

Programming Language Support for Dynamic Software Evolution

Alessandro Margara
margara@elet.polimi.it

Liliana Pasquale
pasquale@elet.polimi.it

Alessandro Sivieri
alessandro.sivieri@mail.polimi.it

1. INTRODUCTION

Software systems are affected by continuous changes. Despite changes are necessary to maintain a software product (e.g. for bug removal), include new required features or improve existing ones, they may lower software quality, increasing its complexity. Informally, this is the message stated in the *Laws of Software Evolution* [27], which studies the process of software updating that follows the initial development of a system [28]. To reduce the negative impact that a change introduces to software quality, solutions are provided at different levels, from the architectural level to the programming languages level.

Software evolution becomes particularly interesting when applied to computer programs that must run continuously and with no interruption. In many fields (e.g., financial transaction processors, telephone switches, reservation services, etc.) run-time software changes are addressed adopting redundant hardware to enable *hot standby*. This approach consists in selectively performing upgrades on a set of machines that are taken temporarily down, while the remaining part of the machines continuously provide required services. Besides the obvious drawback in term of costs, this solution is practically infeasible in a wide range of scenarios, in which hardware is a limited resource. Such scenarios are increasingly emerging in the era of the so called *Internet of Things*, in which software is deployed inside a wide range of heterogeneous devices, including PDAs, sensors, etc. In this context devices usually need to change their behavior dynamically to adapt to the different situations in which they may operate.

These examples motivate the need for software evolution during program execution: this feature is usually referred to as dynamic software evolution [23]. In the following we analyze this research field from the programming languages perspective. In particular we focus on the support offered by object oriented programming (OOP), functional programming and aspect oriented programming (AOP) languages. We start from languages analysis to extract the more suitable features able to support dynamic changes. In particular, during our review, we try to answer the following questions:

- *Which types of changes can be addressed?*

We do not provide a unified framework to classify software changes (as in [15, 16, 14]), but we focus on the limitations posed by a given language on the type of allowed changes. For example some languages are not suitable to manage distributed applications, while others do not have explicit functionalities to update

existing state etc.

- *Which mechanisms are used to enable dynamic changes?*
We focus on the low level features provided by the language and its environment (compiler etc.) that somehow make runtime changes possible.
- *Which language abstractions are defined to ease software changes?*
We focus on the features embedded inside a language or tool that simplify the design and implementation of applications that include the possibility to change software at runtime.

The rest of the paper is organized as follows. Section 2 analyzes the features provided by object oriented languages to support dynamic changes. Section 3 explains how dynamic updates are addressed in some functional programming languages and Section 4 illustrates dynamic AOP techniques to perform runtime software evolution. Finally, Section 5 concludes the paper discussing the advantages and disadvantages of presented approaches.

2. THE OBJECT ORIENTED PERSPECTIVE

In this Section we consider two of the most adopted OOP languages: Java and C#. Both their platforms already contain implementations of useful technologies to achieve runtime evolution. More enhanced features will also be included in the next versions (respectively Java 7 and C# 4).

Object oriented paradigm itself tries to force a developer to write modular code, through the use of interfaces and abstract types, packages and method overriding. If these features are correctly followed, then the application can naturally evolve and be extended, without losing compatibility with the other modules (components) with which it cooperates. Of course, this paradigm is completely independent on the type of evolution itself, be it static or dynamic.

The following paragraphs introduce three technologies that can be adopted to support runtime evolution: *HotSwap*, *Compiler API* and *Dynamic programming*. It is worth to note that all these technologies only provide low level mechanisms and do not represent real language abstractions. The first two technologies involve a compilation step, so they apply a static check on newly introduced code versions. On the contrary, the last one does not involve compilation, but exploits features of loosely typed languages. This way dynamic programming offers less guarantees of compatibility between new and old versions, and this may bring an application to runtime exceptions if type constraints are violated.

2.1 HotSwap

HotSwap is a technology that allows developers to alter the code of a running application without having to stop its execution. The same name is also used in other fields: for example in hardware development this characteristic may be offered for hot removal of hard drives. Dmitriev [17], proposes a staged description to classify the capabilities offered by different HotSwap implementations. The work is related to the *HotSpot JVM*, but it can be also used as a comparison metric with the *Common Language Runtime (CLR)*, the counterpart of *Java Virtual Machine* for the .NET Platform. It comprehends the following steps:

1. *Stage 1*: changes of method bodies.
2. *Stage 2*: binary compatible changes.
3. *Stage 3*: arbitrary changes, except those involving instance formats.
4. *Stage 4*: arbitrary changes.

A new version is binary compatible with the previous one if it can be compiled and linked to the rest of the program without incurring in compile time errors. Hence binary compatible changes keep the running program in a consistent state (e.g., adding a method is binary compatible, while removing a method is not). Changes on the instance format involve adding, removing or modifying fields in a class.

2.1.1 Java

JVM HotSpot contains two different libraries that support HotSwap. The first one is called *Java Platform Debugger Architecture (JPDA)* [5]. Since version 1.4.2, it implements the stage 1 of the above mentioned metric. It provides low-level facilities to create end-user debugging tools, that enable on-the-fly bug correction, including the possibility to suspend a computation, change the body of a method and resume the computation to test modification's effects.

Since version 1.5, a new inspection interface has been introduced, called *JVM Tool Interface (JVMTI)* [6]. Among provided features it permits to monitor threads (e.g. to get their state, or suspend/resume/stop/interrupt their execution) or control stack/heap status (e.g. to get the full stack of a thread, or pop some frames). Moreover it allows to allocate/deallocate memory space, force a early return of a method at any point during its execution, or read and modify classes' bytecode. The last capability, in particular, allows the replacement of a class definition. In fact the old method declarations become obsolete, and are used only if they still have active stack frames, while new method's invocations refer to the new definition.

To introduce a real example, the JVMTI library is the underlying API used by Eclipse to enable developers to monitor their Java applications, collect and analyze performance data, create and inject monitoring probes, profile specific information and apply reverse engineering to specific parts of the program. A limitation of this approach is that a new class implementation can only change method bodies, while the constant pool and the attributes have to remain unaltered. Hence changes that add new methods, modify a method's signature, or alter the inheritance tree are not allowed. Both JPDA and JVMTI can be placed at stage 1, according to the classification shown above.

