Anticipating Scientific Software Evolution as a Combined Technological and Design Approach

Catherine Letondal
Computing Center
Institut Pasteur
Paris, France
letondal@pasteur.fr

Uwe Zdun
Department of Information Systems
Vienna University of Economics
Vienna, Austria
zdun@acm.org

Abstract

Evolution in scientific software is often according to a specific pattern of software changes: professional scientists, who are not professional software developers, need rapid, dynamic, and domain-specific changes of the software they work with. To address unanticipated software evolution in this field, our objective is to enable these end-users (here: biologists) to change software from the user interface. An approach is presented that integrates technological and methodological solutions. We explain why these solutions are complementary, and how they can be integrated and co-evolved from software design to actual use.

1 Introduction

This paper addresses the issue of software evolution in the context of scientific software. In this field, software tools must be rapidly adapted to new scientific ideas, situations, and results; software evolution and flexibility thus are fully inherent to the field. Many software engineering approaches primarily cope with software evolution issues by iterations of software development or maintenance cycles. In scientific research, however, software tools not only must adapt to a fast evolving domain, but they also must be very adaptable to different users and their particular work tasks. As a consequence, the notion of software evolution has to be extended to *customization* and *end-user programming*, or even, as we show later, to full programmability.

We will discuss these issues with practical examples from the biology domain and experiences with our prototype, called biok [13]. On the technological side, the prototype relies on the language XOTcl [22] that provides dynamic and reflective language functionalities. These proved to be extremely efficient in the design of biok by enabling us to weave *using* and *programming* as two levels of interaction on the user side.

The complex and fundamental aspect of flexibility in scientific software, however, lead us to consider not only technological solutions for software evolution, but also methodological ones. These enable us to address software evolution *at the design level*. Or, to be more precise, we have observed that technical and design approaches benefit from being used together. Conceptually our approach mainly draws from two fields:

- *User-centred design and participatory design methods* [7]: These design methods enable users to actively participate in the early stages of software development. That helps the developers to anticipate both critical features of the software and dimensions that may be subject to evolution.
- *Reflective systems* [18] and *meta-object protocols* [11]: We have explored protocols (both intercessory and introspective) to open the system and make it a "white box," as well as meta-constructs for enhancing unanticipated user control inside the functional part of our system.

Methodological and technological parts of our approach are deeply tangled, as we try to explain in this paper. This does not mean however that the end-user has to understand the meta-level functionalities.

In this paper, we first describe some important characteristics of software development and evolution in biology, as well as situations where biologists who are not professional programmers may need to change the software they use. Next, we introduce the idea of programmability, or how to enable programmatic changes for the end-user within the user interface. We also present the participatory design approach, or how to let the user actively participate in the design. Next, we describe our prototype biok and the underlying language, XOTcl, and discuss how their reflective architecture permits a significant range of software evolutions. Finally, we conclude on how to combine user-centered design and technological approaches to achieve development cycles supporting software evolution at each design, development, and use step.

2 Scientific Software Evolution

The evolution of a scientific tool, in addition to changes that occur in any software, follows a somewhat specific pattern. We describe it by firstly indicating some characteristics of software development in biology that might explain why and how biologists, as opposed to software engineers, are concerned with the activity of software development. We also discuss some scenarios of programming situations and some fundamental reasons why biologists should be able to program themselves.

2.1 Software Development in Biology

As we were able to observe by having installed scientific software for several years at the Pasteur Institute, and by having taught to biologists how to use the scientific software, software development in the field of academic biology research follows two main lines:

- large-scale projects such as [23], development in important bioinformatics centers such as the US National Center for Biotechnology Information (NCBI) or the European Bioinformatics Institute (EBI), or research in algorithmics by researchers in computer science;
- local developments by biologists who have learned some programming but who are not professional developers (used either to deal with everyday tasks for managing data and analysis results, or to model and test scientific ideas).

The two lines often merge, since biologists also contribute to open-source projects and distribute the software they have programmed for their own research in public repositories. We focus on the second type of development that shows the following characteristics:

- *Very dynamic software activity:* Scientific ideas need to be put into an "active" model and tested on other researcher's data. This sometimes requires a hands-on approach of software development, the researcher cannot always wait for a professional software developer to produce a prototype or software adaptation first, before testing new ideas.
- *Majority of programs developed by domain experts:* Most of the software in the scientific area is not produced and evolved by professional programmers, but by domain experts, which is a sign that it is the biologist who needs to program.
- *Software production is not the goal:* In scientific research, software is created to test new ideas rapidly, thus its production does not always follow typical software engineering life-cycles. Except for large-scale projects, programming and building software is very often not the goal of the biologist.
- *Domain-specific evolution:* Software evolution happens usually rather on a level of domain semantics than on the level of the internal software structures of the software tool (see also the examples provided in the next section).

