, then it is answered back"
        msg messageAction = LSComMessage messageActionPinpoint ifTrue: [ self answerPinpointRequest ].

        "If the other drone entered the safe mode, the system also enters safe mode"
        msg messageAction = LSComMessage messageActionSafeMode ifTrue: [ safeMode := msg messageValue ].

"Instances of FollowerDrone class can pinpoint their position using the gps of a GpsDrone instance"
LSDrone subclass: #FollowerDrone
    instanceVariableNames: 'peerDronePosition distanceSensor encryptedMessagesMode'

    "The proceedMessageQueue method in GpsDrone process incoming messages depending on their type.
    Messages are received and sent from an instance of GpsDrone."
    proceedMessageQueue
        [... ]
        msg := messageQueue next.
        "If the message is an answer to a gps request, then it is processed and the gps position is recovered"
        msg messageAction = LSComMessage messageActionPinpointAnswer
            ifTrue: [ self peerDronePosition: (self decrypt: msg messageValue) ]

"Communication port are drone−to−drone or drone−to−operator communication channels"
Object subclass: #CommunicationPort
    instanceVariableNames: 'remote owner name waitForConnection'

    sendMessage: aMessage
        "Sends a message through a communication channel"
        [...]

    receiveMessage: aMessage
        "Receives a message through a communication channel"
        [...]
```

We can follow the code execution through the following log traces. The simulation starts and both drones use their operator communication channel to log their status. They also use their dedicated communication channel to communicate with each other. These four objects, one operator and one dedicated communication channel per drone, are instances of class *CommunicationPort*. The simulation starts as followed:

```
1  t = 5
2  gps−drone moves to : 200@25
3  follower−drone moves to : 100@50
4
5  t = 6
6  gps−drone moves to : 200@50
7  follower−drone moves to : 100@25
8  follower−drone lost its GPS signal
9  follower−drone is in safe mode
10
11 t = 7
12 gps−drone is in safe mode
13 follower−drone lost its GPS signal
14 follower−drone is in safe mode
```

At simulation time t = 5 (line 1), the *follower-drone* communicates with the *gps-drone* to use its GPS coordinates and position itself. However it enters a *safe mode* at t = 6 (line 5): it cannot move anymore. As a response, the *gps-drone* (t = 7, line 11) also enters a *safe mode* and the fleet is stuck. From a human operator point of view, the situation is not obvious. In our case, the communication between the drones is fine. We suppose that there is some way for the operator to know it, like a green light for the drone communication indicator on a control screen.

We could monitor in *follower-drone* the received GPS data from *gps-drone* to see if something is wrong with it. However this behavior is not present in the system so that is not possible *as is*. Furthermore we do not know which entity to monitor. We need a debug capability which is clearly unanticipated, as the time of the adaptation as well as the entity to adapt were unknown beforehand. We propose to adapt the behavior of the communication channel between the two drones in *follower-drone*, to log the received messages from *gps-drone*. The class *ComPortWithLog* defines an interesting behavior similar to class *CommunicationPort*, except that it implements a logging mechanism. *ComPortWithLog* is defined as followed:

```
1  Object subclass: #ComPortWithLog
2    logMessage: aMessage
3      "logs a communication message"
4      [... logging code ...]
5
6    receiveMessage: aMessage
7      self logMessage: aMessage
8      [... original receive message code ...]
9
10   sendMessage: aMessage
11     self logMessage: aMessage
12     [... original send message code ...]
```

We would like to use part of this class' behavior to adapt the communication channel in *follower-drone*. We are interested in adapting only the logging mechanism when receiving a message and not when sending a message to the other drone. This behavior selection is done by defining an adaptation as followed:

```
1  lub := Adaptation targetClass: ComPortWithLog with: #(#receiveMessage: #logMessage:).
2
3  followerDrone peerDroneComPort adaptWith: lub
```

The adaptation is then applied to the drone communication port of *follower-drone* to adapt its behavior. Live access to this object is easy in Pharo: introspection allows to inspect objects and their current states. We suppose that running the simulation opens an inspector on the simulation itself, allowing to browse objects and to retrieve the *followerDrone* instance. Now that it is possible to monitor data received in *follower-drone*, we expect to see coordinates in the answer of the GPS request. For example as the last coordinates of *gps-drone* are *200@75*, the expected answer data would be *200@75*.

```
1  t = 10
2  gps−drone at: 200@75
3  gps−drone is in safe mode
4
5  follower−drone at: 100@25
6  ### follower−drone receiving message: pinpointAnswer
7  ### contents: #[50 48 48 46 55 53]
8  follower−drone lost its GPS signal
9  follower−drone is in safe mode
```

We can see that the communication port of *follower-drone* has been adapted and now logs the data it receives. The other objects instances of the same class (*CommunicationPort*) have not been affected and only the adapted instance logs its received data, so operator communication channels and the communication port in *gps-drone* have not been affected. They

do not log received data. Messages sends are not logged either, even if it is part of the behavior from class *ComPortWithLog*. This is due to the behavior selection in the adaptation definition.