The impact this limitation has on real use cases has been analyzed in [20]. In this paper the authors classify 92 changes made to an open source web server, *Jetty*, during several minor releases. Most of the changes involved bug removal or minor improvements, and they were detected from the analysis of the Version Control System log. The authors detect the changes that is possible to perform according to each stage of the classification, obtaining the following results:

- stage 1 handles 37% of changes, and this because most bugs can be solved only with method body changes;
- stage 4 handles 33% of changes, and this is caused by the need, in large changes, of introducing new class fields or attributes;
- stages 2-3 handle minor percentage of changes;
- 14% of modifications cannot be handled by any of these stages, since they may require distributed coordination among the parties involved in a change or some information not available yet during the analysis.

However, since stages 2-4 are not supported by JPDA, they are considered only from a theoretical point of view.

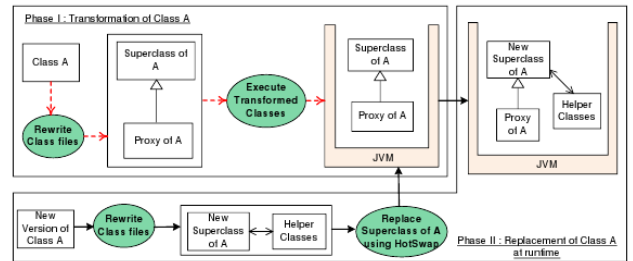


Figure 1: Binary rewriting procedure.

An interesting approach that overcomes JPDA's limitations is called *binary rewriting* [26]. It relies on JVMTI and uses binary refactoring for introducing indirect reference to the application, through the Proxy pattern. To perform class evolution two steps are required. First, the original class becomes a proxy class which inherits from a *Virtual Superclass*, and contains all the original functionality. Furthermore, client classes refer to the proxy class instead of the original one. This step is easily supported by the JVM compiler which is also able to inline delegate method calls, if they do not involve dynamic dispatch, eliminating indirection overhead. In the second step a library calculates the differences between the old and the new source, including them in a new helper class, which is added to the application. Each call to a new method is then translated, by the proxy class, to a call to the helper class, instead. The proxy class and the helper class are loaded by the HotSwap API at runtime, without incurring in update errors.

2.1.2 Java serialization

Java 1.6 introduced also a new feature, similar to the HotSwap approach but developed to allow the introduction of compatible changes in serialization mechanisms [4]; compatible changes are essentially addition of fields and addition or removal of classes.

The new serialization library allows developers to evolve

their code without having to worry about implementing different serialization methods, one for each class version; the correct one is automatically identified and loaded, and the overhead for the nonversioned code is kept to a minimum.

2.1.3 C#

C# offers HotSwap mechanisms through the following APIs:

- *CLR Profiler API*, which can be used for collecting data at runtime, and for getting information, through a callback mechanism, of module load/unload, method entry/exit and compilation invocations.
- *Unmanaged Metadata API*, which is a low-level interface for reading and writing metadata, that are description of types, like method signatures or class attributes, and also internal implementation details, involving bytecode addresses and pointers.

These libraries can be located at stage 3 of our classification. However C# platform does not offer high-level tools to use these APIs, with the only exception of Microsoft Visual Studio debugger. To overcome this limitation, a unified framework is proposed in [34]. This framework uses method shadows, that are very similar to the Proxy classes adopted in Java to perform binary rewriting. This approach manages modifications changing the Common Intermediate Language (CIL) bytecode of the modified classes and, at the level of bytecode addresses, makes the new references to those classes point to the new bytecode version. As a final step, a just-in-time compilation is run to produce native assembly instructions.

2.2 Compiler API

The creation of a high-level library for accessing compilation capabilities directly from the code, in a way officially supported by the language, allows developers to compile code on-the-fly. Moreover the combination with dynamic code loading capabilities (see Section 2.3) enables partial support for dynamic evolution.

Java. Sun's JVM contains, since version 1.6, a new package, *javax.tools*, which implements the Java Specification Request number 199. It allows source code compilation and offers a few diagnostic services, for getting compilation warnings or errors.

C#. Mono implementation of C# specifications has already introduced an API for interfacing directly to the compiler [8]. Its capabilities are reduced, because it is mostly an evaluator, and it can compute only statements and expressions, thus not allowing complete class compilation. The Microsoft .NET implementation is introducing similar characteristics in its next major version.

2.3 Dynamic programming

To overcome some of the limitations described above, both Java and C# enable programmers to directly interact with external dynamic programming languages. Such languages are usually not compiled, but interpreted on the fly, during computation. This feature enables programmers to change code during runtime and consequently to provide partial support for software evolution. Obviously, this support has a limited scope, as it only affects the parts of the application delegated to external dynamic languages. Despite the

use of dynamic programming language enables runtime code change, on the other hand it reduces the possibility to perform compile time type checking.

To enable the interaction with dynamic languages, two main complementary features are introduced. The first one is a development API, which can be used to load a dynamic language interpreter and execute a file or a string containing code written in that specific language. This API also allows passing object instances between the host and the guest language and invoking a library function from one language to the other.

As of today, both Java and C# can execute code written in Python, Ruby or Javascript [22, 7], and also in dynamic languages built on top of one of those two platforms, like Groovy (a new Java dialect); for each of these, an interpreter has been written in the host language, and the scripting code is passed to that interpreter, also called *engine*.

The second required characteristic is the development of a new language runtime, which offers new common services for simplifying the development of different engines, and easing the interaction between host and guest languages. According to this requirement, C# is developing its *Dynamic Language Runtime* [3], born with IronPython, which will be included in .NET 4.0, while Java is developing the *Da Vinci Machine* [2], with the same aims of the DLR, to be included in Java 1.7.

Java. Java already offers, from version 1.6, the Scripting API (JSR 223), which has a *ScriptEngine* class for creating a specific language engine, and an *eval()* method for evaluating an expression or a script file. In Java 1.7, the JSR 292 will be added, with the inclusion of a new low-level instruction, called *invokedynamic*, which will be used in conjunction of a new bookmark interface, *java.dyn.Dynamic*, for deferring a type binding from compile time to runtime. This feature will enable code compilation even if a return type is not known in a first place, but it will become known only when the dynamic script will be executed.

