Pymoult : On-Line Updates for Python Programs

Sébastien Martinez and Fabien Dagnat
IRISA, Télécom Bretagne
Brest, France
Email: first.last@telecom-bretagne.eu

Jérémy Buisson
IRISA, Écoles de Saint-Cyr Coëtquidan
Guer, France
Email: jeremy.buisson@irisa.fr

***Abstract*—On-line updates have proved to be essential for critical long running applications that hardly can be stopped. Indeed, security patches or feature enhancements need to be applied frequently. Pymoult is a platform allowing on-line updates for Python programs. It provides many mechanisms from the literature for updating running programs without requiring them to be stopped, allowing update developers to combine and configure the mechanisms for each update. This paper presents the design of Pymoult and details the implementation of several mechanisms it provides. With the help of an example, this paper also presents how mechanisms can be combined and configured to design on-line updates with Pymoult.**

Keywords–On-line updates; Python; Software maintenance

I. Introduction

Today's world expects software systems to be available at every moment, whether the system provides critical services like airport traffic control or whether its downtime would cause user discomfort like an operating system forcing a reboot for updating. Updating running software systems becomes a critical issue as it requires the system to be restarted, causing downtime and loss of state as well as financiary losses [1]. Not applying updates or postponing them is dangerous, as updates are necessary to keep software safe from bugs and security breaches. Dynamic Software Updating (DSU) allows updates to be applied on running software without requiring it to be restarted, causing little service disruption and no loss of data. This goal is reached by using DSU mechanisms for modifying the control flow (redefining functions) and the data flow (converting the data to a new version) of a given program. The majority of DSU platforms gather a predetermined set of these mechanisms they use to apply each update.

A lot of platforms have been proposed [2], [3], defining several mechanisms. Each mechanism has different properties and constraints. A DSU platform selects the best suited mechanisms for the type of program it targets and the kind of updates it expects. For example, K42 [4] is an operating system embedding its own DSU system. It handles its updates by swapping modified components when all old threads running out of date code are terminated. These mechanisms are best suited to the design of K42, which has a component based architecture and runs short lived threads. Updates often consist in the modification of components and, because the threads are short-lived, waiting for old threads to terminate is an easy way to ensure that components are swapped when they are quiescent. But when applying an unforeseen kind of update, the fixed set of mechanisms provided by the DSU system might be inefficient or even it may be impossible. For example, K42 does not handle API changes very well because they need to apply changes across the components.

Pymoult is a DSU platform providing several DSU mechanisms for updating Python programs. Its approach is to let an update developer select and configure the DSU mechanisms best suited for its update. While it requires more work from update developers than automated DSU platforms, it ensures that every update can be applied with best suited mechanisms. This paper presents the design and implementation of Pymoult. Section II discusses the implementation of DSU mechanisms in Python and presents Pypy-dsu, our custom Pypy interpreter enhanced for DSU support. Section III details the design of Pymoult and discusses the implementation of some of the mechanisms it provides before presenting an example of dynamic update using Pymoult in section IV. Section V compares Pymoult to other DSU platforms and Section VI introduces future work before concluding this paper.

II. Python and on-line updates

While many DSU mechanisms can be implemented in Python, some of them are impossible to develop using the standard implementation. For that reason, Pymoult uses Pypy-dsu, a Python interpreter enhanced with DSU features.

A. DSU capabilities of bare Python

Python is a dynamically typed, interpreted, object-oriented language. It has natural indirection and allows dynamic manipulation of programs models. The flexibility of Python and its introspection features make it easy to implement DSU mechanisms. For example, object fields, class methods, variables and functions are treated the same way, they are manipulated directly through their name. This allows, for example, to easily redefine a function `foo` by calling `foo=foo_v2` since each call to `foo` resolves the function name.

Fields can be added or deleted from objects and classes, allowing easy modification of objects or classes. The type of an object is kept as a `__class__` field which refers to that type. By consequence, changing the type of an object corresponds to changing the class the `__class__` field refers to and adding or deleting fields to conform the object to its new type.

