Influence of type systems on dynamic software evolution

Yves Vandewoude*, Yolande Berbers
Department of Computer Science
KULeuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
{yvesv, yolande}@cs.kuleuven.ac.be

Peter Ebraert*, Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussel, Belgium
{pebraert, tjdhondt}@vub.ac.be

Abstract

Currently, no programming language support exists for dynamic software evolution. Additional effort is required from developers to compensate for this lack of support. In this paper we analyze and categorize the most important aspects a programming language should offer in order to adequately support dynamic software evolution. We subsequently analyze the suitability of existing programming languages for dynamic software evolution. Their type system is taken as a distinguishing characteristic and the impact of the type system on many different aspects of dynamic software evolution is investigated. The main contribution of our paper is a table that summarizes our findings and clearly shows that a more flexible type system implies more power to carry out runtime changes but at a cost of reduced security. This table can be used by application developers to select a suitable language for their project or by language designers in the quest for a dynamic adaptation language.

1 Introduction

"If it's not broken: don't fix it." This idiom is above all applicable to software. Nevertheless, software evolution is an enormous problem that makes up for more than 80% of the cost of a software system ([23]). Even if a software system seems to work flawlessly from a user's point of view, it may be difficult to maintain or adapt. Despite many modern design technologies, evolving software in order to add new features, fix a bug or even port it to a new platform is an extremely cumbersome process. Techniques such as refactoring attempt to lower the cost of evolution by continuously enhancing the structure of an application by structural changes such as reducing coupling.

*Authors funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen)"

Live, that attempt to evolve a running system without shutting it down, dramatically increases the complexity of the evolution problem since there are considerably more constraints on a running system. The updates must complete in a short timeframe, must deal with the state contained in the active application and consistency must be preserved during and after the change.

2 Problem statement

Currently more than two thousand programming languages already exist, each one different from the other. These programming languages can be grouped in categories with common characteristics. Some widely accepted taxonomies exist that define key properties of different programming languages (such as [1, 25]).

Such taxonomies clearly show that there are no superior or inferior languages. Every language has its strengths and weaknesses and was designed with a specific goal in mind: Scheme for educational purposes, SQL for database querying, C for system programming, Fortran for mathematical computations and Java for portability. These goals are clearly reflected in the design of the language as specific design decisions have been made to ensure that the primary goal could be optimally supported.

Evidently, many programming languages are general purpose and can be used for purposes they were not designed for. However, using these languages often implies an additional effort from the programmer. This is what currently happens for dynamic software evolution. Languages that were not specifically designed to support runtime changes are used to implement adaptable software systems. Since no language support is provided for problems specific to dynamic change, programmers typically attempt to bypass them using a layered approach [5, 11, 20, 31, 33]. We claim that a programming language specifically designed with dynamic evolution in mind would resolve some

of the more common problems and increase the practical applicability of adaptable software by providing features such as security and state consistency. Unfortunately, to the best of our knowledge, no such programming language has been developed so far.

The goal of our research is to establish the specification of programming languages that are well suited for dynamic software evolution. This specification can then be used (1) by software engineers to choose an appropriate language for developing software, and (2) by language engineers, to develop a new programming language towards dynamic software evolution.

Discussing every aspect of each programming language is not feasable. Therefore, based on the previously mentioned taxonomies of programming languages, this paper will use the type system of the language as its distinguishing feature. As will be shown, many other features of a programming language are directly related to its type system (or lack thereof). In order to investigate the influence of the type system on different aspects of evolution, these aspects must be identified as well. For that purpose, a taxonomy developed by experienced researchers in the field of software evolution is used [6].

The remainder of this paper is structured as follows: in section 3 different aspects of typing in programming languages are discussed. Section 4 presents their influence on the different dimensions of evolution (as identified in [6]). In section 5, a real life example that illustrates the results from our analysis is presented. We continue with related and future work (section 6) and conclude in section 7.

3 Type systems

This paper uses the type system of a programming language as its distinguishing feature. A type system itself, however, is not a trivial concept. Since it is of vital importance to establish accurate terminology for the remainder of this paper, this section introduces relevant concepts and definitions on type systems. According to [25], the four defining characteristics of a type system are *type binding*, *type checking*, *type conversions* and *type strength*.