C#. Dynamic programming has been introduced during the last Microsoft Professional Developers Conference (held in 2008), by one of the language architects, Anders Hejlsberg [22]. As said above, the dynamic language runtime will be used for bringing some services on top of CLR. These services are necessary to enable the execution of code written in dynamic languages. In addition, C# 4.0 will introduce the possibility to declare variables as *dynamic*, thus postponing the actual resolution of the real type from compilation time to run time, using a dynamic binder. This way, the interaction of some C# code with a script written, for example, in Python, will be possible, incorporating the lack of a static type, typical of the latter, with the compilation needs of the former.

3. THE FUNCTIONAL PROGRAMMING PERSPECTIVE

3.1 Erlang

Erlang [10] is a functional concurrent programming language. It is *functional* in the sense that computation is performed by means of (usually recursive) mathematical function evaluation; state and mutable data are avoided. Functions are first-class values and can be passed as parameters or received as results of *higher-order* functions. It is

a concurrent language in the sense that it offers *good support* for concurrency; in particular it is able to manage a large number of processes, while creation and destruction of processes, as well as inter-process communication are efficient operations. Additionally processes share no data and always operate as if they ran on physically separated processors; message passing is the only allowed communication pattern [11].

Erlang is typeless in the same sense as traditional logic languages; it uses pattern matching for variable binding and function selection, it has explicit mechanisms to create concurrent processes and advanced facilities for (distributed) error detection and recovery [12, 37].

3.1.1 Dynamic Code Loading

Dynamic code loading is a feature directly built inside Erlang. In Erlang, code is organized in separated units, called *modules*; each module has two ways to access functionalities exposed by other modules: import them or refer to them using fully qualified names in the form *module-name:function-name*. The default policy for code loading is simple: every time a function is called using its fully qualified name the latest version of the function in the latest version of the module is called, even if the module has been recompiled while the code was running it. This way it is also possible to have two different versions of a single function running side by side inside a single program; the main technical limitation of this approach is that, at a given time, only two versions of a module can exist: the *current* one and the *old* one. If a third version of a module is created, every computation that was still using the first version is immediately aborted: then version 3 becomes the *current* version and version 2 the *old* version.

From this point of view the support for updating code is extremely powerful: it is possible to change virtually everything inside a module and compile and load a new version in every moment during computation, the only limitation being represented by the concurrent presence of only two versions of each module.

3.1.2 Behaviors

The aforementioned features for dynamic code loading make run-time software changes feasible, but do not offer facilities to ease updates management.

However, the functional programming paradigm in general, and Erlang in particular, allow developers to implement programs following well known design patterns, whose aim is usually that of reaching a high degree of modularization, so that two important goals are met: first, the different aspects of the problem a program has to deal with are split into different modules; second, functions are usually organized in abstraction layers that simplify reuse of code. One benefit that derives from such modularization is the possibility to emphasize the distinction between *functional* modules, i.e. all those modules that directly implement the functionality a system is designed for, and *non-functional* modules, i.e. all those modules that implement solutions to meet additional requirements like scalability and fault tolerance.

In languages that, like Erlang, include some form of dynamic code modification, runtime software change is usually mentioned among non functional properties; this means that it is often realized through specialized modules shared by different applications that need to implement such feature.

Precise definition of a set of design principles and full implementations of some commonly used design patterns are offered inside the OTP [9]. OTP stands for Open Telecom Platform; it is an application operating system and a set of libraries and procedures used for building large-scale, fault-tolerant, distributed applications. It was developed at the Swedish telecom company Ericsson and is used within Ericsson for building fault-tolerant systems.

Implementations of design patterns are called *behaviors*: they realize the aforementioned division of the code by implementing all the non-functional modules required by a large set of applications and dictating fixed schemes for the functional parts. In particular the functional part is usually required in the form of modules, named *callback modules*, which expose predefined set of functions, the *callback functions*. The *gen-server* behavior, that implements a generic server for distributed client server communications, is one of the most cited examples: the behavior already deals with all aspects concerning communication, and only requires the developers to specify the code that a server executes every time it is called.

Design principles assure that new written code can easily cooperate with existing built-in modules; for example each application written in Erlang/OTP has to adopt a common structure model for its processes: the *supervision tree* structure, that splits workers from supervisors and defines different degrees of supervision. Once a programmer adopts the conventions defined for this structure, he can easily delegate all problems concerning inter process communication and fault tolerance to well known (and well tested) behaviors.

Large part of the behaviors defined in Erlang/OTP are designed to take into account and simplify the evolution of software modules at runtime. This way dynamic software change management is disciplined, explicitly indicating places in the application code where changes are allowed and consequently reducing the possibility of errors.

3.1.3 Release Handling

In Erlang/OTP an *application* is a unit of code that implements a specific functionality, usually adopting one or more predefined or user defined behaviors. An application can be thought as an autonomous component containing what is needed to achieve the goal it has been designed for. Thus, unlike behaviors, applications are not designed to be as much generic as possible, but to be easily reusable to perform a specific task. OTP defines a common schema for applications: in particular each application has to provide functions that enable external modules to start and stop it and, if needed, to configure its behavior.

It is common in Erlang to define applications using already existing applications to solve the different problems involved by the addressed tasks. This approach lead to the creation of hierarchies of applications: it is worth noting that in a hierarchy not only different applications can reside on different nodes, but also control can be fully distributed, allowing single applications to be restarted on working sites if the node they are working on crashes.

At the application layer, OTP introduces facilities to manage versions, called *releases*. Releases are annotated with the information used to manage them. For example, such annotations are used to simplify deployment on different sites and recovery if applications crash; interestingly, it is also possible to define dependencies between releases. Annotations

are processed and generate scripts to enable automatic upgrades and downgrades. As an example, using annotations it is possible to specify that a version 2 of a given application A requires a function f to be defined in an application B ; in this case, when the annotations are processed, the code for adding and removing f from B during upgrade or downgrade of A is automatically generated.