Thanks to the meta-object protocol embedded in Python, Pymoult can implement a lazy method for updating objects. In Python, attributes and methods of objects are accessed (for writing, reading or calling) using `__getattribute__` and `__setattr__` methods of their class. By default, these methods resolve to the implementation in the `object` class. By overriding these methods for a given class, we can run updating code on an object before accessing its fields. Objects can therefore be updated only when actually used (i.e., when one of their fields is accessed).

Python is also a uni-typed language, allowing variables to change type dynamically without requiring specific tools. The type checking uses duck-typing. For example, if `a.foo` is called, the type of object `a` is checked for a method called `foo`. Variables can therefore be modified freely except for the deletion of fields used in the program.

 ISBN: 978-1-61208-438-1

B. The Pypy-dsu interpreter

Several DSU mechanisms can not be properly implemented in Python. For example, in a standard Python interpreter, it is not possible for a thread to suspend another one. This inability is a problem when needing to suspend parts of a program. We therefore decided to extend the features of a Python interpreter. We chose to base ourselves on the Pypy interpreter, a Python interpreter written in Python. Pypy is easier to modify than CPython (the reference Python interpreter) and already extends Python with object proxies and continuations that were helpful when implementing DSU mechanisms. This subsection presents new features that where added in Pypy-dsu, our customized Pypy interpreter.

1) Traces for controlling threads: Suspending a thread is implemented using traces. Python traces are functions called after each statement. To suspend a thread, a trace waiting for an event to be triggered is inserted. On the next statement, the trace will block until the event is triggered, causing the thread to be suspended. In Pypy, a trace cannot be set for a given thread and traces only start on the next call to a function. In Pypy-dsu it is possible to set a trace for a given thread using `sys.settrace_for_thread`. When setting the trace, one can choose whether the trace should start immediately or on the next function call. This feature allowed the development of mechanism to suspend and resume thread and control their execution (see paragraph II-B3 for an example).

2) Intercepting object creation: Although Pypy provides a garbage collector, it cannot be used to get a reference on every object created since the starting of the program. Such feature is essential when implementing mechanisms to update the data of a program. We therefore added the possibility to setup a global hook with `set_instance_hook` that is called each time an object is created. We use that hook to maintain a pool of weak references to each object created by the program. Each time an object is created, a hook creating a weak reference to it and adding it to the pool is called. This pool is used each time a mechanism requires accessing all the data at a same time.

3) Dropping frames: It is not possible in Python to manipulate the stack of a thread, making it impossible to support on stack replacement of functions. We added new instructions to drop frames from the stack. Calling a `dropNframe value` statement will cause the N most recent frames to exit immediately, returning `value`. On stack replacement of a function by a new one is implemented using traces and the `drop2frames` function to force the two most recent frames to exit and return value. A trace calling the new function before using `drop2frames` is inserted in the target thread. When the thread enters a frame running the old function, the trace captures the local state of that frame and calls the new function, giving that state as an argument. The return value of the new function is then given as argument to `drop2frames`. The last two frames (i.e., the frame of the trace and the frame of the old function) are dropped and the return value of the new function is returned to caller of the old function.

III. PYMOULT

To our knowledge, Pymoult is the first DSU platform for Python programs. Its approach is to provide as many DSU mechanisms as possible through an API that allows their combination and configuration. Since the creation of Pymoult in 2012, we implemented over 30 DSU mechanisms. For that reason and for the features we previously detailed, we think that Python is a good language for writing DSU mechanisms, and testing platforms designs. The design of Pymoult is the result of incremental work. Since the first version of Pymoult [5] and throughout the experiments we conduced with it, the design evolved to its actual form we present in this section.

