Computer Languages, Systems & Structures 51 (2018) 71–89

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

μ-DSU: A Micro-Language Based Approach to Dynamic Software Updating

Walter Cazzola [a,*], Ruzanna Chitchyan [c], Awais Rashid [b], Albert Shaqiri [a]

[a] Computer Science Department, Università degli Studi di Milano, Italy
[b] School of Computing and Communications, Lancaster University, United Kingdom
[c] Department of Computer Science, Merchant Venturer's School of Engineering, University of Bristol, United Kingdom

ARTICLE INFO

Article history:
Received 14 March 2017
Revised 3 July 2017
Accepted 4 July 2017
Available online 14 July 2017

ABSTRACT

Today software systems play a critical role in society's infrastructures and many are required to provide uninterrupted services in their constantly changing environments. As the problem domain and the operational context of such software changes, the software itself must be updated accordingly.

In this paper we propose to support dynamic software updating through language semantic adaptation; this is done through use of micro-languages that confine the effect of the introduced change to specific application features. Micro-languages provide a logical layer over a programming language and associate an application feature with the portion of the programming language used to implement it. Thus, they permit to update the application feature by updating the underlying programming constructs without affecting the behaviour of the other application features.

Such a linguistic approach provides the benefit of easy addition/removal of application features (with a special focus on non-functional features) to/from a running application by separating the implementation of the new feature from the original application, allowing for the application to remain unaware of any extensions. The feasibility of this approach is demonstrated with two studies; its benefits and drawbacks are also analysed.

1. Introduction

Today software applications govern many of our daily activities: from stock trading to traffic control, and in-body integrated insulin release pumps. Many such applications must provide continuous, uninterrupted services, and if interrupted, could lead to risk to life and safety (e.g., nuclear reactor control software) and substantial financial loss (e.g., software for stock trading).

Yet, as operational environment and expectations on such applications evolve, they do need to be maintained and updated. This problem is well-known, with several *dynamic software updating* (DSU) approaches proposed over the years to try to solve it, e.g., [1–3]. To date, evolution with DSUs is a two-steps process: i) the source code is modified according to the evolutionary needs and ii) the changed code is deployed on the running system using code injection, class reloading

* Corresponding author.
E-mail addresses: cazzola@di.unimi.it (W. Cazzola), r.chitchyan@bristol.ac.uk (R. Chitchyan), marash@comp.lancs.ac.uk (A. Rashid), shaqiri@di.unimi.it (A. Shaqiri).

http://dx.doi.org/10.1016/j.cl.2017.07.003
1477-8424/

and dynamic/multiple method tables. Yet, these solutions inevitably result in long-term performance decay (e.g., JavAdaptor [1,4] and DUSC [5]) or impose limitations on what can or cannot be updated (e.g., JRebel [2]).

In spite of all these efforts, software evolution is still work in progress, especially in supporting the application extension with new non-functional features[1] whose *a posteriori* design and addition is often cumbersome and could be jeopardized during application execution. In literature, we find many examples of this kind of evolution, e.g., for persistence [6,7], parallelization [8,9], optimization [6] and pluggable type systems [10].

To fully understand the problem, consider, for example, a sequential application that could be sped up on a multi-core laptop computer by executing each independent stage of a loop[2] in parallel on a different core. This parallelization is clearly a non-functional feature that can fast the goal achievement but its absence does not compromise its feasibility. The parallelization can be supported by manually (or using an IDE) extracting the body of the loops with independent stages into methods and replacing these loops with the spawning of the corresponding method calls into the available cores. This update must be applied to all the loops which must be parallelised. Several new methods, which have no direct relation to the application logic, must be introduced into the code. This makes the code more obscure and complex. To complicate the scenario, let us consider the fact that we would like to pass from a sequential to a parallel version (which is more energy intensive but quicker) when our laptop is plugged into the mains, and revert back to sequential processing when the laptop runs on the battery. This is a clearly feasible change that most DSUs can support but it can hardly be automated and a continuous manual switch from one version to the other could lead to errors, e.g., forgetting to parallelise some of the loops. Also, though in theory it is feasible to keep two separate versions and swap these at need, in practice it is complicated as one has to synchronize these versions on the number of already executed stages. Alternatively, we advocate in this paper that the same evolutionary change can be realised by changing the semantics of the loop construct in such a way that it implements the spawning of the stages on the different cores without any further cluttering of the original code. This change would affect a single language construct and is less error-prone when passing from one version to the next, as the change is confined to a single element.

Little to no support for change through language evolution exists in literature in particular if it should be deployed during the application execution. This is mainly due to the fact that language implementations are mostly monolithic and do not support this fine-grained kind of language adaptation but also because an indiscriminate change to a language construct implementation would affect any use of such a construct and this is not always desirable (e.g., when parallelising loops in an application that uses both loops with dependent and independent stages). Therefore, to be effective, such an approach should be supported through an easily evolvable language implementation. Whereas the classic language implementations tend to be monolithic and hard to adapt, in the last decade, several modular development frameworks, such as StrategoXT/Spoofax [11], Lisa [12] and Neverlang [13] have been developed. These provide a natural environment for the proposed software evolution via language adaptation based approach.

In this paper we propose *to move the evolution from the application level to the programming language level* to support the dynamic addition of non-functional features which originally were not designed into the application. The basic idea is centred around the facts that

- any application feature is implemented by a subset of the language constructs used to implement that application;
- application feature behaviour is governed by the semantics of the language constructs used in its implementation; and
- the non-functional features introduce additional properties to the existing features (i.e., not stand alone behaviour).

So, changing the semantics of the language constructs used in the implementation of the given feature is directly equivalent to changing the feature itself and provides a simple mechanism to extend it with non-functional features. The obvious benefit of our proposed approach is that the application source code will be left untouched. Moreover, the same evolutionary change can be applied to several code portions through a single implementation, potentially reducing code repetition, scope for error introduction, and limiting application code complexity. Furthermore, in a recent work Chitchyan *et al.* [14] introduced the concept of *micro-language*, which is very well suited for handling the change propagation scoping challenge. Micro-languages provide a logical overlay over programming languages, supporting grouping of language concepts in accordance with some application features.

This proposal is supported by a novel DSU framework, called μ-DSU, that supports software evolution and, in particular, the addition of non-functional features via language evolution and exploits *micro-languages* to confine the effect of the language changes to only the intended application features. The paper provides three major contributions

- a clear definition of micro-languages and their relationship with traditional language terminology (Section 2);
- the definition of an architecture to support dynamic software updating through changes in the language semantics by using micro-languages (Section 3);
- the demonstration of the approach feasibility through the implementation of the μ-DSU framework (Section 4) and showcasing it with two demonstrative studies (Section 5).

[1] In this paper, with non-functional we denote a functionality that could be removed without compromising the achievement of the main goal of the application but that represents a sort of general help to get the goal, such as support for concurrency, sustainability and logging.

[2] A loop with independent stages is a loop whose stages do not rely on the results calculated in the previous stage.

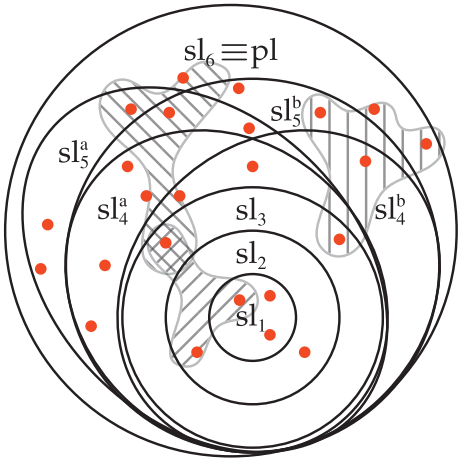


Fig. 1. Relationship among programming (pl), its sub- (sl_i) and micro-languages (the striped blots). Dots represent the different language features.

With μ-DSU a micro-language can be defined for each evolution-prone application feature. As this application feature needs to be evolved this is realised through changes to the implementation of its contained language features. The underlying language implementation for μ-DSU is such, that, when the application feature must change, its micro-language can change without side effects on other micro-languages. Thus, the language environment allows each language element to be updated in a localised fashion, with all the change effects absorbed into the evolving element. μ-DSU is defined on top of the Neverlang [13] framework for modular language implementation.

In the following, Section 2 explains what a micro-language is and its relationship with other programming language concepts. Section 3 shows a generic DSU architecture that exploits language adaptation and micro-languages to support dynamic software evolution. Section 4 introduces μ-DSU that represents a proof-of-concepts for the feasibility of proposed DSU architecture. Section 5 shows the micro-language approach to dynamic software evolution on two demonstrative studies. Section 6 discusses benefits and drawbacks of this approach, while Section 7 presents some related work. Finally, Section 8 concludes the paper.

2. Introducing micro languages

A micro-language is a logical grouping of constituents from its host programming language determined by the programmatic constructs used in a given piece of code (an application feature). Furthermore, a micro-language is a concept orthogonal to those of language features and sub-languages as it can consist of a number of language features from a number of sub-languages (as depicted in Fig. 1). To illustrate this, we will first present the notions of language features and sub-languages and then show how a micro-language relates to them.

2.1. Language features and sub-languages

From a programming language viewpoint, the minimal meaningful concept or construct of a language is called a *language feature* [13]. Examples of language features are the “for” statement, inheritance, or method invocation. As detailed in [13], a *language component* provides the language feature implementation.

Any given programming language can be viewed as the composition of several *sub-languages*, where every sub-language contains a subset of language features of the given (so-called *host*) language. For instance, the SQL language hosts such sub-languages as data definition (which includes, for instance, the create, and drop features), data manipulation (including, for example, select and insert features), transaction boundaries (including, for example, commit and rollback features), etc. In [15], Cazzola and Olivares demonstrated that Javascript can be decomposed into 13 sub-languages.

Sub-languages provide language features to support one (well-defined) *programming aspect* (e.g., setting transaction boundaries in SQL, or dealing with exceptions in Javascript). Yet, to do so they often rely on the presence of other language features provided by other sub-languages (such as creation of a table as a transaction in SQL, or object in Javascript). Therefore, a sub-language needs to be composed with other sub-languages in order to have all its host language features usable. Thus, programming languages could be composed of sub-languages. In either case, a sub-language contains (a number of) language feature(s) that supports one (or more) well-defined aspect(s) of programming. Fig. 1 depicts the relationship between language features (the red dots) and a possible set of sub-languages (sl_i) of a hypothetical programming language (pl); note that each sub-language restricts the programming language limiting the number of included language features.

```
function factorial(n) {
  if (n <= 1) return 1;
  else return n*factorial(n-1)
}
```

Listing 1. Factorial in Javascript.