Upgrades and downgrades are defined in the following way. The modules in which a process has its tail-recursive loop functions are called *residence modules*, while a module that is not a residence module for any process is defined *functional module*. Functional modules are updated by directly exploiting dynamic loading; after the change, whenever their functions are called, the new version is executed. Residence modules are more complex, since during recursion data can be passed as parameter from call to call and new versions may need new structures for such data. To enable changes in residence modules, OTP allows developers to specify a *change function* that defines how existing variables have to be transformed in order to be used by the new version of the code. Runtime changes are consequently executed in four steps: first every active process executing functions belonging to the module to change is suspended; then the change function is executed in order to update existing data; third the new version of the module is loaded and finally every suspended process is resumed. It is worth mentioning that annotations associated to releases also allow to specify synchronization criteria to ease changes in distributed applications where modules running on different nodes cooperate.

The described approach, however, presents some limitations: it is not always possible to upgrade modules without shutting down the running program (for example in presence of circular dependencies); additionally, errors may occur during migration between versions (for example as a consequence to bugs in the change functions). To deal with these cases release annotations may contain also arbitrary commands to execute in case of unexpected crashes: such commands can be used to restart an application and to bring it to a convenient state.

3.1.4 Conclusions

Erlang offers facilities to support and manage software evolution at different abstraction layers: first, it has the embedded capability to dynamically load code. Second, as all functional programming languages, Erlang allows functions to be passed as parameters during the communication between different modules, thus allowing instructions to migrate at runtime. OTP, through the use of behaviors, defines and implements design principles that simplify code reuse and separate heterogeneous functionalities in different modules. Predefined behaviors take into account and discipline software evolution by explicitly separating functional and non functional modules, so that functional ones can easily be changed limiting the risk of error occurrences. Finally, autonomous components are realized in Erlang using applications: the possibility to annotate releases introduces support for software changes, and in particular to version management at a very high level of abstraction. An important aspect of this mechanism is the possibility to change not only the code of a module at runtime, but also the data used by such code: this feature is usually hard to find in languages and tools supporting runtime software changes. Erlang is

a typeless language; this means that, when code changes, there is no possibility for the compiler to provide even static guarantees on which kind of information a function will receive as parameter. To highlight how static type checking can influence mechanisms for runtime software changes, in the following we analyze a programming language allowing for dynamic changes without dynamic typing: Dynamic ML.

3.2 Dynamic ML

Dynamic ML [19] is a programming language that derives from Standard ML [29] and introduces support for dynamic code evolution without losing compile time type checking.

Standard ML is a reliable programming language which provides the expressiveness of a functional programming language in addition to imperative programming constructs. The programmer can choose which paradigm to choose, also supported by a clear distinction between mutable and immutable values. Standard ML is a strongly typed language: type checking is supported by making a clear distinction between *elaboration* and *evaluation*; elaboration is performed at compile time and assures static type safety; programs that have not been correctly elaborated cannot be evaluated (executed) at all. The rigid ordering of these two activities forbids the execution of any program that attempts to use data values in ways that are not allowed by their types, thus eliminating a large number of errors that could manifest at runtime. However, this strong distinction into phases also prevents the modification of programs at runtime.

Dynamic ML introduces the possibility to dynamically replace compiled program code at runtime. Dynamic ML inherits from Standard ML a polymorphic type system supporting type inference. In order to keep type coherence, it needs to provide a technique for software changes that enables not only code modification, but also migration of existing values from the data structures defined for the old type to the data structure defined for the new type. Changes are limited to address only internal structure and/or signature replacement. Additionally, to ensure preservation of type constraints a type S can be replaced by a type $S2$ only if $S2$ matches all the constraints defined for S . This means that $S2$ must not omit functions, values and types exported by S . This way, Dynamic ML only allows to define changes that add functionalities or modify its internal implementation.

Runtime software change is implemented in Dynamic ML by altering the behavior of the garbage collection procedure. In Standard ML garbage collection is performed by dividing the heap into two contiguous spaces: the *from space* and the *to space*. During normal execution only the *from space* is used; memory is allocated linearly until an allocation fails. At this point the garbage collector is called to look for free space: the *from space* is scanned recursively starting from root objects, and only living objects are copied in the *to space*. Then the role of the two semispaces is reversed, until a new garbage collection is performed. To execute software changes Dynamic ML requires the programmer to write an *Install* function which expresses how values of the old type we need to change can be transformed into values of the new type. Such function is then applied to all objects during garbage collection, before they are copied into the *to space*.

This process has some strong practical limitations: in particular it presents problems when objects are shared between different processes in a distributed scenario. In conclusion, the example of the Dynamic ML language shows how type

checking greatly influences the mechanisms defined for runtime software evolution.

3.3 Scala

As we said, software changes are used to address a variety of issues that can emerge from different scenarios. Recently, in the scenario of pervasive computing, one of the most important issues to be solved is the reuse of software components in different devices. In this context software components are not always usable by all devices as they are; for this reason they need to be *adapted* to meet the requirements of the platform they have to be deployed on.

Adaptation can be considered as a form of evolution: different programming languages have been defined to ease reuse and adaptation of software components. To introduce the topic, we briefly analyze one of them: the Scala programming language [31]. Scala cannot be considered a traditional functional programming language; however it has been put in this section because it shares with other functional languages some abstractions that can ease runtime software evolution, like for example higher order function. Scala is sometimes considered a superset of Java: actually, it is designed to work well in association with Java (as well as with C#) so that it can maximize the reuse of existing components written in these languages, but it cannot be exactly defined as a superset. It adds some functionalities but it also removes others and re-interprets some constructs to provide better uniformity of concepts. To summarize the main features of Scala we can say that:

- It has a uniform object model in the sense that every value is an object and every operation is a method call.
- It is also a functional programming language, in the sense that functions are first class values.
- It has uniform and powerful abstraction concepts for both types and values.
- It has flexible symmetric mixin-composition constructs for composing classes and traits¹.
- It allows external extension of components to ease adaptation.
- It is currently implemented on Java and on .Net platforms.