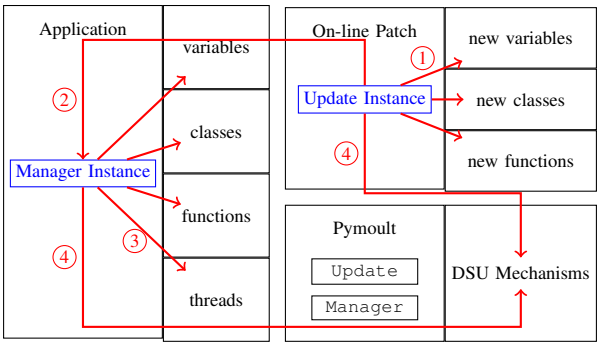

Figure 1. Map of an on-line update

To update a running program with Pymoult, the program developer must start a specific Pymoult thread called **`Listener`** in the program. That thread enables the supplying of on-line patches for the running application. An on-line patch is a piece of Python code that uses the Pymoult API. It contains the code of the updated elements of the program (e.g., functions, classes) and instructions on which DSU mechanisms to use. Dynamic updates rely on **`Manager`** and **`Update`** classes. A manager (an instance of the **`Manager`** class) is responsible for applying modifications according to the instructions given by an update object (an instance of the **`Update`** class).

Subsection III-A presents the design of Pymoult and details how to write an on-line patch with Pymoult. Subsection III-B details the implementation of some mechanisms provided by Pymoult and section IV presents the example of an on-line update using Pymoult.

A. Design

In Pymoult, an update is composed of several instances of an **`Update`** class. These instances are supplied to a manager that will apply them. For the remainder of this section, we use the term *update object* to refer to instances of **`Update`**. An *on-line patch* is therefore a set of update objects.

Figure 1 presents the architecture of a program undergoing an on-line update. An on-line patch embeds new variables, classes and functions ① which are used by an update object to specify the instructions for the manager ②. The manager controls and modifies the elements of the program ③ using DSU mechanisms provided by Pymoult and as specified by the update object ④.

Pymoult provides several off-the-shelf manager classes that can be instantiated in the program or in an on-line patch to create new managers. The regular **`Manager`** class describes a manager that operates only when the program calls its `apply_next_update` method. This kind of manager allows the program to decide when modifications can be applied to it. The **`ThreadedManager`** class describes a manager that operates in its own thread. It applies modifications each time an update object is supplied to it. Pymoult also provides preconfigured managers that are bound to an **`Update`** class and will always use the same DSU mechanisms to apply every modification. Lastly, one can extend the **`Manager`** class to

 ISBN: 978-1-61208-438-1

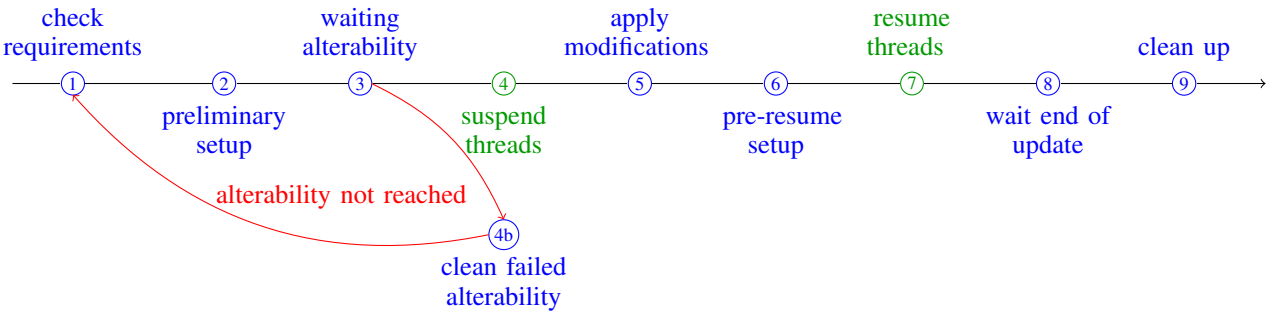

Figure 2. The updating process

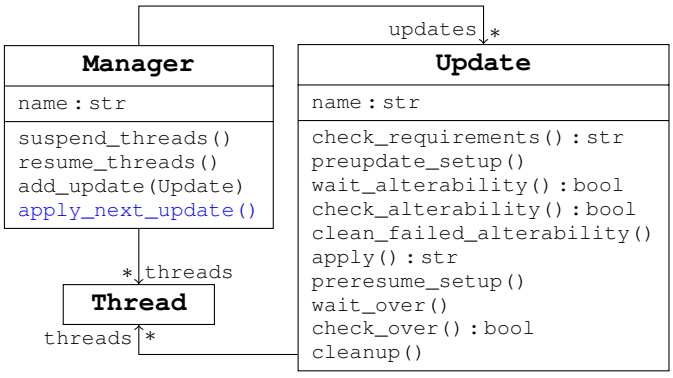

Figure 3. **Update** and **Manager** classes

define one's own manager. The classes involved in the updating process are presented in Figure 3.