2.2. Application features and micro-languages

A software application can be described in term of its functionality, often called *application features*.[3] From the application viewpoint, it is irrelevant which language features from which sub-languages are used to implement an application feature. It is rather common to have several language features used in the implementation of a specific application feature, and these language features do not have to belong to a single sub-language. Sub-languages support a single programming aspect but many programming aspects could be required for the implementation of a given application feature.

As an example, let us consider the Javascript implementation for the factorial function (shown in Listing 1). This small piece of code uses—among others—a less equal operator, a conditional statement and a function definition (all language features) belonging to boolean expression, control flow and functional sub-languages, respectively. It is evident that this implementation of the factorial application feature relies on (at least) three sub-languages (out of 13 reported in [15]), and each involved sub-language also contains many more language features than those used to implement the factorial. For instance, the boolean expression sub-language also contains all the other comparison operators, not used in this example.

We define a *micro-language as the set of language features involved in the implementation of a specific application feature*. A micro-language is (normally) orthogonal to any sub-language provided by the host programming language. Should we wish to define a micro-language for factorial calculation in Javascript, it would have to utilise language features from at least the above-noted sub-languages.

A micro-language is a logical definition aimed to align an application feature with the language features used in its implementation. Different application features are bound to different micro-languages that could overlap. Two micro-languages μl_1 and μl_2—associated with the application features af_1 and af_2, respectively—overlap when there exists a language feature *lf* that is used to implement both af_1 and af_2 and therefore it belongs to both μl_1 and μl_2.

Since a micro-language selectively encapsulates the relevant language features from across relevant sub-languages, a single micro-language will rarely ever be also a usable programming language or be able to construct a full application. Instead, a set of micro-languages will be required to represent the various features of the complete application. Note that the union of such a set of micro-languages represents a full coverage of all language features provided by the host language.

Out of one host programming language, a myriad of different micro-languages could be defined, each crossing sub-language boundaries of the host language. It is important to note that micro-languages do not require any specific interpreters/compilers and they do not have to be composed or interfere with each other but only provide a logical overlay on the used programming language that relates part of it to the application features. In Fig. 1, some micro-languages are highlighted by striped blots.

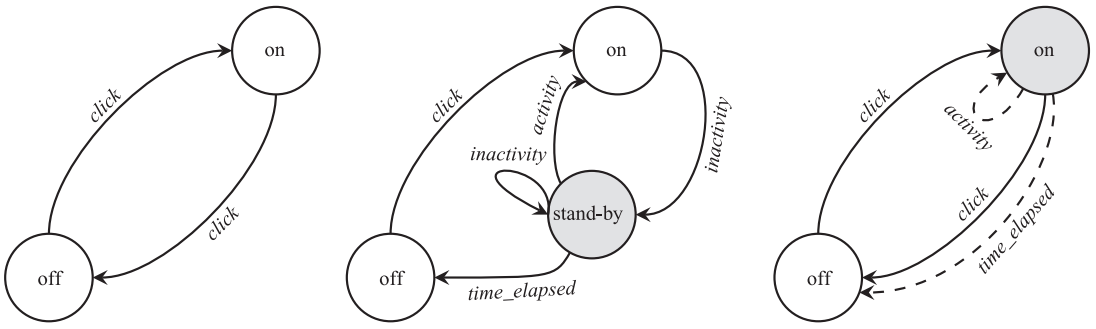
2.3. An Illustrative decomposition in micro-languages

Let us demonstrate the notion of a micro-language with an example for a state machine governing the controller for an automatic vacuum cleaner [14]. The behaviour of any electrical household appliance can be easily modelled with a state-machine: it has some feasible states (e.g., *on, off*, etc.) and some transitions that, under given events (e.g., clicks on a button), move the appliance from one feasible state to another. In our case, the default behaviour simply turns the vacuum cleaner on and off when the switch is turned to "on"/ "off" respectively—as depicted in Fig. 2(a).

The code snippet in Listing 2(a) shows the implementation for the default behaviour of the vacuum cleaner (leaving out the code for moving the vacuum cleaner that it is not relevant to the discussion). At least, two application features are implemented: i) the *turning on* of the vacuum cleaner and ii) the *turning off* of the vacuum cleaner. These two application features are bound to two micro-languages denoted by μ_{ton} and μ_{toff}, respectively. Turn on for the vacuum cleaner consists of changing the `off` *state* into the `on` *state* when a `click` *event* occurs. Looking at the language:

- the definition of a particular *state* is governed by the *state initialization* language feature identified by the **`states`** declaration statement;
- the initialization of the `on` *state* is governed by the *turning on* language feature identified by the **`turn-on`** operation;
- the transition from one state to another is governed by the *transition definition* language feature identified by the **`transitions`** declaration statement.

[3] Czarnecki and Eisenecker [16] defines an application feature as «a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.»



(a) without stand-by (b) with stand-by (c) with merged nodes

Fig. 2. Vacuum cleaner's default behaviour.

```
states
  on  = turn-on
  off = turn-off
events
  click = get_click()
transitions
  on  { click ⇒ off }
  off { click ⇒ on  }
```

(a) default behaviour

```
counter t
states
  on       = turn-on ; t ⇐ 0
  off      = turn-off
  stand-by = t ⇐ t + 1
events
  click      = get_click()
  elapsed    = t > 10
  activity   = get_activity()
  inactivity = get_time()
transitions
  on  {click ⇒ off ; inactivity ⇒ stand-by}
  off {click ⇒ on}
  stand-by {
    click ⇒ off ; elapsed ⇒ off ;
    inactivity ⇒ stand-by ; activity ⇒ on
  }
```

(b) extended behaviour

Listing 2. Vacuum cleaner's code.

Similar considerations apply to the *turning off* application feature with the only difference that the state to be defined is `off` and this is governed by the *turning off* language feature. Thus, the language features describing the two micro-languages are:

$\mu_{ton} = \{$ *turning on, state initialization, transition definition* $\}$
$\mu_{toff} = \{$ *turning off, state initialization, transition definition* $\}$
From this example we see that:

- micro-languages differ from the language used to implement the behaviour of the vacuum cleaner
- a micro-language could be unusable, e.g., neither μ_{ton} nor μ_{toff} include the language features necessary to define the available or specific events without which it is impossible to define a working state machine.
- micro-languages can overlap.

Thus, to re-cap: micro-languages overlay a logical structure on the *host language*; each of them defines and governs the behaviour of a given application feature and straddles several sub-languages from those provided by the host language.

3. Evolution via micro-languages

An application feature is bound to the set of language features needed to implement it that are grouped by a micro-language. Thus, any change that should affect an application feature can be governed and deployed through changes to the respective micro-language.

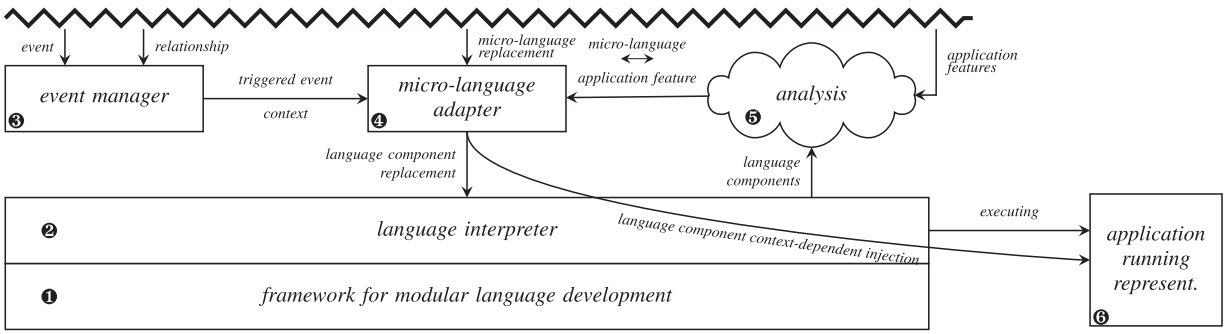


Fig. 3. Architecture for micro-language based adaptation of a running application.

3.1. Driving the adaptation

Fig. 3 graphically overviews the architecture of a DSU framework that would adapt running applications through the adaptation of micro-languages. A modular language development framework (marked with ❶ in Fig. 3) lays the foundation for this architecture and furnishes the necessary mechanism to deal with the micro-language adaptation. The modular language interpreter of the chosen language (marked with ❷ in Fig. 3)—e.g., that of the state machine language used in Sect. 2.3—runs the application that should be evolved (marked with ❻ in Fig. 3)—e.g., the vacuum cleaner control application. Each micro-language will correspond to one application feature. Currently the application features are provided by the application developer and it is represented by a set of coordinates (code lines and element description) to the application source code. A way to automatically elicit from the application analysis both the application features and the language features used in its implementation—and therefore the micro-language associated to the application feature—is currently under investigation. This activity is represented by the component marked with ❺ in Fig. 3.

The evolution of an application feature is implemented through the evolution of the corresponding micro-language. Each change to a language feature in the micro-language affects the application feature behaviour. The adaptation process is event-driven; the *event manager* component (marked with ❸ in Fig. 3) waits for the (application specific and user-defined) *update event*s whose occurrence will trigger the adaptation process. The event manager component deals both with the update events it is to monitor and with the associations between these events and the application features (and hence also the micro-languages) interested in the occurring events. The relationship between occurring events and changes to the application features along with micro-languages, provides the system with the *context* information necessary to limit the effect of the micro-language adaptation. For instance, in the vacuum cleaner example, the inactivity of the running appliance is the event that triggers the adaptation, thereby affecting the *turning on* application feature. The *micro-language adapter* component (marked with ❹ in Fig. 3) is in charge of the adaptation process.