Here, for space constraints, we focus only on how it is possible to give external extension to components in order to facilitate adaptation and reuse. We do not discuss here how such features can be introduced at runtime, since Scala exploits the low level mechanisms provided by Java and .Net platforms, that have been already discussed in Section 2.

Scala enables extension of components using *views* that allow to augment a class with new members and supported traits. A view is introduced by a normal Scala method definition: the only difference is that views are inserted dynamically by the compiler. If we suppose that e is a value of type T ; a view is implicitly applied to e in one of two possible situations: when the expected type of e is not (a

¹Traits are abstract classes; like Java interfaces they cannot encapsulate state, neither in the form of variables, neither using parameterized constructors; unlike Java interfaces they can implement concrete methods.

supertype of) T or when a member selected from e is not a member of T . Views are very useful for adaptation as they allow programmers to introduce methods to a given type, so that it matches the requests of the platforms it has to be deployed on. The use of views can be controlled by considering their scope: views can be applied only if defined within the scope of execution and competing views can exist in the same program. Furthermore, views can be bounded, so that, for example, a view can limit the range of its application not to all instances of *Lists* types, but only for those implementing a *Comparable* interface. Finally, when multiple views can be applied, the compiler always chooses the most specific one among all candidates, where specificity is interpreted in the same way as for overloading resolution in Java.

4. THE ASPECT ORIENTED PROGRAMMING PERSPECTIVE

4.1 AOP in a nutshell

Separation of concerns [18] is one of the main principles of software design: it allows to deal with different parts of a problem, and concentrate on each of them individually. Concerns can belong to a *dominant dimension*, which comprehends software functional requirements, or to a *cross-cutting dimension* that may carry on non-functional requirements (e.g., security, logging, profiling, etc.). The latter are called **aspects** and are difficult to detect and separate from the dominant dimension, since the code carrying on their requirements is scattered throughout the software system. Moreover the nature of procedural and object oriented programming languages does not ease the decomposition of the different aspects of a problem and the separation between aspects' functionality and their execution point. For these reasons Kiczales et al. proposed Aspect Oriented Programming (AOP) [25], a new design technique supporting separation of cross-cutting concerns, their definition and reuse throughout the system.

AOP offers mechanisms to isolate aspects into separate components, putting in one point the code which implements a cross-cutting functionality that otherwise would be scattered throughout the system. The program execution points in which aspects must be applied are called **join-points** [24] (e.g. a call/execution of a method, access to a field, etc.). Join-points are detected specifying a **pointcut**, which represents a pattern the desired execution points must satisfy, and provides mechanisms to access to the execution context of a join-point. It is worth to note that there are pointcuts that are independent from the execution context, since their join-points can be statically bound prior execution (e.g., a method call). While there are *undeployable pointcuts* that can only be bound at run-time, since their join-points depend on the execution context. Examples of undeployable pointcuts are those that satisfy a logic expression, or those that depend on the reaching of another join-point by the program execution flow.

Each pointcut is associated with an **advice**, which specifies the behavior of the aspect and indicates if it must be executed before (*before advice*), after (*after advice*) or instead (*around advice*) the associated join points. Once the main aspects of a system are detected and isolated, it is necessary to integrate them with the main system components according to specific rules. This operation is called

weaving and can take place at different times: *build time* (during program compilation), *load time* (during class loading), and *run-time* (during program execution). Run-time weaving is necessary to deal with dynamic aspects that can be added and removed from the software system at run-time or can affect a set of locations that are not fixed prior execution. Tools providing this kind of weaving are usually classified under the name of *dynamic AOP*. Examples are PROSE [30], Steamloom [21], Wool [35], Handiwrap [13], JAsCO [36].

Dynamic AOP is suitable to support unanticipated software changes. In fact if the desired change is encapsulated into an aspect, it is possible to add/remove/modify it dynamically. Dynamic AOP provides several improvements over compile-time/load-time approaches. First, load-time/compile-time approaches do not allow to modify aspects after weaving is performed, hence it is not possible to modify aspects at run-time. Moreover load-time/compile approaches manage undeployable pointcuts in a less efficient way. In fact advices are woven in all potential locations in which an aspect may be applied and a simple code snippet (*residual*) is added in these locations to verify if the corresponding advice has to be executed. This way *residuals* are executed together with the main application, even if the advice they are associated with must not be performed, and, for this reason, they may cause slowdowns. Instead, dynamic AOP weaves aspects only when the execution flow allows to clearly identify the execution point in which they must be applied (e.g., when a code location is reached or a property is verified). This way no check is performed to verify if the necessary conditions for the advice executions are met. Finally, Dynamic AOP provides better support for *aspectual polymorphism*: even if an aspect's type is statically known, the set of affected join-points as well as the advice code are dynamically bound to a given abstract aspect. This also provides aspects reusability in different context and their customization depending on the context.

In the following we illustrate the main features of PROSE and Steamloom, since they are the most mature works available so far. During the discussion we use a running example: we want to augment all PDAs connected to a node of the network with dynamic persistency features and we want to remove this capability when a PDA leaves that node.

4.2 PROSE

PROSE is an infrastructure that support dynamic adaptation of Java applications by extending their functionality with new code. PROSE has evolved through a number of versions [33] [32] [30], each of them based on different forms of interception and weaving.

PROSE injects aspects through method code replacement without interrupting application execution. PROSE supports the following join points, as shown in Figure 2:

- *Method boundaries*: intercept method entry/exit to enable adaptations that extend an existing application by adding new functionality (e.g. messages encryption, or adding transactional controls to method calls).
- *Method redefinition*: replace a method body with a new one, modifying its behaviour (e.g. to substitute a communication protocol with a different one).
- *Field access and modification*: enable adaptation that

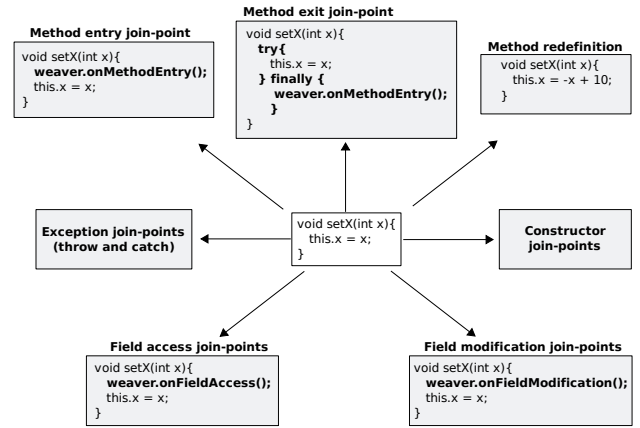


Figure 2: PROSE's join points execution model.

may involve, for example, orthogonal persistence, correct errors in measurements taken by sensors, or implementation forms of shared memory among mobile devices.