Update developers can define their own update classes by extending the **Update** class. An update class has one method for each step of the updating process. These methods can use the DSU mechanisms provided by Pymoult through calls to specific functions. Update objects are instances of developer defined update classes and are supplied to managers. The managers implement the updating process pictured in Figure 2. When an update is supplied to a manager, that manager checks the requirements of the update ①. To do so, it calls the `check_requirements` method of the update that returns `"yes"`, `"no"` or `"never"` if the requirements are (respectively) met, not met or can never be met. If `"no"` is returned, the update is postponed. If `"never"` is returned, the update is canceled and if `"yes"` is returned, the updating process continues. The manager then proceeds to the preliminary setup step ② where it installs elements required for the next steps. To do so, it calls the `preupdate_setup` method of the update. When the preliminary setup is finished (i.e., the `preupdate_setup` method has returned), the manager waits for the application to be in a safe state we call *alterability* ③ by calling the `wait_alterability` method of the update. That method returns `True` when the application can be safely modified or `False` if a safe state could not be met in a fixed amount of time. If `False` is returned, the manager invokes a cleanup step ④b in which it calls the `clean_failed_alterability` method of the update for uninstalling the elements that were set up in the preliminary setup step. The update is then postponed. If `wait_alterability` returns `True`, the manager suspends some threads of the program ④ by calling its `suspend_threads` method. If the update specifies threads in its `threads` attribute, `suspend_threads` will suspend them, if not it will suspend the threads controlled by the manager (i.e., the threads in its `threads` attribute). If the manager does not control any threads, no thread is suspended. The manager then proceeds to the apply step ⑤ where it calls the `apply` method of the update. That method realizes all the modifications needed by the update (*e.g.* redefine functions, transform the data). The following step of the manager, the pre-resume setup step ⑥, calls the `preresume_setup` method of the update that follows the same principle as the `preupdate_setup` method. Suspended threads are then resumed by the `resume_threads` method of the manager ⑦. When all threads are resumed, the manager waits for the update to be over ⑧ by calling the `wait_over` method of the update that returns when the update is over. Indeed the apply step may have started tasks that run along the rest of the program. For example, the update can start lazy modifications of objects and requires all the objects to be transformed before completing. When the update is over, the manager cleans up any element installed in the preliminary setup and pre-resume steps ⑨ by calling the `cleanup` method of the update.

While this updating process is exactly followed as we just described by instances of **ThreadedManager**, instances of **Manager** wait passively for a safe state and for the end of the update. They give back the hand to the program each time they have to wait. For that purpose, they call the non-blocking `check_alterability` method (resp. `check_over`) instead of `wait_alterability` (resp. `wait_over`).

B. Mechanisms

Mechanisms are provided as functions that can be called in the methods of update classes. In this subsection, we follow the updating process detailed in the previous one and present some mechanisms that can be used for each step. Figure 4 presents an update object using the mechanisms discussed here.

1) Preliminary setup: Some mechanisms provided by Pymoult need preliminary installation before being used. This is the case of the `forceQuiescence` mechanism that forces a function to be quiescent. In the pre-update setup step, the `setupForceQuiescence` function replaces the targeted function by a stub that blocks all incoming, non-recursive calls by waiting for a specific *continue* event to be activated. A *watcher* thread is then started. That thread watches the quiescence of the targeted function.

2) Waiting alterability: We call *alterability* the state of a program when it can be updated without provoking errors. Indeed, if the update is applied at a wrong moment, updated code can call obsolete code and cause a crash. For example, an outdated function could try to access an updated piece of data that is no longer compatible with the function.