In our scenario, the operator realizes that the received data is encrypted because *gps-drone* entered an area that needs data encryption. The other drone has not been notified of this change of context and therefore does not know it now receives encrypted data. This notification mechanism is missing from the original design. To adapt to this context, we need *gps-drone* to detect when the communication security policy changes and notify *follower-drone* by sending it a message. Then *follower-drone* needs to be able to interpret the message and to enter a secure communication mode.

To adapt the drones, we model two classes: *AreaTracker* and *MessageQueueWithEncryptionState*. The *AreaTracker* owns an instance variable (*currentArea*) that stores the current area. This is a new state that is not present in the original drone object. The new behavior is to track the changes of areas. It also implements behavior to notify the other drone if communications must be encrypted or not, depending of the current area. This notifications are sent only if there is a context change, *i.e.* if the current area changes and if the encryption mode changes accordingly. We will adapt *gps-drone* with this new behavior. The second class, that models the behavior to adapt *follower-drone*, is the class *MessageQueueWithEncryptionState*. It adapts the behavior of the message processing to change the encryption policy upon reception of the new messages sent from *gps-drone* after its adaptation with *AreaTracker*. Excerpts of these classes are shown below:

```
1  Object subclass: #AreaTracker
2     instanceVariableNames: 'currentArea'
3
4     "Retrieves the current area: if the security policy has changed then it is broadcasted to other drones"
5     getCurrentArea
6     | area |
7     area := [... get the current area ...]
8     self currentArea = area
9        ifFalse: [ area isSecured
10             ifTrue: [ self sendQuitEncryptModeMessage ]
11             ifFalse: [ self sendEnterEncryptModeMessage ] ].
12    self currentArea: area.
13    ^ area
14
15    sendEnterEncryptModeMessage
16    "sends a message to follower−drone to indicate the
17     fleet enters a secure mode"
18
19    sendQuitEncryptModeMessage
20    "sends a message to follower−drone to indicate the
21     fleet exits a secure mode"
22
23 "The MessageQueueWithEncryptionState adds management behavior for the state of the encryption policy."
24 Object subclass: #MessageQueueWithEncryptionState
25
26    proceedMessageQueue
27       [... original message processing code ...]
28       messageAction = #encryptMode
29          ifTrue: [self encryptedMessagesMode: messageValue ]
```

Now we define two adaptations to adapt our object: one to adapt the *gps-drone* that we can retrieve in the temporary variable *gpsDrone*, and one to adapt *follower-drone* in the temporary variable *followerDrone*. As explained above, these references to our objects can easily be retrieved with the Pharo inspection tools, and are presented here as if their references were found during runtime in the Pharo inspector. The adaptation is done as followed:

```
1 areaTrackerLub := Adaptation targetClass: AreaTracker.
2 areaTrackerLub initializeWith: {(#currentArea −> nil)} asDictionary.
3 gpsDrone adaptWith: areaTrackerLub.
4
5 messageEncryptionLub := Adaptation targetClass: MessageQueueWithEncryptionState.
6 followerDrone adaptWith: messageEncryptionLub
```

The *gpsDrone* object is adapted with new behavior and also new state: the *currentArea* instance variable. Its initialize value is pre-set during the definition of the adaptation (line 1-2) and once the object is adapted (line 3), the new instance variable is initialized with the set value. These adaptations produce the following output on the log view:

```
1  t = 12
2  gps−drone at: 200@75
3  gps−drone is in safe mode
4  gps−drone starting encryption mode.
5
6  ### follower−drone receiving message:
7  ### encryptMode
8  ### contents: true
9  follower−drone at: 100@25
10
11 t = 13
12 ### follower−drone receiving message:
13 ### pinpointAnswer
14 ### contents: #[50 48 48 46 55 53]
15 gps−drone moves to : 200@75
16 follower−drone moves to : 100@50
```

At t = 12 (line 1) *gpsDrone* does enter the message encryption mode. The *follower-drone* receives a message that is logged by the previously adapted communication channel. At t = 13 (line 11), upon reception of the GPS request answer, both drones start moving again. We can deduce that first, *gpsDrone* has detected the change of area and sent a message to tell *follower-drone* that next messages will be encrypted. Secondly, as the two drones resumed their course, we can deduce that *follower-drone* has been able to decipher the message and to pinpoint its coordinates. Object's behaviors are adapted and the situation is resolved.

However, the communication channel of *follower-drone* is still logging received data. This was an adaptation meant to understand the problem. It has no more purpose now that the fleet of drones can adapt to these contexts of area changes. The *CommunicationPort* instance needs to recover its original behavior. The following listing shows at line 1 how we remove an adaptation from the object and the log just after (line 3) shows that the logging behavior has been removed.

```
1  followerDrone peerDroneComPort adaptWith: nil
2
3  "Log results"
4  t = 14
5  gps−drone moves to : 200@100
6  follower−drone moves to : 100@75
```

Throughout these adaptations, we can see that objects preserve their states. For example, the drones have the same names and the same coordinates before and after the adaptations. References are also preserved, which is implicitly shown in the logs: the communications channels instances of *CommunicationPort* in each drone are still connected to their remote sibling in the other drone. Therefore they can continue to send and receive network messages.