3.1 Type binding and type checking

Definition 1 (Type Binding) *Type binding is the process of assigning a type to a variable. A distinction is made between static and dynamic binding depending on whether the binding occurs at compile-time or at run-time respectively.*

Definition 2 (Type Checking) *Type checking is the process that verifies whether the operands of an operator have compatible types. When an operator is applied to an operand of an incorrect type, a type error occurs. Depending on when the type checks occur, the term 'static type check' or 'dynamic type check' are used.*

These definitions present two distinct properties. Static type checking does not always imply static type binding (e.g. generics). In addition, a language could have static binding with dynamic checking or type binding without checking. The combination of static type binding and static type checking is common, and is referred to by the term static typing. This corresponds to the intuitive definition which says that a language is statically typed if type information is present (and used) at compile time. It is not relevant whether the type information is directly available through annotations (such as in Java or C#) or inferred (like in Haskell or ML). By analogy, the term dynamic typing refers to a language that is both dynamically bound and dynamically checked.

A common mistake is the idea that a programming language must be either statically or dynamically typed [7]. It is vital to stress that static and dynamic typing are *not* mutually exclusive. Many programming languages combine both static and dynamic typing (e.g. Java, C#), other languages are untyped (Prolog, assembly).

3.2 Type conversion

The term *type conversion* is used to describe the conversion of a variable from one type to another. Two different type conversions exist: implicit conversions (*coercions*) and explicit conversions (*castings*). The difference between both kinds of conversions lies in the annotation: whereas explicit conversions are always annotated, implicit conversions are detected and automatically added by the programming system.

3.3 Type strength

About consensus on the definition of type strength, the renowned programming language expert Benjamin C. Pierce stated:

> I spent a few weeks trying to sort out the terminology of strongly typed, statically typed, safe typed, etc., and found it amazingly difficult. The usage of these terms is so various as to render them almost useless.

Very often, ad hoc definitions are given that are based on consequences of type strength. Many definitions in literature are contradictory, others are merely orthogonal ([12, 24, 27, 17]). It is not a goal of this paper to try and choose *the best* definition. However, since a terminology is required for the remainder of this paper we take the following definition:

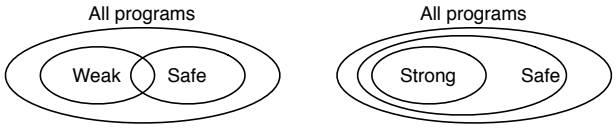

Figure 1. Weak versus Strong languages

Definition 3 (Type Strength) *The terms strong and weak typing refer to the effectiveness with which a type system prevents type errors. A strongly typed language prevents any operation on the wrong type of data. In weakly typed languages there are ways to escape this restriction: coercion.*

For example: the Python programming language is strongly and dynamically typed. Recently, discussions within the community lean towards the inclusions of (optional) static annotations for Python as this would enhance the readability of the code and the compilation speed. Since these annotations are optional, type checking is only performed on these parts of the program that have been annotated. The new Python would therefore be weaker typed than the current language, since coercions would be necessary to combine not annotated with annotated code. This example shows that type strength is not entirely independent from type binding and type checking.

Figure 1 shows the relation between type strength and type safety. A program is considered to be type safe, if it cannot have any type errors. The figure shows that weakly typed systems accept unsafe programs. Strongly typed systems only accept these programs for whom it can guarantee that there are no type errors. This is only a subset of all type safe programs. An extreme example is a type system that rejects every possible program. While evidently safe, it is also extremely useless. This is an example of the well known trade-off between type-safety on the one hand and language-power on the other hand [27]. Note that a `ClassCastException` is not considered to be a type error since its effect is defined by the language specification.

3.4 Impact

One of the main reasons that the type system of a language has a large impact on its dynamic evolution capabilities is because of the strong relationship between the type system and reflection. Reflection is the ability to inspect (introspection) and modify (intercession) the high-level structure of a program at runtime. Reflection therefore implies the ability to generate new programs at runtime [18].

To realize full reflection, all system entities must be available as first-class entities which can be inspected or modified (in OO languages these entities are methods and objects). However, the reification of a runtime structure into a first-class entity requires that such a structure exists. This is only the case in the presence of dynamic typing.