The change at language level, if not limited, will affect the behaviour of the whole application. This can be a useful trait in cases where the required change is general: it has the obvious benefit of a limited adaptation instead of looking for, and individually changing each part of the relevant application features. On the other hand, often an adaptation is desired for a specific, localised, case and such a change should affect only a few application features. Thus, the language adaptation-based framework has to support both system wide and selectively targeted exchange of language features; both types of changes should be done without forcing a regeneration of the interpreter or a mandatory stopping and restarting of the running application. Let us demonstrate these two adaptation scopes using the vacuum cleaner example presented in Section 2.3. The two micro-languages μ_{ton} and μ_{toff} share the *state initialization* language feature implemented by the `State` slice. A system wide change at the interpreter-level to the *state initialization* language feature would affect both micro-languages and the associated application features. Instead, a selective exchange of language features would permit a fine-grain control of the adaptation process. It would permit to subordinate how a language feature (*state initialization* in our case) is interpreted according to the belonging to a specific micro-language. The association of an application feature to its AST permits to directly inject such a semantic action only in the nodes relevant to the μ_{ton} micro-language and to avoid any interference with the same type of nodes that are associated to the μ_{toff} micro-language. This is possible because the application feature to adapt is represented by a specific AST portion; the association with a micro-language permits to identify such an AST portion and to limit the effect of the adaptation only to that portion, granting both a fine-grain control and a context-dependent language adaptation. Therefore, given the context, the adapter will not modify the interpreter, but instead will operate directly on the AST by modifying the semantic actions that should be executed when specific nodes of the tree are visited. Details about how this should work are explained in Section 4.

3.2. Evolution of micro-language semantics

Let us consider the scenario where the requirements are evolved in the vacuum cleaner example presented in Section 2.3. The new version of the system requirements pay more attention to reduction of energy waste. Now the requirement for

switching the system on/off is updated to stating that (for safety and to reduce energy waste) the controller should switch the vacuum cleaner off when it is not in productive use (e.g., when it is accidentally switched on by a toddler). By definition, when in productive use the vacuum cleaner should be in motion; if it is not in motion for 10 seconds, it is considered not in use.

Supporting the behaviour of "turning the vacuum cleaner off when inactive for a given period" at the code-level would require (Fig. 2(b)) addition of new:

- state `stand-by` used when the appliance is on but inactive,
- event `time-elapsed` which is triggered when the specified number of seconds have elapsed without any activity,
- transition from the `stand-by` state to `off` state when the `time-elapsed` event occurs, and
- events handling activity/inactivity of the vacuum cleaner.

Although such code-level adaptation is quite simple and general, it is also very invasive and thus not advisable if it is to be applied to a large set of various types of electronic appliances. The adapted code is shown in Listing 2(b).

When using the micro-languages, instead of modifying the controller program with condition checks for motion monitoring (as per Listing 2(b)), we can change the semantics of some of the language constructs. We call such adaptation *seamless*[4]. In this example the meaning of being in the `on` state can be changed by re-defining the semantics of the *turning on* language feature. In short, the **`turn-on`** operation will not only turn on the appliance but also store the timestamp of when it is turned on. Seamlessly adapted controller behaviour is represented by the state machine in Fig. 2(c) where two (internal) events: `activity` and `time_elapsed` are added, as well as transitions (dashed in the figure to note the fact that they are a sort of a side effect of the changes to the language semantics) from the state `on` to the state `on` (guarded by `activity`) and to state `off` (guarded by `time_elapsed`). The former event will reset the stored time as a consequence of some activity, while the latter will check the current time against the stored time to detect if it is time to turn the appliance off due to prolonged inactivity.

Such a *seamless* adaptation is enabled by changing the implementation of the μ_{ton} micro-language and will leave the controller code unchanged. Therefore, any program written in this language will remain unaltered, yet will incorporate the new behaviour. Moreover, a micro-language provides a context in which the language feature is used; and this context can be used to limit the effect of the change to a specific occurrence of the language feature. Furthermore, if the host programming language is based on a framework for modular language development [12,13,17], adapting a micro-language is only a matter of plugging/unplugging a few language features and in some cases this can be done without regenerating the interpreter.

As shown, language adaptation via micro-languages can ease the application's adaptation. Here the adaptation is moved from the application to the language implementation.

4. μ-DSU: implementation details

As a proof-of-concept, the architecture in Fig. 3 has been implemented into the μ-DSU framework over Neverlang [13].

To provide an usable micro-languages-based approach to software evolution, we must provide an operational environment for it. Modular language development frameworks [11–13,18] emphasise the separation of language features as pluggable and composable units (slices in Neverlang) and they represent a perfect fit when the composition can occur during the application interpretation such as in Neverlang [19,20].

Note that the micro-language concept does not have a direct match in any modular language development framework but it can be defined within the framework concepts. In the Neverlang parlance, a micro-language is a logical cluster of some of the slices used in the interpreter for the host language.

4.1. Neverlang in a nutshell

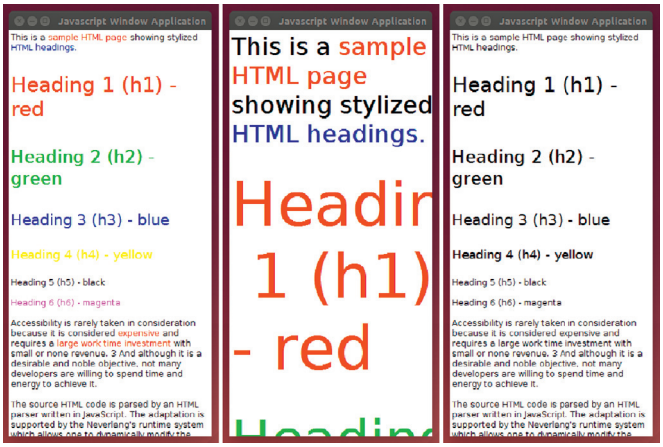
The *Neverlang* [13,21–23] framework is built around the language feature concept. Language components, called *slice*s, embodying the language features are developed as separate units that can be compiled and tested independently, enabling developers to share and reuse the same units across different language implementations. Here the development base unit is the `module` (Listing 3). A module may contain a `syntax` definition and/or a semantic `role`. A role defines actions that should be executed when some syntax is recognized, as prescribed by the *syntax-directed translation* technique [24]. Syntax definitions and semantic roles are tied together using `slices`. Let us see the Neverlang realization of the `State` module for the vacuum cleaner example shown in Listing 3. Here the module `StateModule` declares a reference syntax for the state concept (lines 2–4) and actions are attached to the nonterminals on the right of the production (line 7). Semantic actions are attached to nonterminals by referring to their position in the grammar or through a label: numbering starts with 0 from the top left to the bottom right[5], so the first `State` on line 3 is referred to as 0, `StateName` as 1, and the `Expr` is referred as 2. The slice `State` declares in line 12 that we will be using *this* syntax (which is the `concrete syntax`)

[4] This is in contrast to an *explicit adaptation* that introduces new syntax and forces an application change. This is not further discussed for clarity.

[5] Neverlang also permits to label a production and refer nonterminals via an offset from such a label, e.g., `$STATE[0]` is the head of the `STATE` production.

```
module StateModule {
  reference syntax {
    State ← StateName "=" Expr;
  }
  role(execution) {
    0 { newstate($1.state, $2.action); }.
  }
}
slice State {
  concrete syntax from StateModule
  module StateModule with role execution
}
language HooverLang {
  slices Program StateDecl EventDecl TransDecl StateLst State StateName
      EventList Event EventName Expr BExpr TransList Transition Support
  roles syntax < execution
}
```

Listing 3. Slice the syntax and semantics for the state concept.



(a) normal (b) long-sighted (c) colour-blind

Fig. 4. A sample page displayed according to 3 of the possible profiles.

```
function view(element) {
   var parsed = parse(element); set font color parsed.color;
   set font size parsed.size; print parsed.text;
}
```

Listing 4. JavaScript snippet of the view application feature.

in our language, with those particular semantics (line 13). Finally, the `language` descriptor indicates (lines 17–18) which slices are to be composed together to generate the language interpreter. Composition in Neverlang is, therefore, twofold: (1) between modules, which yields slices, and (2) between slices, which yields a language implementation. The composition result is independent of the order of specified slices. The grammars are merged to generate the complete language parser. Semantic actions are performed with respect to the parse tree of the input program; roles are executed in the order specified in the `roles` clause of the `language` descriptor. Please see [13] for further details.

4.2. μ-DSU over Neverlang

The Neverlang [13] framework permits changes to the interpreted language without regenerating its parser [25]. Changes can be introduced during the interpretation of the application both at system-wide level and localised on the application

Table 1
Summary of the μDA DSL.

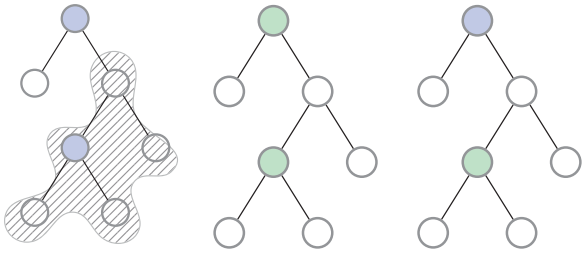
Context Definition
[endemic] slice «id$_1$» [, «id$_2$», ...] : «slc»; *To bind the (endemic) slice «slc» to a name «id$_1$»; if multiple names are provided they are all aliases for the same (endemic) slice.* **production** «id$_1$» [, «id$_2$», ...] : «rule» **from module** «mod» ; *To bind a rule «rule» from a module «mod» to a name «id$_1$»; if multiple names are provided they all refer to the same rule.* **nt** «id$_1$» [, «id$_2$», ...] : «rule» **from module** «mod» ; *To unpack into «id$_1$», «id$_2$», ..., «id$_n$» the first n nonterminals in «rule» from the module «mod».* **action** «id» : «nonterminal» **from module** «mod» **role** «name» ; *To bind the action associated to the «nonterminal» from the module «mod» to the name «id».*
Matching Expressions
«id»[[«cond$_1$(attr$_1$)» [, «cond$_2$(attr$_2$)», ...]]] *Matches the PT node identified by «id» when its (specified) attributes («attr$_i$») verify some conditions («cond$_i$()»). The conditions are predicates that compares the current value of the attribute against a constant; if no condition is given the node is matched by name.* «id$_1$» [[«cond(attr)»]] < «id$_2$»[[«cond(attr)»]]] [| «id»] *Matches a path on the PT where the node «id$_1$» is the parent of the node «id$_2$». As before conditions on the node attributes can be given and the filter operator (|) permits to bind the result to the desired node otherwise it will be the parent.* «id$_1$» [[«cond(attr)»]] ≪ «id$_2$»[[«cond(attr)»]]] [| «id»] *Matches a path on the PT where the node «id$_2$» can be reached from the node «id$_1$». As before, it is possible to express conditions on the node attributes and a filter operator.*
Manipulation Operations
add action «id$_1$» [**to** «id$_2$»] **in role** «name» ; *To add the action «id$_1$» to the node «id$_2$» in the PT for the role «name»; if the target node is omitted the one matched is used.* **remove action** «id$_1$» [**from** «id$_2$»] **in role** «name» ; *To remove the action «id$_1$» from the node «id$_2$» in the PT for the role «name»; if the target node is omitted the one matched is used.* **set specialized action** «id$_1$» [**to** «id$_2$»] **in role** «name» ; *To set the specialized action for the nonterminal «id$_2$» to «id$_1$» in the role «name»; if the target node is omitted the one matched is used.*
System-Wide Manipulation Operations
replace slice «id$_1$» **with** «id$_2$» ; *To replace the slice «id$_1$» with slice «id$_2$».* **redo** [**from** «node»] [**in role** «name»] ; *To restart the visit of the role «name» from a given node «node» as expressed in the context; the visit restarts from the root of the current role when «node» and «name» are omitted respectively.*

parse tree (PT) [20,26]. These Neverlang's characteristics allow μ-DSU to support dynamic language adaptation in two ways: (i) by replacing language components and (ii) by directly modifying how the language feature is interpreted. The former simply consists of replacing a language component (a slice in Neverlang parlance) affecting every use of such a language feature. The latter permits to programmatically modify—according to the micro-language definitions—a language feature locally to a single use of such a feature by altering its behaviour according to the context provided by the PT [20].