7.3. Summary and discussion

In both examples, running code has been updated with non-intrusive adaptations. The original code is preserved: old behaviors are dynamically replaced by new behaviors injected at runtime. The choice of which behavior to execute is achieved by the lookup indirection specified by an adaptation. New states can be acquired and the class where the lookup starts from can change, but states and relationships of an object remain unaffected: adapted objects preserve their identities.

Section 7.1 showed how the lookup behaves according to Lub adaptations. An object can adapt by acquiring new behavior or overriding part of its behavior, by starting the lookup in another class than its own. It also showed that during these adaptations, the *self* reference is constant, and always references the adapted object whatever the class in which the lookup is performed.

Section 7.2 depicts a scenario of the fleet of drones use-case that showed how a plausible case of unanticipated context requires unpredictable adaptations at runtime. A first adaptation to probe a particular object is done, without affecting other objects of the same class. Then a second adaptation is designed and applied once the problem is found. One of these adaptations requires a new state that is acquired by the adapted object, but all prior states are preserved during the scenario and until its end. This is a typical case of unanticipated adaptation: the problem is clearly due to a bug, which is by definition not foreseen at design time. Until it happens and until a solution is found, the adaptation remains unknown. In this scenario, this also leads to an adaptation of a whole fleet, because it is necessary to adapt behavior in the two drones to solve the problem.

Understanding runtime problems is facilitated by the live access to the context of running programs. Probing and experimenting through adaptations allows the developer to dive in the possibly complex environment of the software. Without this possibility to access the context, finding and understanding all necessary conditions to reproduce a bug can be a challenging task. However it can prove difficult to identify an entry point in the system. As we investigate and adapt object's behaviors, one must know which object to observe and to change. A lot of objects can live in a running program, some of them with a limited lifespan. Choosing which object to adapt and how to design adaptations necessarily rely on the

developer's understanding of the software's design and depends on his interpretation of information provided by the live context.

Furthermore, every kind of problem cannot be fixed by means of unanticipated adaptation. For example, if the application is stopped because of a bug, or if the bug affects the communication with the developer. Uncaught exceptions may provoke non-deterministic behaviors if they are not trapped and forwarded to an hypothetical connected debugger. These kind of disruptive problems could prevent the use of unanticipated adaptation mechanisms to fix the software.

8. Discussion

In the following section we discuss the Lub pattern and its evaluation in regard to the literature and existing solutions.

8.1. Unanticipated adaptations of running systems

Behavioral variations can be dynamically applied on a running system to assist the debugging activity: first it can bring unanticipated observation capabilities to the program and second it can actually adapt the functional behavior of the application. The subsections below discuss these two steps of the adaptation process.

8.1.1. Observing running software

Adaptations can be used to observe and build views of a running system. Most evident use cases are understanding or debugging systems. However such instrumentation is not always possible nor desirable, for example if the software has not been designed with this capability or if the instrumentation involves altering the system's behavior, performances or stability. Even if observation is possible, it also needs to be flexible enough for the developer to express under which conditions he wants to inspect the system. Discussions about this capability to look into a running system are numerous, for example to investigate performance problems and bugs [18,8,19,20], to monitor instance variable access at runtime [21] or to provide on-demand views of a system for experts and end-users [22]. An interesting point is that none of these solutions alters the core behavior of the software: the business code is never impacted as adaptations are only meant to observe the system. This is what we achieved as demonstrated in section 7.2: a key object in the system has been adapted to log information but its core behavior remained untouched. This instrumentation capability without affecting the business behavior can be seen as a first step before performing real adaptation. Indeed, when facing unanticipated behavior adaptation, one must first understand what is going on before triggering behavior changes.

8.1.2. Adaptation as an evolving capability

After having observed the system, behavior adaptation can be decided. That implies the chosen solution to be also capable of modifying functional behavior. *H-Graphs* [22] allow to build custom views and browsers of a live system but are not designed for behavior adaptation. *DiSL* [21] also specifically targets dynamic observation of the running code but cannot be used to evolve the software.

Lyrt [23,24] and SMAG [25] are frameworks based on roles, in which dynamic restructurations of the system's objects or components can be done to provide behavior variations. In both cases unanticipated adaptation is achieved by uploading new roles or components and changing the relationships between the software's running entities. Adaptations are not modeled as classes but as components or roles entities. These entities are then composed between them or with classes or objects to provide adapted behaviors. In Lub, an adaptation is modeled as a class with a selection of structural features. The dynamic adaptation capability is integrated at the meta-level, but it does not shift the programming model. The expression of adaptations and their application on an object remain in the object-oriented paradigm.