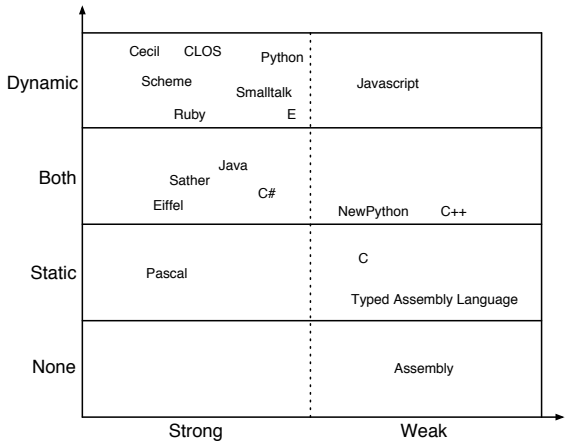

Figure 2. Prog. languages and their typing

Dynamic typing does not prevent the presence of a static type system. Nevertheless, static typing and reflection are two properties that do hinder each other. The first reason is that static typing is based on deduction of universal properties about a programs behavior before it is run. Reflection strongly complicates this, since it can be used to alter behavior based on run-time input, which is not available for prior analysis[1][3]. A second explanation of the clash between a static type system and reflection can be found in the fact that a static type system adds constraints that forbid certain types of changes to occur (all changes that violate type safety). Reflective systems for purely dynamically typed languages (such as Smalltalk) are therefore often more powerful than those for statically typed languages. It is important to remark that there is no fundamental technical issue in implementing reflective capabilities that violate type safety for statically typed languages, but that such an implementation would be in fundamental violation of the language itself (for an example: see section 4.3.3).

Figure 2 shows an overview of a few well known programming languages and their typing. There is no clear separation between strongly and weakly typed languages. The further away from the origin of the graph, the more flexible and powerful a language becomes. Purely dynamically typed languages are more flexible than languages with both static and dynamic typing, since their reflection mechanisms are not bound by language restrictions that are caused by their static typing. Hence, their location at the very top of the figure. The power of languages without static typing also implies a danger. Therefore, when safety is an issue, safer languages are desired which are located closer to the origin. As runtime type information is required at runtime

[1] This is also why reflection is ignored in most type formalisms.

in order to achieve full reflection, it can only be provided by these languages that are either dynamically, or both statically and dynamically typed.

4 Evolution Taxonomy

Directly analyzing the applicability of all different programming languages to runtime software evolution is not only unfeasible, it would also very quickly result in ideological discussions of specific language features. In this paper, we take the reverse approach and start from an existing taxonomy on evolution in which all different aspects of dynamic software evolution are treated. For each of these aspects we examine how specific language features (such as their type system or reflective capabilities) influence the ability of a language to deal with the evolution property under consideration.

As a base for discussion we take the taxonomy by renowned researchers in the field of software evolution [6]. Their taxonomy classifies different aspects of evolution in four dimensions: the object of change (where), change support from the underlying system (how), properties of the system under consideration (what) and temporal aspects (when). In addition, we will also touch very briefly on a fifth dimension: the purpose of change (why).

4.1 Purpose (why)

Most papers in literature identify four major types of maintenance activities: perfective, adaptive, corrective and preventive [16]. A more extensive taxonomy of maintenance activities based on their purpose is given in [8] in which Chapin et al. identify twelve different types of evolution. In general, the reason why one would attempt a runtime change to a running program is independent from the technology that is used to achieve this goal. Nevertheless, the better a technology supports runtime software evolution, the more likely it is that such technology will actually be used.

In statically typed languages, performing a runtime change is generally quite difficult, since one must uphold guarantees of the language that no type errors can occur in the process of a change. While certainly not impossible, such an attempt is non-trivial and requires a lot of preparation. It is very unlikely that one would take the effort to do so unless it can not be avoided.

For some purely dynamically typed languages however, certain forms of dynamic modification can be executed with relative ease. The `become` operator in Smalltalk allows switching two objects, is easy to use and has global impact on the active image. In addition, interactive development environments (such as the Smalltalk browser, but also some Prolog development environments) allow modifications so easily that runtime changes are truly incorporated in the development process: programs are constructed through incremental change. This process is essentially part of software development and differs from true live updates since many important problems (such as state transfer) are not addressed.

4.2 Temporal Aspects (when)

The *when* question addresses temporal properties of software maintenance tasks such as the moment when the change is executed, the change frequency and how different versions of the same software are treated and maintained.

4.2.1 Time of Change

Depending on the programming language and development environment being used, changes can be executed in different phases of the software development cycle. Changes can either be made to the software offline (i.e. while the software is not active) or online (during its execution).