- *Exception*: intercept exceptions catch and throw to enable adaptations that correct anomalous situations that can happen, for example, in small devices and mobile computing settings.
- *Constructor*: intercept object instantiations, which may be useful for online monitoring and profiling. For instance, executing certain test after the instantiation of an object allows to detect performance bottlenecks.

Figure 3 shows an example of an aspect defined in PROSE which implements dynamic persistency when property field of class `Service` is modified. PROSE encapsulates aspects in a java class, which must extend the base class `DefaultAspect`. Advices and pointcuts are encapsulated into object `Crosscut`. PROSE offers several predefined crosscuts: `SetCut` and `GetCut` intercept accesses to object's fields; `MethodCut` intercepts executions of method entries and exits; `MethodRedefineCut` captures method bodies; `ConstructorCut` intercepts object instantiations; `ThrowCut` and `CatchCut` intercept throw and catch operations. In our example we use `SetCut` since we need to intercept modification of property `field`.

Advices and pointcuts are both implemented as methods. The advice name depends on the crosscut type, e.g. in our case our advice is `SET_ARGS`, since we adopted crosscut `SetCut`. Other advice types are also provided. Each advice takes a set of arguments that form the signature used to define potential methods that have to be replaced. The arguments in the example indicate that the advice will match all methods of all object types (`ANY target`) that have an Integer as a first parameter (`Integer arg1`). This set of methods is further qualified by the pointcut (method `pointCutter`), which specifies that the set method must be defined in the class `Service` (`Field.declareInClassService`) and the property name must be `field` (`Field.named(field.*)`). Together with the specializer `Field` adopted for our example, PROSE provides other specializers: i.e., `this`, which allows the execution of an advice depending on the properties of the object referenced by `this`; `target`, which works as the

```

public class PersistenceAspect extends DefaultAspect {
    public SessionFactory sf;
    public Crosscut makePersistent = new SetCut {
    public void SET_ARGS(ANY target, Integer f){
        Session ses = sf.openSession();
        ses.saveOrUpdate(o);
        ses.flush();
        ses.connection().commit();
        ses.close();
    }
    PointCutter pointCutter() {
        return((Fields.declareInClass("Service.*").AND(Fields.named("field.*"))));
    }
}
public static void main(String[] args) {
    ProseSystem.startup();
    DefaultAspect asp = new PersistenceAspect();
    ProseSystem.getAspectManager().insert(asp);
}
}

```

Figure 3: Definition of an Aspect in PROSE.

previous one but depends on the property of the method target. It is also worth to note that PROSE supports the construct `proceed`, that permits to invoke the original method that was replaced by an around advice.

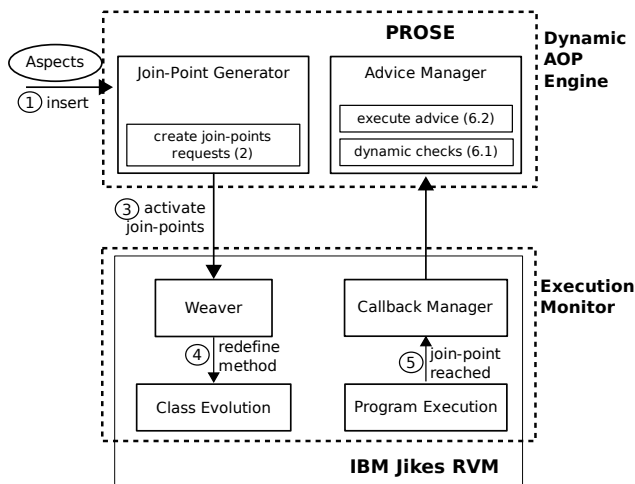


Figure 4: PROSE Architecture.

PROSE provides support for aspect weaving and execution through a flexible architecture shown in Figure 4. It is divided in two main components: the **Dynamic AOP Engine** and the **Execution Monitor**. The **AOP Engine** accepts aspects (1) and transforms them into join-point requests (2), that are a description of the code location where the execution must be interrupted to call an advice. The **Execution Monitor** extends the Jikes RVM by adding support for method code replacement at run time (**Class Evolution**) and method execution interception (**Program Execution**). Furthermore, it provides a **Weaver** which accepts weaving requests from the **AOP Engine** (3), gets the original bytecode of each method, and instruments the affected methods using the services offered by the **Class Evolution** module (4). When the program execution reaches one of the activated join-points (5), the **Callback Manager** notifies the **Advice Manager** in the **AOP engine** which then executes the corresponding advice (6).

PROSE uses (a) an advice weaving strategy for method re-

placement join-points and (b) a stub weaving mechanism for the other join-points that involve external advices [?], that simply add new functionalities to the existing application code. Since Jikes RVM executes native code, every time an aspect is woven/unwoven re-JIT (Just In Time) compilation is necessary. Jikes RVM has the advantage to allow to modify bytecode also of active methods. The different kinds of weaving are meant as alternatives depending on the type of application involved. For example weaving advice code has the advantage of providing more concise advices, though it may cause security problems [32], since it is not possible to keep control on what local resources can or cannot be used by advice code originating from foreign hosts. While a stub weaving mechanism allows to cope with security problems when an advice contains untrusted code. In fact advice and application code are separated and the advice can be executed externally, i.e. in another virtual machine.

PROSE's architecture is flexible, since once defined a uniform interface for the **Execution Monitor**, it is possible to use any AOP engine on top of it. PROSE demonstrated to have high efficiency: the overhead introduced without weaving is only 1.87% and its performance are comparable with compile-time AOP approaches. The adoption of Java language to represent aspects reduces the programmers' effort to create an aspect. PROSE's drawbacks are that its pointcut model is not powerful enough, since pointcut definition is split between the advice signature and in the pointcutter definition.