Context-Traits [26] provide dynamic trait composition at runtime. Traits now affect single objects and can change behaviors on an instance basis. However composition brings conflicts that must be resolved explicitly before being applied. Composition Policies are used to automatically resolve conflicts at runtime. In Lub, we consider behavior composition and conflicts resolution to be a design concern and not an adaptation concern. Composition can be done while designing an adaptation, for example using a dedicated behavior composition mechanism like *regular* Traits [27]. After that, the adaptation is specified to target this composed class and applied to a given object using a higher level semantics. An other inconvenient while using traits is the need for *glue-code*, because traits can require interfaces that must be implemented somewhere. It puts an additional constraint on the software design.

Reflectivity and Geppetto [8,19] bring the concept of instrumenting Abstract Syntax Tree – AST – nodes, and therefore can modify methods or even variable access at runtime. It can be done with a very fine grain, for example by modifying a method before, after or instead of its original behavior. However as AST nodes are located in classes, when performing such instrumentation all instances of a given class are affected. As it is based on AST instrumentation, these solution are quite low level and require an advanced knowledge of the system.

Talents [13] are objects that model adaptations. A single object can acquire or lose dynamically a talent at runtime, thus acquiring or losing behavior. The developer must instantiate a new Talent object, model the behavior (*e.g.* define new methods from scratch) and ask the object to acquire the talent object. Talents can also be composed and conflicts have to be resolved manually. However, there are no abstraction nor control structure to properly handle adaptations, which also makes Talents a low level solution.

Iguana/J [18] allows the definition of meta-objects organized in consistent Meta-Object Protocols (MOP). MOPs can alter behavior for a class, for all instances of a class or for single objects. Non-functional behaviors can also be modified by this means. Methods execution is intercepted and forwarded to the MOP in charge to provide behavioral variations for the target entity (objects or classes). MOPs can be dynamically defined, uploaded and connected to objects. Method interception only occurs for adapted objects and not for other running entities. Dynamic adaptation using Iguana/J is demonstrated in [18] through the same example from section 7.2, in which an object from the class *CommunicationPort* has two adapted methods *sendMessage:* and *receiveMessage:*. To completely adapt behavior of these methods with Iguana/J, the developer has to think and to write non-functional behavior by manually instrumenting the lookup. In Lub we just define an adaptation: it targets a class and selects part of its interface (possibly with new behaviors). The lookup is automatically instrumented to find the selection of behaviors in the target class. It is not required to manually route the lookup through non-functional behaviors.

Other approaches from a pure language perspective have been developed, *e.g.* for Java [20] for which the Java Virtual Machine is modified to provide necessary reflection and adaptation mechanisms. Dynamic unanticipated evolution is granted, but these solutions are usually not portable and remain specific to their implementation language. Modifications of virtual machines make the language less portable, but allows better optimization for performances.

8.1.3. Restoring behaviors

The capability to go back to a previous behavior (possibly the original one) is mandatory to build visualizations of the application's internals and to experiment adaptations at runtime. This is not easily possible if the functional behavior of the application evolved, for example with a patch. The old behavior might not be available anymore. A solution could be to inject a new evolution with the original behavior that cancels the dynamic changes that were made. This adds a lot of constraints and complexity to the adaptation process. This behavior rollback is more flexible with solutions in which adaptations are dynamic associations of objects (or classes) with meta-objects (*i.e.* the adaptations), such as Reflectivity, Talents, Lyrt or Lub. Reverting an object to its original behavior can be done by removing its associated adaptations from the running system (or by disabling the associations).

However it is worth noting that while reverting a given behavior to a previous version (possibly involving multiple methods), there can be states inconsistencies. An adaptation using an instance variable of an object could leave this variable with a state that is not compatible anymore with the original behavior. This should ideally be handled by the underlying adaptation mechanism used to extend a language with Lub. If not, the developer has to take care that states inconsistencies due to behavior rollback never happen. Another possibility, though outside the scope of this paper, would be to validate an adaptation before applying it to ensure that such inconsistency cannot occur.

8.1.4. Enabling dynamic adaptation on running systems

Chisel [28] is a generic framework for dynamic unanticipated adaptation. It features adaptation of objects and classes, which are associated with meta-objects providing mutated non-functional and functional behaviors. These associations are driven by *user-defined* policies. As a full framework, Chisel provides all necessary tools to perform dynamic runtime adaptation (introspection of the running code, behavior update, etc.). As it is generic, it can be implemented on top of any technology and can be used to develop applications for any system. However it could be difficult to use the framework on an already running application, developed by other means before knowing that unanticipated adaptation would be needed. Injecting the framework and its dependencies on a completely independent system could not be possible. Chisel uses Iguana/J as an adaptation backend, which could be replaced by another adaptation mechanism (*e.g.* Lub) as long as the same adaptation capabilities are provided.

Systems based on Components [29–34] can also achieve dynamic adaptations. An adaptation is usually done by reconfiguring components and their relationships in the running system. Lots of research has been done on the efficiency, the safety and the performances of such reconfiguration. Some of these systems provide full dynamic adaptations, by defining and uploading new components and re-configuring the application. Some of these systems focus on static automated adaptations, but they could be upgraded with the capability to upload new components at runtime. For the same reason as above, the original software must be designed and developed with a component system providing dedicated adaptation mechanisms. If a running application is not based on components, it is not possible to completely change the software to enable components reconfiguration.