These characteristics indicate that there is a need for programming in everyday biology research but also that this activity is not central as a professional objective.

2.2 Examples of Programming Situations

Recent increase in the use of biological computing, mainly due to research on genomics, means that biologists have to manipulate a lot of data and programs to do their research. This leads to problems for biologists who do not program at all. Below is a list of real programming situation examples required when working with either DNA or protein sequences (a sequence is a molecule that is very often represented by a character string, composed of either DNA letters – A, C, T, and G – or amino-acid letters – 20 letters):

- *Scripting:* search for a protein sequence pattern, *then* retrieve all the corresponding secondary structures in a database.
- *Parsing:* search for the best match in a database similarity search report *but relative to each subsection*.
- *Formatting:* renumber one's sequence positions from -3000 to $+500$ instead of 0 to 3500.
- *Variation:* search for patterns in a sequence, *except repeated ones*.
- *Finer control on the computation:* control in what order multiple sequences are compared and aligned.
- *Simple operations (not available or predefined in the user interface or software tool):* search in a DNA sequence for the characters other than A, C, T, and G.

These examples show that, in spite of the fact that there are already many software systems for biological computing, unforeseen change requirements may arise at any time. In these situations, programming is needed, even though all example situations correspond to quite simple programs.

The examples illustrate some fundamental reasons why biologists would need to program; in particular:

- Scientific programs represent ideas that evolve and need to be refutable.
- It is easier to think directly in the medium in which the problem is generally expressed [4].
- There is a critical and general need for flexibility: hence the use of flexible, informal tools such as spreadsheets and text processors to support data analysis [21].

3 Programming and Using

The majority of biologists do not program, although, as discussed in the previous section, they would really need it for many of their particular work tasks. There are several ways to address the problems of programming, when user-specific customizations are required: a biologist can either learn programming or hire a programmer. The second solution is not always feasible, and both solutions rather have high costs. Regarding the first solution, there are actually many biologists who have successfully learned how to program, although very few actually do. Our hypothesis is that *it is difficult to program a little*, not only in the sense of programming *occasionally*, but also in the sense of programming *incrementally*.

In this section, we discuss end-user programming as an alternative solution with certain limitations regarding unanticipated changes. Programmability for end-users addresses these limitations.

3.1 End-User Programming

End-User Programming (EUP) approaches [15] basically rely on the idea that, since the user already knows the user interface, it can be used as an "indirect" programming language. While using the software, the user may define a kind of program. This "program" can be simple behavior specification that can be reused later on. Or it can be a more complex software artifact, such as a database request or a grammar rule [16].

The first problem in this approach, although very powerful and not enough applied, lies in its lack of generality. The elements of the user interface language (like button click, line drawing, typing, selection, etc.) have to be prepared in a way or another to be used as a means for building the "program." Any possible kind of change has to be anticipated by the designer of the user interface language.

The main problem is that these techniques are not really designed for software evolution but rather for end-user program building. The scope of changes, if any, is often very limited: for instance, in a spreadsheet, you cannot change the behaviour of the spreadsheet itself. According to [19], the only type of evolution that is dealt with by EUP approaches is integration, as opposed to extension or modification.

3.2 Programmability for the End-User

What is needed for software development in biology is both software evolution and EUP. Meta-object protocols (MOP) [11] provide a conceptual framework for modifying the internals of a system. The main idea is that a base-level component that is subject to evolution should be *reified* to offer a proper interface for modification, a *meta-level* interface. A metaphor that can be used for this is a theater, where on-stage represents the system behaviour, and back-stage, the place where everything on the scene is defined, i.e. the system definition. A MOP does not only open the back-stage, but makes it an "on-back-stage" where it is possible to redefine the system behaviour in a constrained way.

To make software evolution accessible for biologists by opening the back-stage, we have to require several aspects:

- As in a MOP, the system must contain explicit and documented places that can be subject to change.
- Since the user in our context is not a professional programmer, there is a critical need for tools to help associate user interface elements with system elements (in the theater metaphor, there must be some threads between the stage and the back-stage to show what in the back-stage controls the elements on the stage); this helps to reduce the cognitive distance between the code and the user interface [20].
- The user of our system does not have any particular interest in programming: his or her task is to perform research in biology. What we want for the biologist is just to have the same situation as for the programmer, where:

  $$\textit{work environment} = \textit{programming environment}.$$

  For the biologist, this means that the data analysis environment should be at the same time a programming environment. This way, no switch from a mode to another is required, programming is just another kind of using. This is the idea of Programming In The User Interface (PITUI) [8].