4.3 Steamloom

Steamloom is the first VM implementation which supports dynamic aspects and exposes an API to access to the AOP functionality offered by the VM. Its implementation extends IBM's Jikes RVM with a bytecode manipulation toolkit, i.e. BAT (Binary Adaptation Toolkit), that allows to easily modify methods' bytecode and querying them for join-points reaching notifications.

```

public class PersistenceAspect{
    public static Asepect setupAspect(){
        Method persistMethod = PersistenceAspect.class.getDeclaredMethod("persist", null);
        Advice persistAdvice = new AfterAdvice(persistMethod, null);
        PointcutDesignator persistPcd = SimpleParser.getPointcut("set(* Integer Service.field.*)");
        Aspect a = new Aspect();
        a.associate(persistAdvice, persistPcd);
        return a;
    }

    public static void persist() {
        Session ses = sf.openSession();
        ses.saveOrUpdate(o);
        ses.flush();
        ses.connection().commit();
        ses.close();
    }
}

public static void main(String[] args){
    Aspect a = null;
    a = setupAspect();
    a.deploy();
}
}

```

Figure 5: Definition of an Aspect in Steamloom.

The language adopted by Steamloom is a modified version of CaesarJ [1]. Figure 3 shows the definition of the aspect which performs dynamic persistency. An aspect is a mere container that maps pointcuts to an advice (`a.associate(persistAdvice, persistPcd)`). Aspects are first class entities that can be dynamically defined (see method `setupAspect`), activated (through instruction `deploy`) and deactivated (through instruction `undeploy`). An instance of class

Advice encapsulates a reflective representation of the advice behavior. In this case we adopted a before advice but also after advices are supported. Steamloom's pointcuts are represented as objects hierarchies of `PointcutDesignator` instances. Steamloom allows to define pointcuts adopting the same syntax of AspectJ, and provides a suitable interpreter to convert pointcuts into their corresponding join-points.

Before explaining how Steamloom performs weaving it is necessary to spend few words on Jikes RVM. It offers the choice among three different compiler systems: a baseline compiler that performs no optimisation, an optimising compiler and an adaptive optimization system (AOS), built on top of other compilers to perform profiling. Each class loaded in the RVM has a reference to all its virtual methods through a TIB (Type Information Block). The first time a method is called a lazy compilation stub is executed. This stub retrieves the method's bytecode, compiles and executes it. Next time the method is called, the compiled code can be executed directly. While at a certain execution point the AOS can decide to generate an optimised version of the code depending on the profiling data received from the measurement subsystem.

Steamloom allows to deploy aspects with different scopes: class-wide, instance local, and thread local aspects. For class-wide aspects, if a method was baseline compiled, the compilation is invalidated and the method will point again to a lazy compilation stub which retrieves the updated bytecode and performs recompilation the next time it is invoked. If the method is optimized, aspect deployment logic immediately recompiles the method at the optimization level it was previously compiled. When an aspect is applied to only one instance of a class, Steamloom clones the affected object's TIB. If the affected method was baseline compiled, the TIB is cloned upon deployment of the aspect and its respective entry is changed to point to the lazy compilation stub. The main drawback of instance-local aspects is that affected methods cannot be inlined. In fact an inlined method is not looked up via the TIB and its code is executed in place, ignoring instance-local aspects. For thread-local aspects a brief snippet of code is inserted before every call to an advice. This code checks the thread identity and skip the advice invocation if the respective aspect is not meant to be active in the current thread. Thread-local deployment is a way to reconcile dynamic aspect deployment and multi-threading to maintain the consistency of the system.

Steamloom offers several advantages over PROSE, since it provides an advanced pointcut system. But it has the disadvantage to adopt its own programming language to create aspects, raising programmer's effort to write aspects. Steamloom is less powerful than PROSE since it does not support around advices, and those that intercepts exceptions catch and throw. It also does not allow to compose aspects and associate more than one aspect to a pointcut. Regarding the proposed use case (see Section 4.1), the solution provided by Steamloom is less flexible, since it requires each PDA to be equipped with a complete modified virtual machine. This approach is also less secure, since weaved code is executed together with the main application (in the same JVM). While PROSE is more flexible, since it requires each PDA to have only weaving functionality while, advice execution can also be managed outside the device. It is more secure since stub weaving allows to call code executed in other places.

5. CONCLUSIONS

In this Section we summarize the answers provided throughout the paper to the questions raised in Section 1. To discuss the type of changes each language is able to apply we adopt two metrics. The first one determines if there exist limitations in the way modules are affected by a change (e.g a change cannot modify the methods' signature, etc.), while the second one indicates if the change also affects live applications instances or only the future ones. Among object oriented languages *C#* performs better than Java, since it allows to change everything except the instance format, while Java allows only to modify method bodies. Since they relies on Hotswap facilities, changes will affect only the future class instances. Erlang allows to perform any kind of change (add/modify/remove modules, modify a module's internal variables), which immediately affects application's live instances. DynamicML poses more limitations than Erlang since changes can only affect a module's internal structure or a module's signature and they must satisfy type constraints. Furthermore, modifications only impact on the future application instances, since they are applied during garbage collection. However, programmers can explicitly specify a transaction function that enables the migration of live instances' structure. Scala is suitable only to provide views, and, since it leverages the services offered by CLR or JVM, changes can only be applied on future class instances. Among dynamic AOP approaches PROSE allows to perform a wider set of changes (e.g. methods replacement and change the value of its internal variables) than Steamloom (method replacement is not permitted) and injected modifications can also affect active methods.

Each language relies on a set of low level mechanisms to perform dynamic changes, also influencing the set of changes supported. Java and *C#* depend on the services provided by their virtual machines for profiling and bytecode rewriting. Erlang leverages dynamic code loading and release management capabilities, that allow it to handle different modules' versions. While DynamicML relies on garbage collection and Scala depends on the platform it is built on. Finally both Steamloom and PROSE rely on the JIT compilation to replace method bytecode. Moreover Steamloom uses BAT to represent bytecode instructions and easily detect join-points.