Offline changes Offline changes are popular because of their ease of implementation: there is no predetermined timeframe in the change must complete, both testing and verification of the change is possible and a variety of information (such as the source code) is available. It therefore makes sense to prepare an online update using offline changes wherever possible [32, 9, 28]. One example of such preparation is the addition of getter methods to facilitate state extraction at runtime. Although offline changes can obviously be applied to any program, regardless the programming language used for the implementation, code instrumentation is much more common with statically typed languages. There are two reasons for this phenomenon:

Information Availability: Code instrumentation embeds information known at design time into the code so that it can be used at runtime. However, for purely dynamically typed languages, hardly any information is available that can not be easily derived at runtime. A static type system provides the user with more information that can be used as a base for analysis or can be included in the code. For instance, in [32], code is instrumented with information on corresponding structures between different versions, which is used by the state transfer algorithm. Without the offline instrumentation, such information could never have been derived at runtime.

Less powerful reflection: Purely dynamically typed languages often have a more advanced reflective system which eliminates the use for some forms of code instrumentation (such as the addition of getter methods to extract state information).

Online changes Generally speaking, static and strong typing hinder online changes, since they enforce a number of rules that prohibit certain changes. The removal of a method from a type, for instance, will not be allowed by the type system. To achieve this result, a new type must be created (identical to the old type except for the removed method) and all old objects would need to be converted to their counterpart from the new type. This is clearly more complex than direct type modification such as those allowed by the reflection mechanisms of Smalltalk.

For this lack of flexibility, the type system offers increased safety in return. However, if not all changes are type-safe (and this is often the case in a typical evolution scenario), the type system only gets in the way. In these cases, correctness must be verified using invariant checkers, which can be implemented regardless of the language used. More information on safety is given in section 4.4.4.

4.2.2 Change history

A change history of a software system refers to the history of all changes that have been made to the software. Static versioning implies that different versions can coexist at compile time or at load-time, but cannot exist simultaneously at runtime. It is clear that static versioning can be supported for any language. All that is required is a versioning system that keeps the sources of different versions available (such as CVS or subversion).

Full versioning implies that multiple versions can coexist at runtime. This is a desirable feature since it allows for lazy conversion: the new implementation is used to construct new instances whereas the old instances remain unchanged and are gradually removed from the system after their task has completed. In addition, the ability to have different versions in memory allows for an easy implementation of a rollback functionality. If only one version can be present at a given time, changes are by definition destructive, and rollback is harder to implement.

While full versioning is possible for both dynamically and statically typed languages, it is more naturally supported for the latter, since both versions are considered to be different types. Nothing prohibits such duplication with purely dynamically typed languages (Smalltalk for instance even allows storage of source code in its classes). However, this construct would sacrifice many of the advantages of purely dynamically typed languages such as type modification through its reflective system.

4.2.3 Change Frequency

The change frequency is inversely proportional to the effort required to achieve a runtime change. Since this effort is in general lower for weakly and purely dynamically typed languages, change frequency will be higher for these languages. This process is in part self-sustaining: if changes are more frequent, they tend to be smaller and easier to implement. This in turn decreases the barrier for their use. In practice this is confirmed by iterative development environments for languages such as CLOS and Smalltalk.

For statically typed languages, these iterative development environments are virtually non-existent. Live updates, even without any proof of their correctness, such as is the case of Smalltalk) are difficult to achieve, and therefore intentional. Using the terminology of [6], live updates for statically typed languages are almost always periodical as opposed to the continuous nature of dynamic adaptations in languages without a static type system.

4.3 Object Of Change (where)

The object of change describes the *where* dimension in the taxonomy of software evolution, as a change is always applied to a specific target. In this section, we assess how the programming language influences the size of the smallest unit of adaptation as well as the impact of the change and its propagation in the rest of the system.

4.3.1 Anticipation

Anticipated changes are easier to tackle at runtime than unanticipated changes. The more changes a piece of software anticipates, the better it is armed to deal with evolution later on in its life-cycle. It is clear however, that at the application level, not every possible change can be anticipated.

The main impact of anticipation is that it indicates what portions of the software can easily be changed. The only goal of techniques such as code instrumentation as a preparation for a live update is to increase this portion. The used language and runtime infrastructure have a large impact on anticipation, since all concepts that are modeled as first class entities can be changed at runtime without difficulty. For instance, a message is a first-class entity in Smalltalk and as such, method invocations can be rewired with ease. This would be much harder in Java, where the same concept is not treated as a first-class entity. Both meta-protocols and reflection are techniques to anticipate a number of changes at the language level and systems that provide these options can be considered as extensible programming languages.