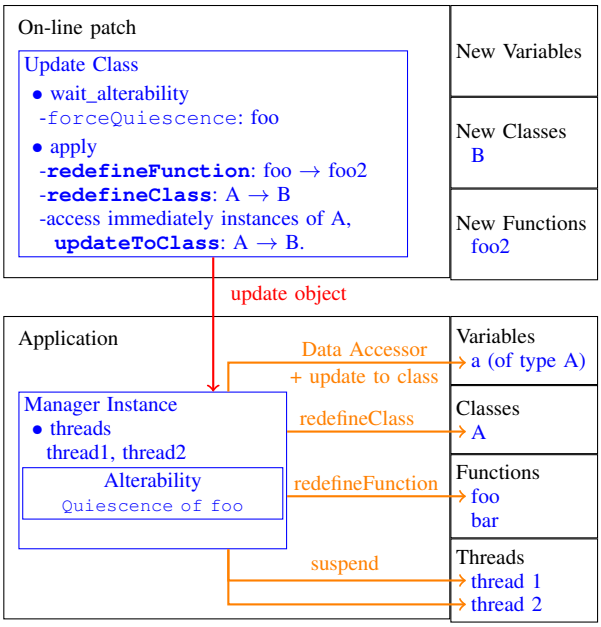

Figure 4. An example of update using Pymoult

Alterability can be detected by watching *alterability criteria* such as the *quiescence* of a component to be updated [6] or any condition on the state of the program. These criteria depend on the modifications applied by the update and may vary among all the updates. Several such criteria are proposed in the literature as the *tranquility* [7] or the *serenity* [8] of components. Pymoult provides several functions for expressing alterabilty crtieria. Here, we discuss the `waitForceQuiescence` function that expresses the criterion "target function must be quiescent" while forcing its quiescence instead of waiting for it. `waitForceQuiescence` waits for the *watcher* thread started in the previous step to detect the quiescence of the target function, then returns.

3) Applying modifications:

a) Accessing and updating data: Pymoult provides two ways to access data through the **`DataAccessor`** class that behaves as an iterator. When creating an instance of **`DataAccessor`**, one must precise the type of objects it accesses and the strategy to use as a string. The *immediate* strategy accesses all the objects when the instance of **`DataAccessor`** is created. It is then possible to iterate over all the objects. The *progressive* strategy uses the meta-object protocol described in section II-B to access objects lazily. Each time an object of the given type is used by the program, it is enqueued to the instance of **`DataAccessor`**. It is possible to iterate over the objects progressively as they are accessed. When the queue of accessed objects is empty, the iteration hangs until new objects are accessed. As a consequence, it is not possible to know a priori when all the objects have been accessed and therefore, when the iteration ends.

When they are accessed, objects can be updated using the **`updateToClass`** function. This function changes the type of a given object to a given class by updating its `__class__` attribute. A transformer supplied by the update developer is then applied to the object to modify its attributes.

b) Updating functions and classes: One way to update a function is to replace it by a new version. This mechanism is provided through the **`redefineFunction`** function that uses the native indirection of Python to change the body bound to the old function's name.

Similarly, classes can be redefined globally using **`redefineClass`** or one can add new fields or modify existing ones with **`addFieldToClass`**. In Python, classes are just special objects that can be modified dynamically as any other object.

4) Pre-resume setup: At this step, a mechanism may require some set up before resuming the excution of the program. For example, this is the case of `forceQuiescence`. In this step, the `cleanForceQuiescence` function activates the *continue* event waited by the blocking stub added during the pre-update setup. As a consequence, all the calls to the targeted function are released.

5) Cleaning failed alterabilty: DSU mechanisms handling alterability watching that required preliminary set up require clean up if the program fails to reach alterability. If the `forceQuiescence` mechanism fails to guide the program to alterability, the `cleanFailedForceQuiescence` function stops the *watcher* thread, activates the *continue* event and removes the stub installed in the pre-update setup step.