Keeney argues in [9] that, to enable unanticipated adaptation, *"only the need to support completely unanticipated dynamic adaptation at runtime must be anticipated at or before runtime"*. This is emphasized in solutions like Chisel or solutions based on other paradigms than object oriented, for example components or roles, in which dynamic adaptation or reconfiguration is enabled by default. However it is conceivable that *"the need to support"* this kind of adaptation was not anticipated or was left aside for specific reasons. Solutions like Lub, Talents or Reflectivity for example could be loaded at runtime as application libraries and dynamically enable unanticipated adaptation. This is possible because they do not put design constraints on the software and because involved concepts are only classes and objects.

8.1.5. Contextualizing adaptations

A very important aspect of the original problem exposed in 2.1 is that there is no need for adaptation until the software is running in production, in highly unpredictable environments, which leads to problems that cannot be foreseen at design time. If we take back the system to a development environment (or to the factory), then problems could be gone. This

is a classic case met by a lot of developers. It is obvious then that living in an external environment, the software meets different contexts that cannot be reproduced in development mode. Context Oriented Programming (COP) [35,36] allows the definition of contexts and behavioral variations: when entering a specific context or composition of contexts, specifics compositions of behaviors are selected for execution. It is semantically possible to define context specific behaviors, and the software can adapt to its contexts. However the main drawback is that context definition is made at design time and once started, new contexts cannot simply be added to the application.

Many context oriented languages have been implemented over the years, in the form of language extensions, like, among others, ContextL for CLOS [37], ContextS for Smalltalk [38] or ContextJ for Java [39]. In most of COP languages, contexts are defined as *layers*. 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 control flow, it is also scoped to a specific thread and fine management of layer activation is difficult.

There have been research on the specific problem of layer activation to overcome the control flow limitation. EventCJ [40] is a DSL that brings a modular control of layer activation, making it possible to manage layer transitions based on events. A layer transition can then be triggered from outside the base program thread and affect a single object (adaptation is instance based). Von Löwis et al. [41] 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 (the *with* keyword) is no longer necessary. Despite the possibility for layer management to rely on external events and/or states provided by these solutions, unanticipated adaptation needs more flexibility. The adaptation strategy may not be known until context related problems happen and binding software design to a predefined strategy (*e.g.* based on events) can limit adaptation possibilities.

The Epsilon model [42] proposes to use *roles* as a mechanism to model dynamic behavior variations. Roles can dynamically be bound to objects to adapt their behavior to their environment, which they call *Context*. The EpsilonJ language, based on the Epsilon model, allows the definition of roles implementing and composing context-specific behavior. NextEJ [43] brings type safety and role scoping to the Epsilon model. As for layers, roles in NextEJ are only active within a scope. Furthermore, roles are preserved upon deactivation, allowing bound objects to reactivate them later and recover their context related states. Both NextEJ and EpsilonJ provide behavior adaptation on a per-instance basis, but they do not allow the definition of new adaptations at runtime.

More recent research focus on generalized layer activation mechanisms [44]. It brings layer activation flexibility with per-instance and per-thread adaptations. However, layers and adaptations must be known before runtime.

The main problems of COP for unanticipated adaptation lie in the constraint to design contexts and layer activation before runtime, and in the difficulty to provide instance-based adaptations.

8.2. Implementation perspectives

We consider Lub as a language pattern designed to extend an existing language, providing new unanticipated adaptation capabilities. The implementation choice can be flexible as long as it provides the necessary features to meet the constraints of unanticipated adaptation. We also want to maintain the ability to reason about classes and objects, as we are integrating with object-oriented systems. Although, as previously discussed, we need a solution meeting our constraints as much as possible, as defined in section 3.6.

Many of the technologies discussed above could serve as adaptation backends for Lub. Other techniques could also be used, for example dynamic Aspects Oriented Programming [45–48] (AOP). AOP is meant to express cross cutting concerns rather than only dynamic adaptation, although it is a powerful means to provide behavioral variations. As such, dynamic AOP is too low-level and it shifts the programming paradigm. However it could be used as a backend, as it is common in COP languages and other adaptation mechanisms.

As explained in section 5, we chose to implement Lub with a solution very similar to Talents [13]. As we only manipulate objects and instrument method lookup, it is easy to build a light implementation of Lub as we describe it in section 4. Reflectivity [8] would be a very interesting alternative, as it features very fine-grained instrumentation: any AST node can be affected, and so it is possible to insert or modify behavior almost anywhere inside a given method. But as explained in section 8.1.2, these AST nodes intrumentations are active for every instance of the class where the AST node is located (*i.e.* where the method is defined). Enabling Reflectivity to act on single objects would open perspectives for very fine grained and very flexible adaptations.

8.3. Summary

This discussion is summarized in Tables 8 and 9. Studied technologies are compared against our requirements from section 3.6. Some of these technologies could be considered *lower-level* than other, in the sense that they could be used to implement *higher-level* solutions (*e.g.* Lub is currently implemented following a Talents inspired technique). We also differentiate programming paradigms, *e.g.* Context-Oriented Programming (*COP*) or Component-Oriented programming (*Components*), and the use of these paradigms as techniques to implement specific dynamic adaptation mechanisms.