In theory, if every concept of the language is fully reified, a language can be constructed that anticipates every possible change and is therefore ideal for dynamically updateable software. Whether the construction of such a language is a realistic undertaking remains subject for debate. However, it is clear that there is a huge difference between different programming languages regarding their support for anticipation. The relation of this support with their type system

is less obvious. Languages without a static type system enforce less rules and therefore allow easier implementation of many types of unanticipated changes. As such, purely dynamically typed languages can be considered more anticipative than statically typed languages.

4.3.2 Artifact

Software consists of a number of inter-related artifacts such as source code, design models, documentation and test-suites. Dynamic software evolution is exclusively concerned with dynamic artifacts (used at runtime) such as modules, functions and objects. While it is clear that co-evolution of the other artifacts is a desirable feature, the evolution of static artifacts is outside the scope of this paper. Some aspects of dynamic evolution (such as code-instrumentation) should be explicitly excluded from static artifacts, since they do not represent a property of a given version, but are directly related to the evolution process itself. They should therefore be transparent to the user.

4.3.3 Granularity

The degree of granularity defines the minimal size of the unit of replacement. The finer this granularity is, the more difficult it is to ensure consistency. After all, coarse grained components typically contain many tightly-coupled objects which are then replaced in group. Fine grained changes will need to deal with these strong interrelations.

Fine grained changes (within a type) are typically inhibited by statically typed languages, since a large number of changes are forbidden by these languages. With purely dynamically typed languages, there is no static type that an object must correspond to. Hence, a type can easily be changed without the need to create a new type in the process. The powerful reflective mechanisms often found in purely dynamically typed languages proof this fact. While it would be technically feasible to develop an equally powerful reflective API for statically typed languages, the language definition forbids all modifications that could violate type safety, which include most changes to a type. For instance, it is perfectly possible to implement a Java API that would allow the removal of methods from a Java class at runtime similar to the reflective system found in Smalltalk. If a method is removed and subsequently used, in Smalltalk, a `MessageNotUnderStood` exception is thrown. However, the programmer is aware that such behavior can occur and the language specification dictates that such an event may occur. In Java, such operation breaks the semantics of the programming language, since Java ensures that once the type of an object is known, all its methods can be used. Two options remain: either check dynamically that the removal of the method can never result in an error or dump the guarantee. The former option, if at all possible, would be extremely expensive to do at runtime. The latter option implies removing static type checking from the language, which would result in a Java-syntactic Smalltalk variant.

The main advantage of fine grained changes is that they usually do not require the transfer of complex state. Statically typed languages can emulate fine-grained changes with coarse grained changes (creating a new type which is identical to the old type except for the fine grained change). However, this is pointless, since the state-preserving features of fine grained changes (their primary advantage) are no longer applicable.

4.3.4 Change Impact

The impact of a change describes to what extend changes, which are applied to a software entity, are contained within the boundaries of that entity. Automatically determining whether a change has local or system wide impact is not always trivial. Static typing clearly helps in the localization of affected portions of the system since type information is the main source of impact-analysis. A clear example of the correlation of the type system and its ability to determine change impact can be found in the context of refactoring [13]. For statically typed languages (e.g. Java) many refactorings (such as a method rename) can be automated completely. This is impossible for pure dynamically typed languages since they do not provide the means to identify the target of a method invocation or variable access at design time. For instance, the type of the parameters of a method invocation can not be used to identify the correct method. As such, tool support is therefore limited to an exhaustive listing of all possible targets from which the developer must make his selection.

Approaches, such as execution traces [4], can provide additional information and narrow down the set of possibly affected locations. Nevertheless, such techniques remain limited with respect to static type information. Also note that the information of a static type system is rarely complete. Most statically typed languages (such as Java, C# and Eiffel) rely on dynamic typing to implement concepts as polymorphism and late binding.

4.3.5 Change Propagation

For dynamic software evolution, languages or methodologies that limit the propagation of a change are desirable. [22] shows that the impact of a change and the effort to execute it are directly proportional: large sets of affected software entities make the update slower and more complex. The influence of the typing mechanism on change propagation is twofold:

1. Static typing prohibits direct modification of types at runtime. Hence, changes often result in the conver-

sion of instances from the old to the new type. It is uncommon that this conversion maintains the identity of the instance. The change therefore propagates to all structures that contain references to the modified instance, resulting in a cascading effect which causes a seemingly local change to have global impact. For purely dynamically typed languages, direct changes to the type are possible. Such changes often preserve object identity and therefore change propagation is less likely.

2. Coexistence of different versions increases localization of a change since lazy object migration is possible: objects are not actively replaced but are gradually removed from the system when they are no longer used. As mentioned in 4.2.2, this is better supported by static type systems.