Adaptation rules for the micro-languages are expressed through a dedicated *micro-dynamic adaptation* domain specific language (DSL): μDA. μ-DSU implements a *micro-language adapter* (component ❹ in Fig. 3), which executes rules written in μDA, and interacts with the interpreter (component ❷ in Fig. 3) written in Neverlang (component ❶ in Fig. 3) during the application execution. The event manager runs a script that registers the set of possible events and the adaptation scripts that need to be activated in case of a given event occurrence. The event register script has to be maintained by the application developer. The *event manager* component (component ❸ in Fig. 3) notifies the adapter about the events for which specific micro-languages have registered interest. The adapter triggers the respective scripts to deploy the required adaptations for each event per each micro-language.

A generic μDA script has two sections. The first section defines the context for the script, where the PT nodes are described in term of their original productions or in terms of the nonterminal these nodes define; these definitions are bound to names so that they can be referenced through these names elsewhere in the script (the set of constructs used in the context definition is shown in Tab. 1). Each context mirrors a micro-language associated to the given application feature and all the elements of the language features of that micro-language are accessible in the script's context (e.g., see Listing 8(a) lines 1–2). In the context section the actions that should be injected are also defined (e.g., see Listing 8(a) line 4). The code to be injected is written in Neverlang slices which serve as containers for new, replacement adaptation actions. In the case of localised manipulations, the second section declares a set of adaptation clauses to be applied to a specific portion of the PT. A sample structure of such clauses is shown below:

```
when «matching expression» occurs {«manipulation operations»}
```



(a) only `for` (b) only `parfor` (c) `for`+`parfor`

Fig. 5. Selective `for` adaptation.

```
module sustainability.HealthyPrint {
  reference syntax {
    Print: Statement <- "print" Expression;
  }
  role (evaluation) {
    Print: @{
      JSType t = (JSType)$Print[1].value;
      view.print(t.stringValue());
    }.
  }
}
slice Print {
  concrete syntax from sustainability.HealthyPrint
  module sustainability.HealthyPrint
           with role evaluation
}
```

(a) default print slice

```
module sustainability.HyperopicPrint {
  reference syntax from sustainability.HealthyPrint
  role (evaluation) {
    Print: @{
      JSType t = (JSType)$Print[1].value;
      int size = view.getCurrentSize();
      view.print(t.stringValue(), size*3);
    }.
  }
}

slice HyperopicPrint {
  concrete syntax from sustainability.HealthyPrint
  module sustainability.HyperopicPrint
           with role evaluation
}
```

(b) print slice for long-sighted users

```
module sustainability.BlindPrint {
  reference syntax from sustainability.HealthyPrint
  role (evaluation) {
    Print: @{
      JSType t = (JSType)$Print[1].value;
      view.print(t.stringValue());
      view.speak(t.stringValue());
    }.
  }
}

slice BlindPrint {
  concrete syntax from sustainability.HealthyPrint
  module sustainability.BlindPrint
           with role evaluation
}
```

(c) print slice for blind users

Listing 5. Three versions of the implementation for the `print` language feature.

The matching expression resembles an AspectJ's pointcut [27] that match against paths of the PT's application. The matching expressions—described in Table 1—permit to identify a node by its kind (as defined in the context section), by the values of its attributes and by node's relative position in the PT (e.g., if it is a parent of another node). Upon the request from the event manager to the adapter (see Fig. 3), the expression is matched against the whole PT. According to the Neverlang's architecture for dynamic adaptation [20], an agent—whose body is composed of the corresponding manipulation operations—is attached to the matched node and its code will be executed immediately before the next visit of the node that will occur during the interpretation and after that removed. The supported manipulation operations permit to add/remove an action to/from a node with respect to a specific role[6]. In the case of a system-wide adaptation, the manipulation is independent of the PT but it directly affects the slices definitions. Therefore no match against the PT is needed and the manipulation operations are simply introduced by the keyword **`system-wide`**. At the moment, only the slice replacement and the restarting of the PT visit are supported. Table 1 summarises the whole set of operations belonging to μDA; examples of its use are showed in Section 5.

The μ-DSU implementation of the *event manager* is quite simple and its behaviour is governed by bash scripts. The manager is a process that waits for external events or from the Neverlang virtual machine. Each event blocks the interpretation of the application. If the event does not require an adaptation the event manager immediately acknowledges the event and the interpretation resumes. Where adaptation is required, the event and its matching contexts (from the registry script) are passed to the micro-language adapter. When the adapter completes the execution of the adaptation scripts, it notifies the Neverlang virtual machine and the interpretation is resumed from the point of the previous suspension, but using the adapted interpreter.

5. Demonstrative studies

In this section, we showcase feasibility of μ-DSU through two demonstrative studies.[7] In the first study, we show how an HTML viewer application seamlessly adapts its configuration to the accessibility needs of the current user. This study showcases a system-wide change, where the change of the evolving micro-language does not create an interference with other micro-languages. The introduced non-functional feature is the accessibility support. In the second study, we speed up the Mandelbrot set drawing by parallelising its computational engine. This study shows a case of local adaptation, where the evolving language features in one micro-language is also part of another micro-language that should stay untouched,

[6] Neverlang supports several user-defined roles that correspond to consecutive visits of the PT and at each visit, a different action can be executed.

[7] The demonstration of both studies is shown in the movie: http://cazzola.di.unimi.it/%C2%B5-dsu/%C2%B5-dsu-demo.mp4.

Table 2
The language features added to Neverlang.JS.

Language feature	Description
`print Expr`	prints `Expr` to screen
`set font size Expr`	sets the font size to `Expr` points
`set font color Expr`	sets the font colour to `Expr`

so the change must be localised. The introduced non-functional feature is optimization. Together these studies demonstrate the main characteristics of our approach.

5.1. System-wide adaptation

Consider the benefits of having an HTML viewer that adapts page visualization to the conditions of the user's eyesight. For non-impaired users, the viewer would render the page as specified by the underlying HTML code. For users suffering from colour-blindness green and red colours would be substituted; for long-sighted users, a larger font would be used; for blind users, the displayed text would also be read aloud. In several occasions this kind of application would need dynamic adaptations, e.g., digital information booth at a convention where several people with different eyesight needs will continuously alternate or a mobile device that can be passed to a friend with different needs.

As a proof-of-concept, a simple HTML viewer has been developed and μ-DSU is used to dynamically adapt such a viewer to the current user's eyesight. This example demonstrates how micro-languages ease *a posteriori* development and deployment of application variants according to dynamically changed needs either neglected at design time—e.g., due to higher expenses—or newly identified. The viewer has been written in Neverlang.JS [15]: a Neverlang implementation of the Javascript interpreter. Neverlang.JS has been extended with three language features (summarised in Table 2) that help to illustrate and simplify the definition of micro-languages.

Central to the HTML viewer is the application feature of *show text on screen* (AF_{show}) responsible for the HTML text visualization. AF_{show} is implemented by the function `view()` (Listing 4) and uses the language features summarised in Table 2. Thus, these three language features effectively capture a micro-language for the AF_{show} application feature:

$$\mu_{show} = \{\textit{set font color, set font size, print}\}$$

The original application does not consider any accessibility issue. The initial AF_{show} application feature simply parses the HTML element which extracts the colour, size and text from it; then, the colour and size font properties are set according to the parsed values and finally the text is printed. The slice implementing the original behaviour of the `print` language feature of the μ_{show} micro-language is shown in Listing 5(a). As it becomes necessary to support visually impaired users we change the way that AF_{show} visualises the HTML text by changing the semantics for the language element of the μ_{show} micro-language. Listing 5(b) shows the implementation for the `print` language feature of μ_{show} intended for the long-sighted users. The semantic action, among other things, extracts the current size setting which is then multiplied by 3 to increase the size of the visualised text. Similarly, Listing 5(c) shows the implementation of the `print` for blind users. Besides printing the text, this slice also reads it aloud using a specific library to read text. Note that the module does not define any syntax. In fact, the `slice` construct permits to reuse the original `print` syntax defined in Listing 5(a) and to link it to the semantic action for blind users. To save space and since it will not add anything to the discussion, the slice for colour-blind users is not shown.

To support any of the visually impaired users, we simply integrate the relevant changes to the given micro-language in the interpreter by replacing the appropriate modules on need. The original application code never changes, i.e., the application is unaware of the evolving requirements. Moreover, the change is confined in one point—the implementation of the `print` feature—that eases its management. Fig. 4 shows HTML page screenshots visualised with the slices for normal, long-sighted, and colour-blind users (the case of blindness is not shown for obvious reasons).

Let us detail how the process explained in Section 3.1 and summarised in Fig. 3 is applied to this demonstrative study. To allow for runtime adaptation based on user's (dis)ability, (separately from the viewer application) user profiles are associated with their log-in accounts. Let us assume that the user profile change is identified according to the fingerprint/login id provided by the new user when starting to look at the viewer that is already in use. Note, the *profile* changes at system level not at the application level; the application still behaves as usual. The profile change is the event that triggers the adaptation of the application. When the user profile changes, the event manager notifies the adapter component about the new profile and the concerned context which in our case is the AF_{show} application feature. Finally, according to the new profile, the slices associated with the language features in the μ_{show} micro-language are replaced with the version corresponding to the visual impairment of the current user. In this case, the change will affect all uses of the changed language features since the same kind of accessibility support is needed for the whole application. Notice that when, in response to a profile change event, we switch slices in the interpreter or we inject semantic actions into the syntax tree, the original application code never changes, i.e., the application remains unaware of the supported adaptation.