Table 8
Requirements for fine grained unanticipated runtime adaptation.

	Granularity	Rollback	Dynamicity	Identity preservation	Flexibility
H-Graphs	Components	No	Static	Yes	Limited[a◇]
DiSL	All objects[b]	Yes	Static	Yes	Limited[c◇]
LyRT	Objects	Yes	Dynamic	Yes	Limited[d]
SMAG	Components	Yes	Dynamic	No	Limited[e]
Context-traits	Objects	Yes	Dynamic	Yes	Limited[f]
Reflectivity	Classes	Yes	Dynamic	Yes	Open[•]
Talents	Objects	Yes	Dynamic	Yes	Limited[g]
Iguana/J	Objects	Yes	Dynamic	Yes	Limited[h]
Chisel	Objects	Yes	Dynamic	Yes	Limited[i]
Components	Components	Yes[⋆]	Dynamic[⋆]	Yes[⋆]	Restricted[j]
Dynamic aspects	Objects	Yes[⋆]	Dynamic	Yes[⋆]	Open[•]
COP	Objects	No	Static	No	Restricted[k]
Lub	Objects	Yes	Dynamic	Yes	Open[•]

[a] Suited for component-based models.
[b] Can restrict intrumentations to specific constructs (*e.g.* loops).
[c] Instrumentation must be present in the base program before runtime.
[d] The base program must use the framework objects for adaptation.
[e] Components are implemented with a domain specific language.
[f] Traits can require glue code to provide adaptations.
[g] Adaptations must be written from scratch each time.
[h] Developer has to write non-functional code.
[i] Adaptations are meta-objects classes and must not conflict with each other.
[j] Adaptations and adapted entities are component-based.
[k] Adaptations specifications are defined in classes or are part of the program.
[◇] Can be used to observe a running system but not to change its behaviors.
[⋆] Only solutions matching the criteria are retained, but other do exist.
[•] Adaptation can be postponed until runtime without additional constraints on software design.

Table 9
Requirements for practical unanticipated runtime adaptation.

	Modularity	Offline composability	Paradigm shift	Intrusive adaptations
H-Graphs	No	Yes	Yes	Yes
DiSL	No	Yes[•]	Yes	Yes
LyRT	No[a]	Yes	Yes[b]	No
SMAG	No[c]	Yes	Yes	Yes
Context-traits	Yes	Yes[•]	Yes	No
Reflectivity	Yes	No	No	No
Talents	Yes	Yes[•]	No	No
Iguana/J	Yes[d]	No[e]	No	No
Chisel	Yes[f]	Yes[•]	Yes	No
Components	No	Yes[⋆]	Yes	Yes
Dynamic aspects	Yes[⋆]	Yes[⋆]	Yes	No[⋆]
COP	No	Yes	Yes	Yes[g]
Lub	Yes	Yes[h]	No	No

[a] The main program must contain instructions to retrieve the framework's objects.
[b] Not object oriented but adaptations are modeled as regular classes.
[c] The original application design must include the pre-loaded solution.
[d] Relies on a java virtual machine extension which must also be loadable at runtime.
[e] Composition can provoke conflicts.
[f] Requires loading the Iguana/J library.
[g] Adaptations are fully part of the original design of the application.
[h] Adaptation composition can be made offline, and is external to the mechanism itself.
[•] Conflicts must still be resolved explicitly.
[⋆] Only solutions matching the criteria are retained, but other do exist.

9. Future works

Lub describes fine-grained adaptations relying on objects and classes, but it lacks the ability to relate these adaptations to contexts. The objectives are not to build a COP language, because the solution is designed to be a language extension pattern. However, as COP layers regroup all behavioral variations for a given context, we would like to define layers to group lub adaptations. As adaptations can be specified by object, that would allow to build a generalization of COP layers that could affect one, multiple or all objects from the same class. We would need a way to express and compose this object selection, and reify the layer activation to control its scope. This new layer construct could be used dynamically as a mean to define higher level adaptations in COP contexts but also, for example, in debugging contexts.

An implementation based on Reflectivity would benefit to Lub, bringing the possibility to easily refine the grain of adaptations. We could inject adaptations after, before, instead or even almost anywhere inside a method, between two lines of code and without ever touching or modifying the source code. The main problem is that Reflectivity affects all instances of a class and we would like it to be able to instrument AST nodes for single objects. This work has started and it has already shown its first results. We plan to have Reflectivity enabled on an instance basis and then start working to build a Lub backend based on it.

With another backend available, it would be interesting to investigate further performances and memory usage. Benchmarking a fully operational embedded application would provide more detailed runtime information of Lub adapted programs, and could help identify drawbacks and necessary implementation enhancement of such adaptations.

We would finally like to experiment Lub on a debugging use-case, for example on robots. First we would have to study how to practically identify which are the objects to adapt, how to inject adaptations in the running system and to verify them. Investigating tool support is also a major concern, as the complexity of running systems could make impossible to manage fine grained adaptations by hand. These concerns have been left apart in this paper, as they were out of our primary scope of research.