IV. An example

Various uses of Pymoult have been tested to validate it. Among the applications we have dynamically updated with Pymoult is the Django application server. Pymoult allowed us to update a running Django server from version 1.6.8 to version 1.6.10, choosing different DSU mechanisms for both successive updates (from 1.6.8 to 1.6.9 and from 1.6.9 to 1.6.10). Such a complex update does not fit as an introductory example. Instead, we present here the example of a program serving pictures through a socket. This program is representative of the Django example while staying simple.

Figure 5 presents the main elements of this program. Picture objects are stored in folders a `files` dictionary. Folders are served by the `serve_folder` method of the `ConnThread` class which defines connection handling threads. When starting, the program creates a listener to receive future on-line patches (as explained in section III). It also creates an instance of **`ThreadedManager`** and an `ObjectPool` that will contain weak references to all the created objects as explained in section II-B. That pool will enable immediate access to objects for future updates. Each time a new client connects to the server, a new `ConnThread` instance starts responding to all the commands it receives. The `do_command` method specifies the reaction to each received command.

Figure 6 presents an on-line patch that introduces support for comments. It is now possible to add a comment to pictures and before serving pictures from a folder, the pictures are annotated with their comment. The on-line patch redefines the `Picture` class and the `serve_folder` and `do_command` methods. In order to update the picture objects, the patch provides a transformer named `pic_trans`.

The `ServerUpdate` class defines a new update class which alterability criteria are the quiescence of the methods `do_command` and `serve_folder`. Because the update aims to redefine these two methods and to modify the picture objects they both use, waiting for their quiescence before updating ensures that it will not provoke errors. Before waiting

 ISBN: 978-1-61208-438-1

```
class Picture(object):
  def __init__(self,path,name):...
  def stream(self):...
class ConnThread(threading.Thread):
  def __init__(self,connection):...
  def serve_folder(self,folder):...
  def do_command(self,command):...
  def run(self):
    while self.connection:
      data = self.connection.recv(1024)
      self.do_command(data.strip())
def main():
  #create a socket to listen for commands
  while True:
    conn,addr = sock.accept()
    ConnThread(conn).start()
if __name__ == "__main__":
  listener = Listener()
  listener.start()
  manager = ThreadedManager()
  manager.start()
  ObjectsPool()
  main()
```

Figure 5. Structure of the program

for alterability, the update captures all the `ConnThread` instances and the main thread as they need to be suspended (Suspending the main thread ensures that no new `ConnThread` is created during the update). For that purpose, the patch defines the method `getAllConnThreads`. When alterability is met, the update uses **`addFieldToClass`** to redefine the methods and uses a **`DataAccessor`** to access the picture objects. It then uses **`updateToClass`** to update the accessed objects and **`redefineClass`** to redefine the `Picture` class.

The on-line patch creates an instance of `ServerUpdate` then supplies it to the manager. When the patch is sent to the listener created by the application, it is loaded in the application and its code is executed. The functions and classes it contains are defined and the update object is created and supplied to the manager.

Writing on-line patches as small programs which execution will update the targeted program allows for a fine control over the DSU mechanisms. For example, as presented in figure 7, we could have chosen to apply the update without waiting for the quiescence of `do_command` and `serve_folder` and use on-stack replacement to update theses methods while they are active. That would be a good choice if `do_command` and `serve_folder` are rarely quiescent at the same time. If the server handles a great amount of pictures, updating them all at the same time is long and disrupts the service since connections are suspended during the update. Updating picture objects lazily would be a better solution as data would be migrated without suspending connections (at the cost of the overhead introduced by the update of objects the first time they are accessed). Figure 7 presents this alternative patch for the update of the server. It uses **`rebootFunction`** to capture the state of currently running `do_command` and `serve_folder` methods then uses on-stack replacement. For that purpose, the patch defines the `command_capture` and `serve_capture` functions. The update uses **`startLazyUpdate`** to start updating picture objects lazily using the meta-object protocol described in II.

```
class Picture_V2(object):
  def __init__(self,path,name):
    ...
    self.commentary = "Witty comment"
    self.basepath = path
  def stream(self):...
  def comment(self,text):...
  def annotate(self):...
def getAllConnThreads():...
def pic_trans(pic):
  pic.basepath = pic.path
  pic.commentary = "Witty comment"
def serve_folder_v2(self,folder):...
def do_command_v2(self,command):...
class ServerUpdate(Update):
  def preupdate_setup(self):
    self.threads = getAllConnThreads()
  def wait_alterability(self):
    return waitQuiescenceOfFunctions([do_command,
                                      serve_folder])
  def apply(self):
    addFieldToClass(ConnThread,"do_command",
                    do_command_v2)
    addFieldToClass(ConnThread,"serve_folder",
                    serve_folder_v2)
    accessor = DataAccessor(Picture,"immediate")
    for picture in accessor:
      updateToClass(picture,Picture,Picture_V2,
                    pic_trans)
    redefineClass(Picture,Picture_V2)
conn_update = ServerUpdate(name="conn_update")
main.manager.add_update(conn_update)
```