Listing 6(b) shows the μDA script for swapping the current slice for the `print` language feature with the one needed when the user's profile change from `healthy` to `blind`. Notice that, the change is driven by the context definition (the

```
current="healthy"
while true; do
   profile=getprofile()
   if ["$profile" != "$current"]; then
      µda "$current→▹$profile.µda"
      current="$profile"
   fi
   sleep 1
done
```

(a) event manager script

```
slice old: sustainability.HealthyPrint;
slice new: sustainability.BlindPrint;

system-wide {
   replace slice old with new;
   redo role evaluation;
}
```

(b) healthy→▹blind.µda

Listing 6. Scripts for the HTML viewer management.

```
var MAX_ITER = 50; var ZOOM = 450; var HEIGHT = 300; var WIDTH = 300;
var I = create2DArray(WIDTH,HEIGHT);
var zx=0; var zy=0; var cX=0; var cY=0; var tmp=0; var iter=0;
for(var y = 0; y<HEIGHT; ++y) {
    for(var x = 0; x<WIDTH; ++x) {
        zx = 0; zy = 0; cX = (x - 400) / ZOOM;
        cY = (y - 300) / ZOOM; iter = MAX_ITER;
        while ((((zx * zx) + (zy * zy)) < 4) && (iter > 0)) {
            var tmp = (zx * zx) - (zy * zy) + cX;
            zy = (2.0 * (zx * zy)) + cY; zx = tmp; iter = iter - 1;
        };
        I[x][y] = iter | (iter << 8);
    };
};
```

Listing 7. Javascript for the Mandelbrot set calculation.

`new` and `old` names) that is provided by the association between the μ_{show} micro-language and the AF_{show} application feature. The micro-language provides information about which language features (slice in the Neverlang parlance) should be affected by the change whereas the application feature describes where such language features are used limiting, de facto, the effect of the change. Listing 6(a) shows the script that drives the adaptation by executing the μDA script in Listing 6(b) (line 5) when the event occurs (i.e., when the user profile changes).

5.2. Localised adaptation

The second demonstrative study considers the calculation and pictorial drawing of the Mandelbrot set, aiming to speed up the set calculation by parallelising the computation cycles [9,28,29]. As discussed below, we define micro-languages for the application features and the parallelising change is supported through these micro-language change.

Listing 7 shows a traditional Javascript implementation of the Mandelbrot set calculation algorithm. Several application features can be identified in this code, such as: AF_{calc} that calculates the *values* of a point (covering the `while` statement at line 8 with its body), AF_{cols} that cycles on the columns of the final picture (the nested `for` at line 5 with its body but excluding the calculation of the point) and AF_{rows} that cycles on the rows of the final picture (the `for` statement at line 4).

Micro-languages bound to AF_{calc}, AF_{cols}, and AF_{rows} (with the language features denoted by their syntactic symbols) are

$$\mu_{calc} = \{while, =, -, +, *, >, \&\&\}$$
$$\mu_{cols} = \{for, =, -, /, <, ++\}$$
$$\mu_{rows} = \{for, =, <, ++\}$$

As can be noticed above, these three micro-languages are largely overlapped, so a change to one of the language features (e.g., to *for* or $=$) will also affect the other application features if performed system-wide.

The two nested `for` loops in Listing 7 clearly have independent stages (the values of each point are calculated fully independently) and so could be both parallelised. This could be easily done by replacing the slice for the `for` statement with one with a parallel implementation and immediately all the occurrences of the `for` would be parallelised (similar to the system-wide adaptation of the `print` language feature in the previous study (Section 5.1)). But a more interesting

```
nt for1,_,_,_ : For from module neverlangJS.ForLoop;
nt for2,_,_,_ : For from module neverlangJS.ForLoop;

action parforAct : ParFor from module neverlangJS.ParForLoop role evaluation;

when for1 << for2 | for2 occurs {
   set specialized action parforAct to for2 in role evaluation;
}
```

(a) for→▹parfor.μda

```
nt for1,_,_,_ : For from module neverlangJS.ForLoop;
nt for2,_,_,_ : For from module neverlangJS.ForLoop;

action forAct : ParFor from module neverlangJS.ForLoop role evaluation;

when for1 << for2 | for2 occurs {
   set specialized action forAct to for2 in role evaluation;
}
```

(b) parfor→▹for.μda

Listing 8. μDA's scripts to toggle `for` and `parfor`.

case is to see if the change could be localised to *only one* occurrence of the `for` statement—let's say, the nested occurrence which is used by the AF_{cols}—application feature leaving the other occurrence unchanged.

Fig. 5(a) visualises a portion of the PT for the Mandelbrot set calculation showed in Listing 7; the nodes representing `for` statements are purple coloured. This PT is rooted at the node for the first `for` statement (line 4). This `for` statement has four parameters: initialization, condition, increment, and body, all of which are referred to by the node's children. For sake of simplicity, in Fig. 5(a) we draw only two children of this node: the node on the left represents the first three parameters and the tree on the right represents the body of the `for` statement. The body of the `for` statement is a sequence of statements so its first node represent the sequence operator (the `;`)[8]. The second `for` statement (line 5) is represented by the left sub-tree of the sequence operator. The rest of the tree is irrelevant for this discussion and not further considered.

Looking at the PT gives an at-a-glance view of the relations between different statements during the interpretation. It is evident that there is a path in the tree from the node of the first `for` to the second one. Similarly, the definition of the application feature for AF_{cols} and therefore the belonging to the μ_{cols} micro-language provide the same information. Note that an application feature is demarcated as a set of code lines/statements. In Fig. 5(a), the μ_{cols} micro-language is represented with a striped bloat. The effect of substituting any occurrence of the `for` language feature with a parallel `for` is illustrated in Fig. 5(b). In this case, all nodes representing `for` occurrences (the purple ones) are replaced by nodes representing parallel `for` occurrences (the light green ones). Instead to get the desired evolution (Fig. 5(c)) with only the second occurrence of the `for` language feature replaced we have to exploit the context information provided by the μ_{cols} micro-language as done in the μDA scripts reported in Listing 8 where the $<<$ operator (line 6) is used to capture the second `for` occurrence.

The whole experiment binds the speed up/slow down of the Mandelbrot set calculation to the presence/absence of a stable power source. When the laptop runs on the battery it uses only one core to save the battery and the sequential implementation for the `for` statement is used. When the laptop is plugged into the mains, more cores can be used and the sequential `for` is replaced with the version that executes the loop stages in parallel on more cores. The event manager monitors the battery and toggles between the two scenarios (script in Listing 9). Two μDA scripts are used (Listings 8(a) and 8(b)) to deal with the toggle of the language features. Both rely on the context information that comes from the micro-language μ_{cols}. The nonterminals (the head in particular) of the production for the `for` statement are retrieved and used to search the PT for its occurrences. Similarly, the action to inject updated code is retrieved from a given slice and used where necessary, in accordance with the path matching operators. The only real difference between the two scripts is the action to be injected, toggling between the sequential and parallelised scenarios.

To informally evaluate the effectiveness of our run-time adaptation solution, we get the program to calculate a Mandelbrot set of 300×300 points, that is, the external `for` loops 300 times. We run the application in 3 batches of 100 runs each, starting with an unplugged (i.e., single core) state. As the application is running, we plug in the laptop, thus triggering adaptation to multi-core calculation. In the first batch of runs, the laptop was plugged in between the 0th and the 75th lap of the external loop and we obtained a speed up of 44.32% on average. In the second batch of runs, the laptop was plugged in between the 76th and the 150th lap of the external loop and the achieved speed up is 31.32% on average. In the

[8] This is true also when the body is a single statement because Neverlang.JS, for sake of simplicity, does not have the simplification rule for this case.

```
[[ $(acpi -a) == *"off"* ]]; plugged=$?
while true; do
   [[ $(acpi -a) == *"off"* ]]; now=$?
   if [ "$plugged" != "$now" ]; then
      if [ "$now" == true ]; then
        # state changed from plugged to unplugged
        echo "Power saving mode enabled";
        µda parfor—▷for.µda
      else
        # state changed from unplugged to plugged
        echo "Performance mode enabled"
        µda for—▷parfor.µda
      fi
      plugged="$now"
   fi
   sleep 1
done
```

Listing 9. Event manager script to deal with power source.

last batch of runs, the laptop was plugged in between the 151st and the 225th lap of the external loop and a speed up of 20.65% on average was achieved. Plugging the laptop after the 225th lap has been considered insignificant from the point of view of the speed up. As expected, the sooner the parallel implementation of the `for` statement was used in the running application, the larger was the achieved speed up.

6. Discussion

Micro-languages are enablers for dynamic software evolution. Here we discuss the strengths and weaknesses of this approach and we provide an initial evaluation of its benefits.

6.1. Modularity through micro-languages

By definition, a micro-language is conceptually aligned with an application feature and so directly encapsulates the feature concern. Hence, the adaptation of an application feature implies a very localised update to the related micro-language and, to (all or some) occurrences of the corresponding language features. In many cases, the update would affect only a few semantic actions. Furthermore, the modular approach that associates an application feature with a micro-language both limits the effect of the change precisely to the relevant application feature, and confines the required verification/validation for the new code to that single feature. This allows for better scoped and effective verification/validation, with no need to consider the scope of the whole application.

It is also notable that this approach also separates the syntactic definition of control flow of the program from the semantics of the executed computation. For instance, in the above presented examples the same given `print` syntactic statement results in different visualizations of the HTML page, and the given `for` structure of the Mandelbrot set algorithm results in different computation sequences. Thus, a new modularity construct for concern separation at the application feature level is provided.

In short, micro-languages foster modularity as: i) they are built from language features that are modularly implemented; ii) each language feature provides a clear composition interface that allows limiting the ripple effects [30] of changes; and iii) a language feature can be implemented and validated separately of the application that will use it.