10. Conclusion

We introduced Lub, a pattern for dynamic unanticipated behavior adaptation in object-oriented software. We specified our constraints for behavior adaptation at runtime and how Lub addresses them. We described Lub's design and how it integrates in object-oriented languages.

We followed the Lub pattern to produce a featherlight extension of Python and of the Pharo language. This latter implementation has been evaluated, and this evaluation is twofold:

- An evaluation of the impact (lookup speed, migration, impact on memory) of Lub adaptations at runtime, which shows that in simple cases the cost of adaptation is not null. It could affect systems with limited resources in which only surgical adaptation could be possible. The responsibility of choosing to bring unanticipated adaptations with Lub ultimately falls to the developer. One must decide if modifications of a running system with such costs could be affordable for the possibilities it brings.
- A simple use-case of a simulation in which a fleet of drones is adapted. The evaluation shows how Lub adaptations can provide behavioral variations and how they can be dynamically applied to the simulation. Adaptations were used to observe the running software and to adapt its behavior to context changes.

We discussed Lub with regard to the current literature and emphasized its benefits and limitations. Its strengths lie in its very fine grained adaptation capabilities and the possibility to design and trigger them at runtime.

We concluded this paper by drawing lines for further research. We would like to study implementation matters, such as other backends for Lub. We are also interested in contextualizing adaptations, through a generalized layer mechanism suited for unanticipated adaptation. Finally, we plan to experiment and evaluate Lub on devices like robots or drones.

References

[1] G. Kniesel, J. Noppen, T. Mens, J. Buckley, Unanticipated software evolution, in: European Conference on Object-Oriented Programming, Springer, 2002, pp. 92–106.
[2] K. Da, M. Dalmau, P. Roose, A survey of adaptation systems, Int. J. Internet Distrib. Comput. Syst. 2 (1) (2011) 1–18.
[3] P. Ebraert, T. D'Hondt, Y. Vandewoude, Y. Berbers, Pitfalls in unanticipated dynamic software evolution, in: RAM-SE, 2005, pp. 41–50, Citeseer.
[4] D. Weyns, Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges.
[5] M. Salehie, L. Tahvildari, Self-adaptive software: landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) 14.
[6] L. Pina, M. Hicks, Rubah: efficient, general-purpose dynamic software updating for Java, in: HotSWUp, 2013.
[7] E. Wernli, D. Gurtner, O. Nierstrasz, Using first-class contexts to realize dynamic software updates, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '11, ACM, New York, NY, USA, 2011, pp. 2:1–2:11, http://doi.acm.org/10.1145/2166929.2166931.
[8] D. Röthlisberger, M. Denker, É. Tanter, Unanticipated partial behavioral reflection, in: International Smalltalk Conference, Springer, 2006, pp. 47–65.
[9] J. Keeney, Completely Unanticipated Dynamic Adaptation of Software, Ph.D. thesis, University of Dublin, 2004.
[10] A. Goldberg, D. Robson, Smalltalk-80: The Language and Its Implementation, Addison–Wesley Longman Publishing Co., Inc., 1983.
[11] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009, http://pharobyexample.org.
[12] Python 3.4.5 documentation, https://docs.python.org/3.4/. (Accessed 6 February 2017).
[13] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, L. Renggli, Talents: an environment for dynamically composing units of reuse, Softw. Pract. Exp. 44 (4) (2014) 413–432.
[14] S. Ducasse, Evaluating message passing control techniques in smalltalk, J. Object-Oriented Program. 12 (1999) 39–50.
[15] N. Papoulias, Remote Debugging and Reflection in Resource Constrained Devices, Ph.D. thesis, Université des Sciences et Technologie de Lille-Lille I, 2013.
[16] A. Bergel, D. Cassou, S. Ducasse, J. Laval Deep Into Pharo, Lulu.com, 2013.
[17] A. Beugnard, Oo languages late-binding signature, in: The Ninth International Workshop on Foundations of Object-Oriented Languages, 2002, Citeseer.
[18] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: European Conference on Object-Oriented Programming, Springer, 2002, pp. 205–230.
[19] O. Nierstrasz, M. Denker, L. Renggli, Model-centric, context-aware software adaptation, in: Software Engineering for Self-Adaptive Systems, Springer, 2009, pp. 128–145.
[20] T. Würthinger, C. Wimmer, L. Stadler, Unrestricted and safe dynamic code evolution for java, Sci. Comput. Program. 78 (5) (2013) 481–498.

[21] A. Rosà, Y. Zheng, H. Sun, O. Javed, W. Binder, Adaptable runtime monitoring for the java virtual machine, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2016, pp. 531–546.
[22] T. Gjerlufsen, M. Ingstrup, J.W. Olsen, Mirrors of meaning: supporting inspectable runtime models, Computer 42 (10) (2009) 61–68.