Each language hides adopted low level mechanisms providing higher level abstractions to support dynamic evolution. Java and *C#* only offers the possibility to invoke external modules, written through dynamic languages. Erlang hides the low level details of dynamic code loading behind well known design patterns, implemented as behaviors. Dynamic ML provides static type checking, which forbids illegal type changes, while Scala offers the concept of view. Finally AOP concepts of separation of concerns and pointcuts permit to easily detect changes on crosscutting features and ease their injection at runtime.

We also discuss how the languages presented above address two main representative use cases in which dynamic software evolution is needed. In the first one users need to fix a bug or to add some basic functionality to a software system. In this case Erlang behaves better since it supports any kind of change, managing concurrency. AOP approaches are less powerful since they allow to apply only changes that involve methods replacement, but, like Erlang, a change can be performed at any time. DynamicML and OOP behaves worst since they support a smaller set of changes that will

affect only applications' future instances.

In the second use case users need to add and remove cross-cutting functionalities (e.g., persistency, logging, etc.). In this case AOP languages perform better since they ease the isolation of crosscutting concerns and allow to inject them at runtime at specific code locations through pointcuts. Erlang and DynamicML perform worst: even if they support modularization, they do not allow to specify pointcuts. Moreover DynamicML has the additional drawback to support a smaller set of changes. OOP languages have the disadvantage to not offer separation of concerns at all. Finally, Scala is not suitable for any of these use case and can be adopted to adapt the same entity to different context: e.g. when a method parameter adopts another format, or an instance adopts another class among those compatible with the previous one.

6. REFERENCES

- [1] CaesarJ Project. <http://caesarj.org/>.
- [2] Da Vinci Machine. <http://openjdk.java.net/projects/mlvm/>.
- [3] Dynamic language runtime. <http://www.codeplex.com/dlr>.
- [4] Java 1.6 serialization specification. <http://java.sun.com/javase/6/docs/platform/serialization/spec/version.html#9419>.
- [5] Java hotswap jpda. <http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html>.
- [6] Java hotswap jvm ti. <http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html>.
- [7] Java specification request 292. <http://www.jcp.org/en/jsr/detail?id=292>.
- [8] Mono c# shell. <http://www.mono-project.com/CsharpRepl>.
- [9] Otp, design principles. http://erlang.org/doc/design_principles/part_frame.html.
- [10] ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- [11] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall, 1996.
- [12] ARMSTRONG, J. L., WILLIAMS, M., VIRDING, R., AND WILKSTRÖM, C. *ERLANG for Concurrent Programming*. Prentice-Hall, Inc., 1993.
- [13] BAKER, J., AND HSIEH, W. Runtime aspect weaving through metaprogramming. In *AOSD '02*, pp. 86–95.
- [14] BENNETT, K. H., AND RAJLICH, V. T. Software maintenance and evolution: a roadmap. In *ICSE '00*, pp. 73–87.
- [15] BUCKLEY, J., MENS, T., ZENGER, M., RASHID, A., AND KNIESEL, G. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.* 17, 5 (2005), 309–332.
- [16] CHAPIN, N., HALE, J. E., KHAM, K. M., RAMIL, J. F., AND TAN, W.-G. Types of software evolution and software maintenance. *Journal of Software Maintenance* 13, 1 (2001), 3–30.
- [17] DMITRIEV, M. Towards flexible and safe technology for runtime evolution of java language applications. In *Towards flexible and safe technology for runtime evolution of java language applications* (October 2001).
- [18] GHEZZI, C., JAZAYERI, M., AND MANDRIOLI, D. *Fundamentals of software engineering*. Prentice-Hall, Inc., 1991.
- [19] GILMORE, S., AND WALTON, C. Dynamic ml without dynamic types. Tech. rep., 1997.
- [20] GUSTAVSSON, J. A classification of unanticipated runtime software changes in java. In *ICSM '03*, p. 4.
- [21] HAUPT, M., MEZINI, M., BOCKISCH, C., DINKELAKER, T., EICHBERG, M., AND KREBS, M. An Execution Layer for Aspect-Oriented Programming Languages. In *VEE '05*, pp. 142–152.
- [22] HEJLSBERG, A. The future of c#. <http://channel9.msdn.com/pdc2008/TL16/>.
- [23] HICKS, M., AND NETTLES, S. Dynamic software updating. *ACM Trans. Program. Lang. Syst.* 27, 6 (2005), 1049–1096.
- [24] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of aspectj. In *ECOOP '01*, pp. 327–353.
- [25] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP* (1997), pp. 220–242.
- [26] KIM, D. K., AND TILEVICH, E. Overcoming jvm hotswap constraints via binary rewriting. In *HotSWUp '08*, pp. 1–5.
- [27] LEHMAN, M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* 68, 9 (Sept. 1980), 1060–1076.
- [28] LIENTZ, B. P., SWANSON, E. B., AND TOMPKINS, G. E. Characteristics of application software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.
- [29] MILNER, R. A proposal for standard ML. In *LFP '84* (1984), pp. 184–197.
- [30] NICOARA, A., ALONSO, G., AND ROSCOE, T. Controlled, systematic, and efficient code replacement for running java programs. In *Eurosys '08*, pp. 233–246.
- [31] ODERSKY, M., AND AL. An overview of the scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland, 2004.
- [32] POPOVICI, A., ALONSO, G., AND GROSS, T. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03*, pp. 100–109.
- [33] POPOVICI, A., GROSS, T., AND ALONSO, G. Dynamic weaving for aspect-oriented programming. In *AOSD '02*, pp. 141–147.
- [34] R. GRIFFITH, G. K. Adding self-healing capabilities to the common language runtime. Tech. rep., Columbia University, 2005.
- [35] SATO, Y., CHIBA, S., AND TATSUBORI, M. A selective, just-in-time aspect weaver. In *GPCE '03*, pp. 189–208.
- [36] SUVÉE, D., VANDERPERREN, W., AND JONCKERS, V. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03*, pp. 21–29.
- [37] WIKSTROM, C. Distributed programming in erlang. In *PASCO'94*, pp. 412–421.