6.2. Composability of micro-languages

While the modularity support of the micro-languages is clear and beneficial, it cannot work without adequate composition support. In this approach composition is carried out through micro-language definition and interpretation at event occurrences. Here: (i) each micro-language relates to the events that are relevant to it; (ii) each micro-language defines what will be executed by re-defining the meaning of the language features available in the given language. The composition is the most challenging part of the implementation and it strongly depends on good tool support for correctness checks, usability, and general acceptance. Being a rather novel proposal, at present, this approach has little to offer in terms of development tool support. This can be particularly difficult to handle when the adaptation effects need to be limited by the context provided by the micro-languages. In the absence of good tool support, a misalignment between the application

Table 3
Number of line of code changed for adaptation.

	application-level		micro-language	
	adaptation	variants	language feat.	applicat. feat.
HTML viewer	26	215	57	0
Battery	12	64	48	0

features and the micro-languages could occur and it could have devastating effects with undesired application adaptations due to an unmarked overlapping of micro-languages. Furthermore, a change at language level could introduce new potential vulnerabilities and security risks—a topic that has yet to be investigated.

6.3. On the ease and cost of adapting application features

In order to validate the potential utility of our approach, we must discuss if the effort needed for dynamically adapting a given application through micro-languages is less than that needed for adapting that same application code without the help of micro-languages. For this discussion point, we conducted a simplistic evaluation experiment that draws on the demonstrative examples presented in Sect. 5. It involves five artefacts using different versions of the HTML viewer presented in Section 5.1: the plain viewer, viewer that supports three types of visually impaired users (both versions presented in Section 5.1), viewer that dims the laptop's screen when the battery's charge goes below 50%[9], and two additional equivalent implementations of the HTML viewer except with adaptations for the mentioned visually impaired support and dimming carried out by changing only the application level code (i.e., no change at the language level for the last two implementations) but still on the same variant of Neverlang.JS to keep the implementations comparable.

As a rough approximation for effort, we calculate the number of lines of application code as well as lines of language-level code needed for adaptations in each noted version. To minimise skill-dependency all examples were developed by an experienced Neverlang developer. To recap, we have a common basis (the plain HTML viewer) and two versions of comparable adaptations realised with and without micro-languages support. Table 3 shows the observed results in terms of lines of code (LOC) needed for these adaptations. The data for versions without/with micro-language support is shown on the left/right of the table respectively. The LOCs provided for the micro-languages-based adaptation include changes needed to modify the language and to adapt the application. The LOCs in the case without micro-languages implementation show changes needed to support the dynamic adaptation and those needed to implement the variants.

We observe that *there is a general reduction in the needed LOCs when micro-languages are used; the reduction is more evident at the application-level as expected*. The first row is for the accessibility adaptation; the second one for the dimming support. An average of 19 (26 and 12 respectively) LOCs was needed to provide a minimal structure for dynamic adaptivity without micro-languages. Moreover, about 215 LOCs were necessary to implement four different behaviours of the `print` language feature from Table 2. On the other hand, given an adaptivity framework as described in Fig. 3, adaptation through micro-languages would require no additional LOCs by the application programmer. Instead, the behavioural variants have to be implemented by the language programmer and we observe that about 57 LOCs were necessary to implement the statements from Table 2 and to program the adapter for micro-languages. The main difference is that with micro-languages the adaptation is confined to one point (the `print` statement implementation) and this is transparently adapted (we change the semantic of the statement not its syntax). Whereas without micro-languages, to smoothly support the change the `print` statement has been replaced by a call to the function implementing the behaviour variability and to change each occurrence of the `print` statement represents an extra cost in terms of changed LOCs. The dimming adaptation is more contained in terms of modifications, we have only one variant instead of three as for long-sighted, colour-blind, and blind impediments. This explains the lower modified LOCs (64 for the application features and 48 for the language features). Please note that also using a pure Javascript implementation—i.e., without a `print` statement—for the adaptation at the application level the entity of the change would be of the same extent since we have still to adapt all the calls to the API used to print.

Thus, from our simplistic experiments, we conclude that our adaptation support has promise for reduction of code size and improved localization of change, and modularity. It also proposes a new way of programming and composition, which cannot be accomplished without good tool support.

6.4. Limitations and open challenges

Several aspects of the proposed DSU approach are still challenging: (i) the definition of the existing application features and their association to micro-languages, and (ii) the use of program data—such as method and class names—to refine the area of influence of a language feature change.

Currently, which piece of code is part of an application feature is manually defined whereas the identification of the language features used to implement such a language feature—that is, the definition of the corresponding micro-language—is delegated to a variant of the language parser that recognized the used language constructs. A manual identification of

[9] This draws on the events used in the Mandelbrot example but acts on a different language facet to dim the screen when the laptop is unplugged.

the application features is a tedious and error-prone task and it is hardly maintainable: any change to the program will imply a reclassification of the code in application features that, if not done, would undermine the correctness of the μ-DSU work. Automation of this task is, therefore, desirable and is part of on-going investigations. The easiest way should consist of asking the developer to decorate the code with meta-data about the piece of code that belongs to a particular application feature; similarly to what was done in [31,32]. This approach would help with the maintenance issue but it still requires some manual identification with all the problems this entails. Some more automated approaches to concern mining are under consideration, such as associating an application feature with: each method as in [33], to all modules that calls the same method as in [34], code elements (methods, fields, ...) that present some naming similarities as in [35] or code fragments identical at the AST-level and widely reused as in [36]. All these techniques are automated but focus on crosscutting application features whereas we need to identify any application feature.

Völter [37] defined the concepts of *linguistic* and *in-language abstractions* to denote the language elements—our language features—and program ones—class, field and method names and definitions—respectively. Both are abstractions that can represent update points. μ-DSU focuses on linguistic abstractions but we are aware that some kind of adaptations—such as corrective adaptation—could be more easily achieved by addressing them at program level or at least having some in-language abstractions at disposal. As μ-DSU is a dynamic software update framework, the program is available during its interpretation under the form of PT and the single program data (not the data used by it) as method names would be available through PT attributes. Therefore, in μ-DSU, language adaptation is just the matter of either replacing a slice or injecting a semantic action at a specific node of the PT whereas a method change corresponds to a PT rewriting. Both approaches are feasible and supported by Neverlang as explained in [20] and [26], respectively. Nonetheless, we consider the support of changes at in-language abstractions too complex with respect to the support provided by other DSUs (that do not support changes at linguistic abstractions as μ-DSU does) and we are still investigating how to limit this complexity before introducing in-language abstractions in μ-DSU.

7. Related work

Significant research has been done in the area of adaptive systems. The work here can be categorized as either *architectural* [38,39] or *linguistic* [40,41], or a combination of both [42]. Architectural approaches adapt an application by either adding, removing, or substituting one of its components. Traditional DSUs (such as JavAdaptor [1,4], DUSC [5], Rubah [3] and JRebel [2]) are a variant of architectural approaches where evolution is not just a matter of system reconfiguration but also supports changes in the code. Such DSUs suffer from performance decay (due to indirections introduced bytable forwarding and object proxies) [1,5], limited program adaptation (no class re-positioning [2], either limited—UpgradeJ [43]—or no support for schema changes—HotSwap [44]), misalignment between design and executable code [31,45] and a general difficulty in maintaining the evolved code [46,47].

On the other hand, linguistic approaches support changes to both the application code and its behaviour. The idea to support evolution through ad hoc language constructs was first introduced in [48]. This idea is not orthogonal to the general DSUs approach, rather it allows to get the same evolution, with a better focus on the introduction of non-functional features. Today we have three types of mainstream linguistic approaches: *aspect-oriented programming* [49] (AOP), *reflection/meta-programming* [50] and *context-oriented programming* [51] (COP). Both AOP and reflection provide mechanisms to inject new code into an application yet keep the new code separate from the original one. AOP supports a complete decoupling of the new code (the crosscutting concerns) from the rest of the application code. Very few AO languages—such as CaesarJ [52] and AspectJ's load-time weaving [27]—have any (limited) support for dynamic weaving at all, to update a running system. Despite AOP's poor support for dynamic updating, a few DSUs were developed on top of it [53–55]. As an interesting by-product, μ-DSU—and in particular the adaptation engine provided by Neverlang [20]—could be used as a mechanism to implement dynamic aspect-oriented weaving at the language-level. Meta-programming approaches rely on the reflective features of the programming language and its runtime system. Reflection is used to observe and adapt the underlying program [56]. Chisel [57], PKUAS [58] and mChaRM [59] are examples of frameworks that use Java's reflection facilities to support software adaptation. COP introduces specific language-level abstractions for behavioural variations. Here the context becomes a first-class construct of a programming language [51,60,61]. The system then dynamically selects the appropriate behaviour or a combination of behaviours based on contextual information and selection conditions. The paradigm allows one to separate context-dependent crosscutting concerns. One drawback of this approach is that the computation and coordination aspects are often interleaved and behavioural variations must often be explicitly activated [62]. Moreover, the adaptation is implemented on a per-application basis. Mixins [63,64] and traits [65] are other linguistic approaches to software adaptation and evolution; these approaches allow to extend a class with extra methods and override/enrich already present methods. Matriona [66] is a framework that uses mixins to support dynamic adaptation.

All these approaches enable solutions bound to the adaptation mechanisms provided by the language itself and they can hardly be ported to different programming languages and execution environments. They require adaptation to be explicitly implemented by the application developer in the application space itself. In support of this, Keeney *et al.* [57] argue that *"the software user, the developer, the designer and indeed the application logic itself all possess invaluable intelligence to gear how software should adapt itself to changing requirements and changing context"*. However, although highly valuable, we believe that this intelligence is rarely possessed by *all* the involved actors and thus it requires frequent collaboration between them, which goes against the principle of removing complexity from the lives of users and developers. On the other hand, as

described in our approach, moving the adaptation to the level of micro-languages limits the complexity to a small group of developers which already possesses enough skills to handle the intricacy of complex systems. Furthermore, possible new behaviours affect a small subset of a language and the adaptation through micro-languages provides technical sustainability for free to all applications built upon them.

Kollár and Forgáč [67] investigated the possibility of adapting a programming language interpreter during its execution. The proposed approach is based on AST rewriting—as implemented by Truffle [68]—and code injection at the PT nodes. Their proposal shares with μ-DSU the idea of working on the PT but commonalities end there: μ-DSU does not inject code but uses agents [20] that are dynamically hooked either before or after the PT node. Moreover, μ-DSU confines the effect of the language change to a specific application feature thanks to the micro-languages and, thanks to μDA, it provides an easy way to specify the needed adaptation whereas the approach proposed by Kollár and Forgáč does not foresee any facility to deal with these aspects.