[23] N. Taing, T. Springer, N. Cardozo, A. Schill, A dynamic instance binding mechanism supporting run-time variability of role-based software systems, in: Companion Proceedings of the 15th International Conference on Modularity, Modularity Companion 2016, ACM, New York, NY, USA, 2016, pp. 137–142, http://doi.acm.org/10.1145/2892664.2892687.
[24] N. Taing, M. Wutzler, T. Springer, N. Cardozo, A. Schill, Consistent unanticipated adaptation for context-dependent applications, in: Proceedings of the 8th International Workshop on Context-Oriented Programming, COP'16, ACM, New York, NY, USA, 2016, pp. 33–38, http://doi.acm.org/10.1145/2951965.2951966.
[25] C. Piechnick, S. Richly, S. Götz, C. Wilke, U. Aßmann, Using role-based composition to support unanticipated, dynamic adaptation-smart application grids, in: Proceedings of ADAPTIVE, 2012, pp. 93–102.
[26] S. González, K. Mens, M. Colacioiu, W. Cazzola, Context traits: dynamic behaviour adaptation through run-time trait recomposition, in: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development, AOSD '13, ACM, New York, NY, USA, 2013, pp. 209–220, http://doi.acm.org/10.1145/2451436.2451461.
[27] N. Schärli, S. Ducasse, O. Nierstrasz, A.P. Black, Traits: composable units of behaviour, in: European Conference on Object-Oriented Programming, Springer, 2003, pp. 248–274.
[28] J. Keeney, V. Cahill, Chisel: a policy-driven, context-aware, dynamic adaptation framework, in: IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003. Proceedings, POLICY 2003, 2003, pp. 3–14.
[29] P.-C. David, T. Ledoux, Towards a Framework for Self-adaptive Component-Based Applications, Springer Berlin Heidelberg, Berlin/Heidelberg, 2003, pp. 1–14, http://dx.doi.org/10.1007/978-3-540-40010-3_1.
[30] Y. Vandewoude, Y. Berbers, Supporting run-time evolution in seescoa, J. Integr. Des. Process Sci. 8 (1) (2004) 77–89.
[31] Y. Vandewoude, Y. Berbers, Component state mapping for runtime evolution, in: PLC, 2005, pp. 230–236.
[32] D. Preuveneers, Y. Vandewoude, P. Rigole, D. Ayed, Y. Berbers, Context-Aware Adaptation for Component-Based Pervasive Computing Systems, 2006.
[33] M. Hammer, A. Knapp, Correct execution of reconfiguration for stateful components, Electron. Notes Theor. Comput. Sci. 260 (2010) 91–108.
[34] F. Adaili, O. Mosbahi, M. Khalgui, S. Bouzefrane, Ra2dl: new flexible solution for adaptive aadl-based control components, in: 5th International Conference on Pervasive and Embedded Computing and Communication Systems, 2015.
[35] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, J. Object Technol. 7 (3) (2017).
[36] G. Salvaneschi, C. Ghezzi, M. Pradella, Context-oriented programming: a software engineering perspective, J. Syst. Softw. 85 (8) (2012) 1801–1817.
[37] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: an overview of contextl, in: Proceedings of the 2005 Symposium on Dynamic Languages, DLS '05, ACM, New York, NY, USA, 2005, pp. 1–10, http://doi.acm.org/10.1145/1146841.1146842.
[38] R. Hirschfeld, P. Costanza, M. Haupt, An introduction to context-oriented programming with contexts, in: Generative and Transformational Techniques in Software Engineering II, Springer, 2008, pp. 396–407.
[39] M. Appeltauer, R. Hirschfeld, M. Haupt, H. Masuhara, Contextj: context-oriented programming with Java, Inf. Media Technol. 6 (2) (2011) 399–419, http://dx.doi.org/10.11185/imt.6.399.
[40] T. Kamina, T. Aotani, 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, ACM, New York, NY, USA, 2011, pp. 253–264, http://doi.acm.org/10.1145/1960275.1960305.
[41] M. Von Löwis, M. Denker, 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, ACM, 2007, pp. 143–156.
[42] T. Tamai, N. Ubayashi, R. Ichiyama, An adaptive object model with dynamic role binding, in: Proceedings of the 27th International Conference on Software Engineering, ACM, 2005, pp. 166–175.
[43] T. Kamina, T. Tamai, Towards safe and flexible object adaptation, in: International Workshop on Context-Oriented Programming, ACM, 2009, p. 4.
[44] T. Kamina, T. Aotani, H. Masuhara, Generalized layer activation mechanism for context-oriented programming, in: Transactions on Modularity and Composition I, Springer, 2016, pp. 123–166.
[45] G.T. Sullivan, Aspect-oriented programming using reflection and metaobject protocols, Commun. ACM 44 (10) (2001) 95–97.
[46] A. Popovici, T. Gross, G. Alonso, Dynamic weaving for aspect-oriented programming, in: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, ACM, 2002, pp. 141–147.
[47] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, Virtual machine support for dynamic join points, in: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, ACM, 2004, pp. 83–92.
[48] R. Hirschfeld, Aspects—aspect-oriented programming with squeak, in: Objects, Components, Architectures, Services, and Applications for a Networked World, 2003, pp. 216–232.