Figure 6. Simplified on-line patch

```
class ServerUpdate(Update):
  def preupdate_setup(self):
    self.threads = getAllConnThreads()
  def wait_alterability(self):
    return True
  def apply(self):
    addFieldToClass(ConnThread,"do_command",
                    do_command_v2)
    addFieldToClass(ConnThread,"serve_folder",
                    serve_folder_v2)
    for thread in self.threads:
      rebootFunction(do_command,do_command_v2,
                     command_capture)
      rebootFunction(serve_folder,serve_folder_v2,
                     serve_capture)
    startLazyUpdate(Picture,PictureV2,pic_update)
    redefineClass(Picture,Picture_V2)
```

Figure 7. An alternate on-line patch (simplified)

V. Related work

To our knowledge, Pymoult is the only DSU platform for Python and its approach letting update developers combine and configure DSU mechanisms is an actual topic in the field. While classical DSU platforms use the same combination of mechanisms to apply every update, some platforms allow update developers to configure some mechanisms.

In ProteOS, [9] Giuffrida et al. propose to let update developers decide the alterability criteria for each update. The criteria are expressed as filters on the state of the OS. ProteOS allows processes to be updated by starting the new version of a process and transferring and updating the data from the old process to the new one. The data is accessed immediately using code instrumentation.

K42 [4] is an operating systems that allows its components to be swapped at runtime. When applying an on-line patch, it

 ISBN: 978-1-61208-438-1

forces all swapped components to be quiescent by suspending all threads created after the on-line update is requested and waiting for the old threads to terminate. Components are progressively swapped when they become quiescent.

Jvolve [10] is a DSU platform for Java programs. It allows classes and methods to be redefined. Update developers provide the source code of the program and of its updated version as well as class transformers (for updating static class fields) and object transformers (for updating object fields). The alterability criteria for every update is that the program must reach a VM safe point (usually a point where the garbage collector is called) where the redefined methods are quiescent. Update developers can also indicate methods whose quiescence will constitute an additional alterability criterion. When alterability is met, all threads are suspended and Jvolve updates methods using indirection at VM level and on-stack replacement. It accesses objects using the garbage collector and updates them immediately.

For these three platforms, the on-line patch supplied by the update developer is made of the source code of the new version of the program plus some instructions (K42: code of the transfomers; Jvolve and ProteOS: code of the transformers and of the alterability criteria). While Jvolve and ProteOS allow update developers to configure mechanisms by giving additional alterabilty criteria, they support little variability on the updating process. These platforms force the configuration of DSU mechanisms (*e.g* alterability criteria are quiescence of some functions) and only allow update developers to extend some of them (*e.g* by giving new functions that need to be quiescent for alterability). To our knowledge, Pymoult is the first DSU platform giving as much control on the mechanisms used for on-line updates.

VI. Conclusion and future work

We presented the design of Pymoult and presented how it allows DSU mechanisms to be combined and configured when writting an on-line patch. We also presented an example of on-line update of a Python program using Pymoult.