Research on modular language development and evolution is also related to our work. Languages evolve too [69,70] and mechanisms to support their evolution are needed as well. Wassermann and Forgáč [71] survey all possible techniques to adapt domain specific languages implementation. μ-DSU enables *grammar-based adaptation*—in Wassermann and Forgáč's parlance—made dynamic thanks to the Neverlang's parser, called DEXTER [25]. It permits to add and remove a grammar rule from an existing parser without regenerating it and therefore recompiling it. This is one of the key concepts that support language adaptation during the program execution. In the literature, it is possible to find parsers with similar properties as Tatoo's [72] and Silver's [18] even if, in these cases, parser extensibility is limited to rules addition and not to their removal.

Modular language development frameworks although not directly related to software evolution, are enablers for micro-languages adaptation and as such they can represent a valid substitute to Neverlang in the application of the presented idea. Examples of modular language development frameworks are: Lisa [12], StrategoXT/Spoofax [11], Monticore [73], JastAdd [74], MPS [75] and Silver [18]. Some of these are comparatively analysed in [13,76]. They all provide some mechanisms for modularising/composing a language, e.g., via inheritance in Lisa and Monticore, term rewriting in Spoofax and MPS and AST manipulation in JastAdd and MPS. Silver represents the closest approach to Neverlang, even if it is based on functional programming and pure attribute grammars. These characteristics enable all these frameworks to support software evolution via micro-languages. Yet, to our knowledge, none of them supports separate compilation and dynamic AST manipulation. Therefore any micro-language extension will imply recompiling the whole interpreter hindering the possibility of deploying adaptations at runtime. Tatoo [72,77] focuses on grammars composition *de facto* enabling separate compilations but it does not support the modularization and composition of the language semantics: it could be integrated with any other modular language development framework to get the needed semantics support. More in general, any language development framework, such as Xtext [78], could provide micro-languages support to DSU, but modular development frameworks are preferable since they provide a finer change granularity and so are more flexible. Also approaches that enable language modularity through interpreters compositions—such as those approaches that use either *bridge interfaces* [79] or *language boxes* [80]—can be enablers for the micro-language support to DSU when the composed interpreters either have been developed by using modular language development framework or can be modularly modified.

8. Conclusions

This paper presents an approach to dynamic software updating through language semantics adaptation, using the novel concept of a micro-language. Essentially, micro-languages are a new, additional tool for modularising implementation of application features which: (i) aligns the application features with the programming language implementing it; (ii) separates the syntactic definition of control flow of the program from the semantics of the executed computation; (iii) provides a new runtime composition mechanism for programs, thus expanding the toolset available for programming run-time adaptation. The approach has been implemented in the μ-DSU framework on top of the Neverlang modular language development environment. Amongst the tasks that we will address in the immediate future are automation of application feature recovery along with identification of their associated micro-languages. This will extend the present work to deal with emergent and changing application features that would force the adaptation of the application. We will also develop a richer adaptation language, to substitute μDA, based on path temporal logic concepts. Further evaluation that also considers proficiency, security and efficiency aspects of this approach are currently underway.

References

[1] Pukall M, Kästner C, Cazzola W, Götz S, Grebhahn A, Schöter R. Javadaptor — flexible runtime updates of java applications. Softw: Pract Exp 2013;43(2):153–85. doi:10.1002/spe.2107.
[2] Kabanov J, Vene V. A thousand years of productivity: the JRebel story. Softw: Pract Exp 2014;44(1):105–27.
[3] Pina L, Veiga L, Hicks M. Rubah: DSU for java on a stock JVM. In: Millstein T, editor. Proceedings of the ACM international conference on object oriented programming systems languages and applications (OOPSLA'14). Portland, OR, USA: ACM; 2014. p. 103–19.
[4] Pukall M, Grebhahn A, Schröter R, Kästner C, Cazzola W, Götz S. Javadaptor: unrestricted dynamic software updates for java. In: Proceedings of the 33rd international conference on software engineering (ICSE'11). Waikiki, Honolulu, Hawaii: IEEE; 2011. p. 989–91.
[5] Orso A, Rao A, Harrold MJ. A technique for dynamic updating of java software. In: Proceedings of the international conference on software maintenance (ICSM'02). Montréal, Canada: IEEE Press; 2002. p. 649–58.
[6] Chiba S. A meta-object protocol for c++. In: Proceedings of the 10th annual conference on object-oriented programming systems, languages, and applications (OOPSLA'95), Vol. 30 of Sigplan Notices. Austin, Texas, USA: ACM; 1995. p. 285–99.

[7] Rashid A, Chitchyan R. Persistence as an aspect. In: Proceedings of the 2nd international conference on aspect-oriented software development (AOSD'03). Boston, Massachusetts: ACM Press; 2003. p. 120–9.
[8] Gonçalves RC, Sobral JL. Pluggable parallelisation. In: Proceedings of the 18th ACM international symposium on high performance distributed computing (HPDC'09). Münich, Germany: ACM; 2009. p. 11–20.
[9] Harbulot B, Gurd JR. Using AspectJ to separate concerns in parallel scientific java code. In: Proceedings of the 3rd international conference on aspect-oriented software development (AOSD'04). Lancaster, UK: ACM Press; 2004. p. 122–31.
[10] Bracha G. Pluggable type systems. Proceedings of ACM OOPSLA'04 workshop on revival of dynamic languages; 2004.
[11] Kats LCL, Visser E. The spoofax language workbench: rules for declarative specification of languages and IDEs. In: Rinard M, Sullivan KJ, Steinberg DH, editors. Proceedings of the ACM international conference on object-oriented programming systems languages and applications (OOPSLA'10). Reno, Nevada, USA: ACM; 2010. p. 444–63.
[12] Mernik M. An object-oriented approach to language compositions for software language engineering. J Syst Softw 2013;86(9):2451–64.
[13] Vacchi E, Cazzola W. Neverlang: a framework for feature-oriented language development. Comput Lang Syst Struct 2015;43(3):1–40. doi:10.1016/j.cl.2015.02.001.
[14] Chitchyan R, Cazzola W, Rashid A. Engineering sustainability through language. In: Proceedings of the 37th international conference on software engineering (ICSE'15). Track on Software Engineering in Society, Firenze, Italy: IEEE; 2015. p. 501–4.
[15] Cazzola W, Olivares DM. Gradually learning programming supported by a growable programming language. IEEE Trans Emerg Top Comput 2016;4(3):404–15. Special Issue on Emerging Trends in Education. doi: 10.1109/TETC.2015.2446192.
[16] Czarnecki K, Eisenecker UW. Generative programming: methods, tools and applications. Addison-Wesley; 2000.
[17] Bravenboer M, Kalleberg KT, Vermaas R, Visser E. Stratego/XT 0.17. a language and toolset for program transformation. J Sci Comput Program 2008;72(1–2):52–70.
[18] Wyk EV, Bodin D, Gao J, Krishnan L. Silver: an extensible attribute grammar system. Electron Notes Theor Comput Sci 2008;203(2):103–16.
[19] Cazzola W, Shaqiri A. Dynamic software evolution through interpreter adaptation. In: Proceedings of the 15th international conference on modularity (Modularity'16). Málaga, Spain: ACM; 2016. p. 16–19.
[20] Cazzola W, Shaqiri A. Open programming language interpreters. The Art, Science and Engineering of Programming Journal 2017;1(2). 5–1–5–34.
[21] Cazzola W. Domain-specific languages in few steps: the Neverlang approach. In: Gschwind T, De Paoli F, Gruhn V, Book M, editors. Proceedings of the 11th international conference on software composition (SC'12). Lecture notes in computer science 7306, Prague, Czech Republic: Springer; 2012. p. 162–77.
[22] Cazzola W, Vacchi E. Neverlang 2: componentised language development for the JVM. In: Binder W, Bodden E, Löwe W, editors. Proceedings of the 12th international conference on software composition (SC'13). Lecture notes in computer science 8088, Budapest, Hungary: Springer; 2013. p. 17–32.
[23] Vacchi E, Olivares DM, Shaqiri A, Cazzola W. Neverlang 2: a framework for modular language implementation. In: Proceedings of the 13th international conference on modularity (Modularity'14). Lugano, Switzerland: ACM; 2014. p. 23–6.
[24] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. Reading, Massachusetts: Addison Wesley; 1986.
[25] Cazzola W, Vacchi E. On the incremental growth and shrinkage of LR goto-graphs. Acta Inf 2014;51(7):419–47. doi:10.1007/s00236-014-0201-2.
[26] Cazzola W, Shaqiri A. Modularity and optimization in synergy. In: Batory D, editor. Proceedings of the 15th international conference on modularity (Modularity'16). Málaga, Spain: ACM; 2016. p. 70–81.
[27] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold B. An overview of AspectJ. In: Knudsen JL, editor. Proceedings of the 15th european conference on object-oriented programming (ECOOP'01), LNCS 2072. Budapest, Hungary: Springer-Verlag; 2001. p. 327–53.
[28] Cazzola W, Cisternino A, Colombo D. Freely annotating c#. J Object Technol 2005;4(10):31–48.
[29] Cazzola W, Vacchi E. @java:bringing a richer annotation model to java. Comput Lang Syst Struct 2014;40(1):2–18. doi:10.1016/j.cl.2014.02.002.
[30] Greenwood P, Bartolomei T, Figueiredo E, Garcia A, Cacho N, Sant'Anna C, et al. On the impact of aspectual decompositions on design stability: an empirical study. In: Proceedings of the 21st european conference on object-oriented programming (ECOOP'07), LNCS 4609. Berlin, Germany: Springer-Verlag; 2007. p. 176–200.
[31] Cazzola W, Pini S, Ghoneim A, Saake G. Co-evolving application code and design models by exploiting meta-data. In: Proceedings of the 22nd annual ACM symposium on applied computing (SAC'07). Seoul, South Korea: ACM Press; 2007. p. 1275–9.
[32] Sulír M, Nosál' M, Porubän J. Recording concerns in source code using annotations. Comput Lang Syst Struct 2016;46:44–65.
[33] Durelli RS, Santibáñez DSM, Anquetil N, Delamaro ME, de Camargo VV. A systematic review on mining techniques for crosscutting concerns. In: Proceedings of the 28th annual ACM symposium on applied computing (SAC'13). Coimbra, Portugal: ACM Press; 2013. p. 1080–7.
[34] Zhang D, Guo Y, Chen X. Automated aspect recommendation through clustering-based fan-in analysis. Proceedings of the 23rd IEEE/ACM International conference on automated software engineering (ASE'08). L'Aquila, Italy: IEEE; 2008. p. 278–87.
[35] Shepherd D, Pollock L, Tourwé T. Using language clues to discover crosscutting concerns. In: Proceedings of the workshop on modeling and analysis of concerns in software (MACS'05). St. Louis,Missouri, USA: IEEE; 2005. p. 1–6.
[36] Bruntink M, van Deursen A, van Engelen R, Tourwé T. On the use of clone detection for identifying crosscutting concern code. IEEE Trans Softw Eng 2005;31(10):804–18.
[37] Voelter M. DSL engineering. dslbook.org; 2013.
[38] Oreizy P, Medvidovic N, Taylor RN. Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on software engineering (ICSE'98). Kyoto, Japan: IEEE Computer Society; 1998. p. 177–86.
[39] Kramer J, Magee J. Self-managed systems: an architectural challenge. In: Briand LC, Wolf AL, editors. Proceedings of 29th international conference on software engineering (ICSE'07): future of software engineering (FoSE'07). Minneapolis, MN, USA: IEEE Computer Society; 2007. p. 259–68.
[40] Salvaneschi G, Ghezzi C, Pradella M. An analysis of language-level support for self-adaptive software. ACM Trans Auton Adapt Syst 2013;8(2). 7:1–7:29
[41] Galletta L. Adaptivity: linguistic mechanisms and static analysis techniques, Pisa, Italy: Università degli Studi di Pisa; 2014. [Ph.D. thesis].
[42] Salehie M, Tahvildari L. Self-adaptive software: landscape and research challenges. ACM Trans Auton Adapt Syst 2009;4(2). 14:1–14:42.
[43] Bierman G, Parkinson M, Noble J. Upgradej: incremental typechecking for class upgrades. In: Vitek J, editor. Proceedings of the 22nd european conference on object-oriented programming (ECOOP'08). Lecture notes in computer science 5142, Paphos, Cyprus: Springer; 2008. p. 235–59.
[44] Dmitriev M. Towards flexible and safe technology for runtime evolution of java language applications. In: Cahill V, Clarke S, Dobson S, Filman R, editors. Proceedings of the 1st workshop on engineering complex object-oriented systems for evolution (ECOOSE'01). FL, USA: Tampa Bay; 2001. p. 14–18.
[45] Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: bridging the gap between design and implementation. In: IEEE Trans Softw Eng, vol. 27; 2001. p. 364–80.
[46] D'Hondt T, De Volder K, Mens K, Wuyts R. Co-evolution of object-oriented software design and implementation. In: Akşit M, editor. Proceedings of the international symposium on software architectures and component technology. Twente, The Netherlands: Kluwer; 2000. p. 207–24.
[47] Ebraert P, Vandewoude Y, D'Hont T, Berbers Y. Pitfalls in unanticipated dynamic software evolution. In: Cazzola W, Chiba S, Saake G, Tourwé T, editors. Proceedings of ECOOP'2005 workshop on reflection, AOP and meta-data for software evolution (RAM-SE'05) Glasgow, Scotland; 2005. p. 3–8.
[48] Mens T, Wermelinger M, Ducasse S, Demeyer S, Hirschfeld R, Jazayeri M. Challenges in software evolution. In: Proceedings of the 8th international workshop on principles of software evolution (IWPSE'05). Lisbon, Portugal: IEEE Press; 2005. p. 13–22.
[49] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier J-M, et al. Aspect-oriented programming. In: 11th european conference on object oriented programming (ECOOP'97). Lecture notes in computer science 1241, Helsinki, Finland: Springer-Verlag; 1997. p. 220–42.
[50] Maes P. Concepts and experiments in computational reflection. In: Meyrowitz NK, editor. Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Vol. 22 of Sigplan Notices. Orlando, Florida, USA: ACM; 1987. p. 147–56.
[51] Hirschfeld R, Costanza P, Nierstrasz O. Context-oriented programming. J Object Technol 2008;7(3):125–51.
[52] Aracic I, Gasiunas V, Mezini M, Ostermann K. An overview of caesarj. Trans Asp-Oriented Softw Dev 2006;1(1):135–73.