The popularity of component-based systems for dynamic updating is caused by its excellent characteristics concerning change impact and change propagation. A component is considered to be a loosely coupled and self-sufficient entity. As such, changes within a component remain within the boundaries of the component. In most cases, a lookup mechanism separates the identity of a component from an actual instance. Methodologies such as component based software engineering or aspect oriented programming stimulate the separation of concerns. They reduce coupling, limit change propagation and thus increase the ability to perform runtime adaptations.

4.4 System properties (what)

This dimension of dynamic software evolution refers to a number of system properties of the software system that is being changed and the underlying platform or middleware. It addresses what kind of changes are allowed.

4.4.1 Availability

The goal of dynamic evolution is to minimize downtime. Ideally, the application remains available during the entire adaptation process. In some cases short downtimes are acceptable provided that the new version can continue where the old version left of (i.e. the state is transferred).

Continuous availability of the entire application is not possible in the presence of state transfer. In order to transfer state, the entity under consideration must be placed in an inactive (quiescent) state [15]. Therefore, the application is partly or completely unavailable (depending on the impact of the change) for the duration of the adaptation. The goal is to minimize both the portion of the application that is inactive and the duration of the update. These two issues are interrelated and are coupled to the granularity: a coarse grained unit of change tends to disable a larger portion of the application for a longer period of time. However, replacing a larger unit is much easier (and often safer), since many tightly coupled constructs are replaced together.

Whenever state transfer is not required, an uninterrupted service can be achieved. Direct type modifications (for example adding a method in a Smalltalk program) do not affect running programs and are fully transparent to the user of the application. For statically typed languages, the lazy update approach can be used: since no active entity is replaced, nothing needs to be deactivated. A major drawback of this technique is that there is no guarantee that the update terminates within a reasonable period of time.

4.4.2 Activeness

In literature [21], a distinction is made between reactive and proactive systems. Whereas changes in a reactive system are driven by an external factor, proactive systems monitor the state of their own execution with sensors and adapt themselves according to a given policy. Since such a policy is always intentionally implemented in advance, proactive systems are by definition anticipated.

Unanticipated changes are always initiated on request and therefore reactive. The request is typically directed to the middleware that executes the change. Powerful reflective systems ease the implementation of a reactive environment since they provide an implicit maintenance interface to the application. However, limitations of the environment can be overcome with code instrumentation or manual implementation. In general there is no direct relation with the type system of the language used.

4.4.3 Openness

Software systems are considered to be open if they are specifically built to allow for extensions. With respect to live updates, there is a strong relation between openness and anticipation. Applications are, by definition, closed towards unanticipated changes and open towards anticipated changes. As such, the argumentation given under section 4.3.1 is valid here as well: purely dynamically typed languages are more open due to their powerful reflection mechanism which allow inspection and modification of the language at a very basic level. Additionally, we can say that weakly typed languages will be more open than strongly typed languages.

4.4.4 Safety

A system features *static safety* if it is able to ensure at compile time that the system will not behave erroneously at runtime. A system provides *dynamic safety* if it contains provisions that prevent or restrict unwanted runtime behavior.

The required change support mechanisms are directly influenced by the desired kind and degree of safety.

Static typing provides a certain degree of safety: the system can verify that both the old and the new version are type-safe, and additional analysis can check for type compatibility issues between different versions. It allows, to a certain degree, to verify compatibility at design time or at load time. Without static typing, this is not possible.

At runtime, verification of compatibility can also be performed using techniques as invariant-checking (evaluation of a function that checks the compatibility between the new version and the running system). Such techniques are difficult and time consuming to implement. However, this is not related to the type system since the implementation of this extra behavior is required anyway. Static type systems only detect type errors, which is clearly insufficient for a typical dynamic adaptation scenario. Many additional conditions must be met for the update to be a success (e.g. access control to initiate the update, consistency checks between versions after a state transfer).

4.5 Change support (how)

The final dimension of the evolution-taxonomy describes the change process itself. The actual technique used to accomplish the update is classified according to a number of orthogonal characteristics, some of which are influenced by the programming language used.

4.5.1 Degree of automation

In general, it is impossible to achieve full automation when state transfer is involved: it is impossible to correctly identify semantically equivalent structures between different versions of a component since not all semantic information is contained in source code. A typical example is the representation of a triangle. In version n, three points may be used while version $n + 1$ uses two edges and an angle. State transfer remains in essence an interactive and at best a semi-automatic process. Nevertheless, tool support can be developed to assist the programmer with this task ([32]). Such tools are more likely to be successful if they have type information at their disposal[2].