Pymoult is built atop a modified version of the Pypy interpreter. Because the modifications we applied to Pypy are little intrusive on the interpreter, they have no impact on the way Pypy interprets Python programs. Pymoult is therefore fully compatible with all the applications that are compatible with Pypy. Nevertheless, many common Python applications have compatibility issues with Pypy. The purpose of Pymoult was to find a design that allows DSU mechanisms to be easily configured and combined. Therefore, compatibility with every Python application was not an issue. Nonetheless, to ensure better compatibility with common Python software, we are developing a custom version of the CPython interpreter, the most used Python interpreter. Having a CPython-dsu interpreter will allow Pymoult to be tested with more real-life Python software.

Updating Django proved that Pymoult can be used to update real world software. Further experiments, such as overhead measurement, are required before validating the use of Pymoult for production software.

Pycots [11], a component model enabling architectural reconfiguration of applications, is an example of the use of Pymoult. The model is paired with a development process for specifying reconfiguration and proving their corectness using Coq before executing them using Pymoult.

The design of Pymoult is well suited for designing customized updates for Python programs. Having a similar design for different languages would be a good thing because it would allow combining DSU platforms for updating complex applications made of several programs using different languages. We are currently working on a C version of Pymoult as a means to establish an equivalent design for C programs.

Pymoult is free software published under GPL License. Its source code, as well as several examples can be found on the project repository [12]. The example presented in subsection IV is based on the "interactive" example.

Acknowledgement

The work presented in this paper is funded by Brittany regional council, as part of project IMAJD.

References

[1] Channelinsider, "Unplanned IT Outages Cost More than $5,000 per Minute: Report," http://www.channelinsider.com/c/a/Spotlight/Unplanned-IT-Outages-Cost-More-than-5000-per-Minute-Report-105393, 2011, [Online; accessed 28-September-2015].

[2] E. Miedes and F. D. Muñoz-Escoí, "A survey about dynamic software updating," Instituto Univ. Mixto Tecnológico de Informática, Universitat Politècnica de València, Tech. Rep. ITI-SIDI-2012/003, May 2012.

[3] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani, "A survey of dynamic software updating," Journal of Software: Evolution and Process, 2012. [Online]. Available: http://dx.doi.org/10.1002/smr.1556

[4] C. A. N. Soules et al., "System support for online reconfiguration," in Proc. of the Usenix Technical Conference, 2003, pp. 141–154.

[5] S. Martinez, F. Dagnat, and J. Buisson, "Prototyping DSU techniques using Python," in HotSWUp 2013 : 5th Workshop on Hot Topics in Software Upgrades, USENIX, Ed., 2013.

[6] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," IEEE Trans. Softw. Eng., vol. 16, no. 11, Nov. 1990, pp. 1293–1306. [Online]. Available: http://dx.doi.org/10.1109/32.60317

[7] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, "Dynamic software updating using a relaxed consistency model," IEEE Transactions on Software Engineering, vol. 37, no. 5, 2011, pp. 679–694.

[8] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi, "An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems," in Proc of the 15th Symposium on Component Based Software Engineering, ser. CBSE. New York, USA: ACM, 2012, pp. 177–182. [Online]. Available: http://doi.acm.org/10.1145/2304736.2304765

[9] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in Proc of the International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS. New York, USA: ACM, 2013, pp. 279–292. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451147

[10] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A vm-centric approach," in Proc of the Conference on Programming Language Design and Implementation, ser. PLDI. New York, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542478

[11] J. Buisson, E. Calvacante, F. Dagnat, S. Martinez, and E. Leroux, "Coqcots & Pycots: non-stopping components for safe dynamic reconfiguration," in Proc of the 17th Symposium on Component-Based Software Engineering, ser. CBSE, ACM, Ed., New York, USA, 2014, pp. 85 – 90.

[12] S. Martinez, J. Buisson, F. Dagnat, A. Saric, D. Gilly, and A. Manoury, "Pymoult," https://bitbucket.org/smartinezgd/pymoult, 2008, [Online; accessed 28-September-2015].

 ISBN: 978-1-61208-438-1