[53] Yang Z, Cheng BHC, Stirewalt REK, Sowell J, Sadjadi SM, McKinley PK. An aspect-oriented approach to dynamic adaptation. In: Garlan D, Kramer J, Wolf AL, editors. Proceedings of the 1st workshop on self-healing systems (WOSS'02). Charleston, SC, USA: ACM; 2002. p. 85–92.
[54] Greenwood P, Blair L. A framework for policy driven auto-adaptive systems using dynamic framed aspects. In: Transactions on aspect-oriented software development, vol. 2. Springer; 2006. p. 30–65.
[55] Zhang G, Rong M. A framework for dynamic evolution based on reflective aspect-oriented software architecture. In: Proceedings of the 4th international conference on computer sciences and convergence information technology (ICCIT'09). Seoul, South Korea; 2009. p. 7–10.
[56] Nierstrasz O, Gîrba T. Lessons in software evolution learned by listening to smalltalk. In: Leeuwen J, Muscholl A, Peleg D, Pokorný J, Rumpe B, editors. Proceedings of the 36th conference on current trends in theory and practice of computer science (SOFSEM'10), LNCS 5901. Špindlerův Mlýn, Czech Republic: Springer; 2010. p. 77–95.
[57] Keeney J, Cahill V. Chisel: a policy-driven, context-aware, dynamic adaptation framework. In: Proceedings of the 4th international workshop on policies for distributed systems and networks (POLICY'03). Como, Italy: IEEE; 2003. p. 3–14.
[58] Huang G, Mei H, Yang F-Q. Runtime software architecture based on reflective middleware. J Inf Sci 2004;47(5):555–76.
[59] Cazzola W. Remote method invocation as a first-class citizen. Distrib Comput 2003;16(4):287–306. doi:10.1007/s00446-003-0094-8.
[60] Keays R, Rakotonirainy A. Context-oriented programming. In: Banerjee S, Cherniack M, editors. Proceedings of the 3rd ACM international workshop on data engineering for wireless and mobile access (MobiDe'03). San Diego, CA, USA: ACM; 2003. p. 9–16.
[61] González S, Mens K, Colăcioiu M, Cazzola W. Context traits: dynamic behaviour adaptation through run-time trait recomposition. In: Kienzle J, editor. Proceedings of the 12th international conference on aspect-oriented software development (AOSD'13). Fukuoka, Japan: ACM; 2013. p. 209–20.
[62] Salvaneschi G, Ghezzi C, Pradella M. Context-oriented programming: a software engineering perspective. J Syst Softw 2012;85(8):1801–17.
[63] Bracha G, Cook W. Mixin-based inheritance. In: Yonezawa A, editor. Proceedings of the european conference on object-oriented programming on object-oriented programming systems, languages, and applications (OOPSLA/ECOOP'90). Ottawa, Canada: ACM; 1990. p. 303–11.
[64] Flatt M, Krishnamurthi S, Felleisen M. Classes and mixins. In: MacQueen DB, Cardelli L, editors. Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on principles of programming languages (PoPL'98). ACM; 1998. p. 171–83.
[65] Schärli N, Ducasse S, Nierstrasz O, Black AP. Traits: composable units of behaviour. In: Cardelli L, editor. Proceedings of the 17th european conference on object-oriented programming (ECOOP'03). Lecture Notes in Computer Science 2743, Darmstadt, Germany: Springer; 2003. p. 248–74.
[66] Springer M, Niephaus F, Hirschfeld R, Masuhara H. Matriona: class nesting with parametrization in squeak/smalltalk. In: Batory D, editor. Proceedings of the 15th international conference on modularity (Modularity'16). Málaga, Spain: ACM; 2016. p. 118–29.
[67] Kollár J, Forgáč M. Combined approach to program and language evolution. Comput Inform 2010;29(6):1103–16.
[68] Würthinger T, Wöß A, Stadler L, Duboscq G, Simon D, Wimmer C. Self-optimizing AST interpreters. In: Warth A, editor. Proceedings of the 8th symposium on dynamic languages (DSL'12). Tucson, AZ, USA: ACM; 2012. p. 73–82.
[69] Favre J-M, Yamamoto S-i. Languages evolve too! changing the software time scale. In: Saeki M, Canfora G, editors. Proceedings of the 8th international workshop on principles of software evolution (IWPSE'05) Lisbon, Portugal; 2005. p. 33–42.
[70] Cazzola W, Poletti D. DSL evolution through composition. Proceedings of the 7th ECOOP workshop on reflection, AOP and meta-data for software evolution (RAM-SE'10). Maribor, Slovenia: ACM; 2010.
[71] Wassermann v, Forgáč M. Adaptation techniques of domain-specific languages. In: Proceedings of the 10th scientific conference of young researchers (SCYR'10) FEI TU of Košice, Košice, Slovakia; 2010. p. 370–3.
[72] Cervelle J, Forax R, Roussel G. A simple implementation of grammar libraries. Comput Sci Inf Syst 2007;4(2):65–77.
[73] Krahn H, Rumpe B, Völkel S. Monticore: a framework for compositional development of domain specific languages. Int J Softw Tools Technol Transf 2010;12(5):353–72.
[74] Hedin G, Magnusson E. Jastadd — an aspect-oriented compiler construction system. Sci Comput Program 2003;47(1):37–58.
[75] Völter M, Pech V. Language modularity with the MPS language workbench. In: Proceedings of the 34th international conference on software engineering (ICSE'12). Zürich, Switzerland: IEEE; 2012. p. 1449–50.
[76] Erdweg S, van der Storm T, Völter M, Tratt L, Bosman R, Cook WR, et al. Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput Lang Syst Struct 2015;44:24–47.
[77] Cervelle J, Forax R, Roussel G. Tatoo: an innovative parser generator. In: Gitzel R, Aleksy M, Shader M, editors. Proceedings of the 4th international symposium on principles and practice of programming in Java (PPPJ'06) Mannheim, Germany; 2006. p. 13–20.
[78] Efftinge S, Völter M. oAW xtext: a framework for textual DSLs. Proceedings of the eclipsecon summit europe 2006 (ESE'06) Esslingen, Germany, vol. 32; 2006.
[79] Grimmer M, Seaton C, Würthinger T, Mössenböck H. Dynamically composing languages in a modular way: supporting c extensions for dynamic languages. In: Leavens G, editor. Proceedings of the 14th international conference on modularity (Modularity'15). Fort Collins, CO, USA: ACM; 2015. p. 1–13.
[80] Barrett E, Bolz CF, Tratt L. Approaches to interpreter composition. Comput Lang Syst Struct 2015;44:199–217. (Part C).