When no state transfer is required, a powerful meta-level protocol or reflective system could allow fully automatic execution of declarative change-specifications such as the addition or removal of a method. This is an argument in favor of purely dynamically typed languages: it is harder (albeit not impossible) to implement such a system on statically typed languages.

[2]The same statement is valid for other forms of evolution, such as refactoring.

4.5.2 Degree of formality

A change process can be described in various levels of formality. Formal proofs of correctness are extremely useful in the context of dynamic software evolution. After all, an update is worth nothing if not executed correctly. While no such proofs known to the authors are of wide practical applicability, it is not impossible that such formalism could ever be constructed. Although in general statically typed languages are better suited for formal descriptions, other formalisms exist (e.g. graph transformations: see [19]) that do not depend on the type system. Therefore, the degree of formality is not directly related to the type system of the programming language used.

4.5.3 Change type

While *structural changes* are said to be behavioral preserving, *behavioral changes* are said to be changing the behavior of the application. As structural changes are mostly used to ease a behavioral change (e.g. refactorings), live updates can be of both kinds. Because statically typed languages have more structural information, they will be more suited for structural changes then dynamically typed languages. The impact of the typing on behavioral changes is non existent.

4.6 Overview

Table 1 summarizes the findings of this paper. For each of the dimensions of evolution, the impact of the type system is shown. For brevity, dimensions that were not influenced by the typing, were omitted.

In the table we distinguish between six different typing combinations ranging from static/strong to dynamic/weak. The overview confirms the expectations from section 3.4 and demonstrates that more flexible typing corresponds with increased power to carry out the changes. However, this flexibility comes at a cost: reduced impact estimation, lower safety and more difficult formalizations.

This table can be used in combination with the language-overview from the section 3.4 (see figure 2) to assist a developer with the selection of a programming language.

5 Example

To illustrate the influence of typing on evolution, consider the following real-life example. Imagine a complex hierarchy of exceptions and a framework with classes that throw some of those exceptions. The software evolves, and at a certain time a new exception is added to the hierarchy and thrown by the classes from the framework. Static type checking applied to exceptions implies that every thrown

Group	Dimension	Section	Static		Both		Dynamic	
			Weak	Strong	Weak	Strong	Weak	Strong
Temporal (when)	Time of change	4.2.1						
	Offline changes		++	++	++	++	+	+
	Online changes		- -	- -	+/-	-	++	+
	Change history	4.2.2	++	+	++	+	+	+-
	Change frequency	4.2.3	-	- -	-	- -	++	+
Object of change (where)	Anticipation	4.3.1	-	-	+/-	+/-	+	+
	Granularity	4.3.3						
	Coarse grained		++	++	++	++	+	+
	Fine grained		-	-	-	-	++	++
	Impact	4.3.4	+	++	+-	+	- -	-
	Change propagation	4.3.5	+/-	+/-	+/-	+/-	++	+
System properties (what)	Availability	4.4.1	-	-	+/-	+/-	+	+
	Openness	4.4.3	+/-	-	+	+/-	++	+
	Safety	4.4.4	+/-	+	+/-	+	- -	-
Change support (how)	Degree of automation	4.5.1	-	-	+/-	+/-	+/-	+/-
	Degree of formality	4.5.2	+	++	+	++	-	+/-
	Change type	4.5.3	+	+	+	+	+/-	+/-

Table 1. Type impact on software evolution

exception is either caught within the method that throws the exception (which is not always desirable) or declared in the method signature. The latter option causes all subclasses of the modified class to declare the new exception. With statically typed languages, people often avoid these problems by throwing generalized exception classes (`throws Exception`) which are later cast to a more specific exception. The cause of this issue is the type system: with dynamically typed languages these problems do not arise since thrown exceptions do not need to be declared. Therefore, some statically typed languages, such as Java, provide *unchecked exceptions* that literally disables type checking on a number of exceptions. This example clearly shows that there is certainly an influence of typing on software evolution.

6 Related and future work

So far, the relation between programming language features and dynamic evolution has not been systematically investigated. Overviews and comparisons of different systems for dynamic adaptation do exist however, and can be found in [14, 26, 30]. While no programming language is specifically designed with dynamic evolution in mind, Hicks modified the Popcorn programming language in order to provide better support for online evolution ([14]). To achieve this goal, more language constructs were reified into a first-class entity so that they could be inspected and/or modified.

This paper primarily dealt with the impact of the type system. Future work includes an analysis of other language properties that directly or indirectly influence dynamic adaptations. The majority of programming languages uses lexical scoping since this is more intuitive than dynamic scoping. However, [10] shows that dynamic scoping is more flexible and possibly better suited for evolution. Another second important characteristic is the programming paradigm: functional languages do not require state transfer, prototype based languages are flexible and have many possibilities for evolution as they have no classes. Here too, flexibility comes at the cost of reduced encapsulation and security. And while aspect oriented programming aims to provide clear separation of concerns, [29] shows that resulting code is not necessarily loosely coupled.

The execution system of a program has impact on evolution properties as well. Interpreted languages are easier to change at runtime than compiled languages. Languages which are compiled to intermediary code (such as Java or C#) often use Class loading techniques to overcome certain issues. While successful to some extend, they are clearly more complex than their interpreted counterparts [9, 28].

A final language feature that has possible implications on evolution is concurrency. Two major implementations exist to avoid data clashes in multi-threaded systems: the shared data and the actor-model approach. In [2], the authors claim that the actor-model approach is better suited for evolution since concurrency functionality is grouped in one place.

7 Conclusion

Live updates (evolving a program without shutting it down) is a complex and hot topic in software research. So far, no programming language has been designed with dynamic software evolution in mind. Therefore, current techniques all require significant additional effort from the designer and the programmer of the software to compensate for the lack of language support.

The primary contribution of this paper is that it thoroughly investigates the relation between a programming language's type system and its suitability for dynamic adaptation. A taxonomy from renowned experts in the field of software evolution [6] is used as the starting point for our

analysis. Our findings resulted in a concise table that clearly summarizes the relation between a type system and evolution. This table shows that the more flexible the type system, the more power is available to carry out actual changes. It also shows the that these powers come at a cost: reduced security. This table can either be used as a starting point for language designers that want to develop a language for dynamic adaptation, or by an application developer to select a suitable language according to his needs.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] L. E. Alanko. Types and reflections. Master's thesis, University of helsinki, November 2004.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] A. Brodsky, D. Brodsky, I. Chan, Y. Coady, S. Gudmundson, J. Pomkoski, and J. S. Ong. Coping with evolution: Aspects vs. aspirin. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, 2001.

[6] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 2003.

[7] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[8] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[9] S. Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), January 2004.

[10] P. Costanza. Dynamically scoped functions as the essence of aop. In *Workshop on Object-Oriented Language Engineering for the Post-Java Era*, 2003.

[11] P. Ebraert and T. Tourwe. A reflective approach to dynamic software evolution. In W. Cazolla, editor, *In the proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'04) in conjunction with the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, Oslo, Norway, June 2004.

[12] R. A. Finkel. *Advanced Programming Language Design*. Addison-Wesley, 1996.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[16] B. Lientz and E. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.

[17] B. J. MacLennan. *Principles of programming languages: Design, Evaluation, and implementation*. Ted Buchholz, second edition edition, 1986.

[18] P. Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[19] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Proceedings of the First International Conference on Graph Transformation*, Barcelona, Spain, October 2002.

[20] T. Mens and M. Wermelinger. Separation of concerns for software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):311–315, 2002.

[21] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, pages 54–62, May/June 1999.

[22] J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. *In Proc. Int. Conf. Software Maintenance*, pages 163–172, October 2000.

[23] A. Rausch. Software evolution in componentware - a practical approach. In *Proc. of the Australian Software Engineering Conference*, 2000.

[24] D. A. Schmidt. *The structure of Typed Programming Languages*. MIT Press, 1994.

[25] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, 2000.

[26] M. E. Segal and O. Frieder. On-the-fly program modification. *IEEE Software*, 10(2):53–65, 1993.

[27] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, second edition, 1996.

[28] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In R. Crocker and G. L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, USA, 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[29] K. G. Tom Tourwe, Johan Brichau. On the existence of the aosd-evolution paradox. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[30] Y. Vandewoude and Y. Berbers. Overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proceedings of SERP02*, pages 521–527, 2002.

[31] Y. Vandewoude and Y. Berbers. Fresco: Flexible and reliable evolution system for components. In *Electronic Notes in Theoretical Computer Science*, 2004.

[32] Y. Vandewoude and Y. Berbers. Deepcompare: Static analysis for runtime software evolution. Technical Report CW405, KULeuven, Belgium, Februari 2005.

[33] R. Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.