These aspects have been explored in the prototype, described in this paper, and are detailed in further sections. Note that our approach is not to let the user use the MOP functionalities directly. Instead we use the MOP functionalities in the prototype to enable simple, localized programming for the user. These user-centered software evolution functionalities are not only embedded technically into the user's working environment, but also methodologically (as described in the next section).

Another key idea of MOPs is the idea of a having both a generic and a default behaviour. Unlike frameworks, systems provided with a MOP are ready-to-use and do not need any specialization in order to be usable. For end-users, this means that the user should not have to program to be able to use the system. Programming and customization should be possible, but not required. Furthermore, the available system can be used as a set of working examples [17]. In Section 7, we describe how the whole system is made available, not just as raw source code, but in a *structured* way using the introspection techniques provided in XOTcl (which are described in Section 6).

4 User-centered and Participatory Design

The user-centered and participatory design approaches, described for instance in [7], build on this rather self-understandable idea that the more you involve the users of the software in the software building process, the more your system will be adapted to the user's requirements. But it can be observed that often user meetings do not

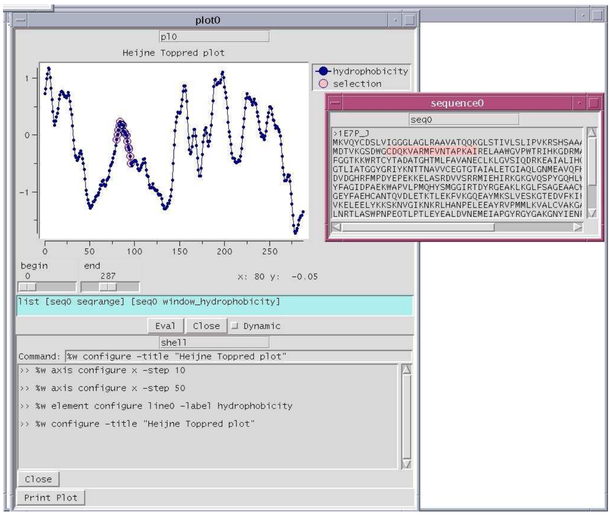

Figure 1: Two graphical objects: the shell of the plot object is opened and the user has entered commands to customize the Tk widget.

provide precise enough information. Thus this approach also suggests that user participation should not only let the user describe what (s)he wants. Several methods can be applied in order to make the user's involvement more active:

- *Brainstorming:* As opposed to a meeting, brainstorming enables us to explore the design space; in a brainstorming session, users are asked to be inventive and to suggest infeasible, unrealistic, or even "stupid" ideas.
- *Interviews and scenarios:* The designer can get a lot of information that the user would have summarized and idealized in a meeting by being able to observe actual tasks at the user work place. For instance, designers can observe the use of existing software. Videotaping the screen can provide data for further analysis of the task. An interview may be an opportunity for the designer to capture a use-scenario that can be used later in prototyping sessions.
- *Mockup prototyping:* In a prototyping workshop a fake software is built with paper, pens, tape, etc. That is, material is used which is both dynamic and familiar to the user (as opposed to programs and abstract diagrams). For example, you can play a scenario by moving paper windows or coloring parts of the printed data without being a professional programmer. The benefit of this approach is to enable the user to show (instead of describe) what (s)he wants within a realistic scenario. Designers still can detect potential ambiguities.

5 Biok: a Biological Interactive Object Kit

Biok is a *prototype* of a programmable environment for sequence analysis. It is written in XOTcl using TK as a graphical toolkit. In biok, the basic building block for both using and programming is a *graphical object*. Graphical objects have a name, which acts as a global variable, and an area for application widgets. Object content may be defined by a formula. For instance, the formula of the plot object in Figure 1 is just defined by a method of the protein sequence to compute the hydrophobicity. Whenever the sequence is changed, the formula is recalculated (the user can control this by a switch). Graphical objects are provided with a shell to run methods (see Figure 1).

One of the central tools of biok is a spreadsheet specialized in displaying and editing sequences. This tool provides visualization mechanisms, as well as the 3D molecule displayer (see Figure 2 and 3). The two objects display structural features of the protein, such as helices and sheets (this visualization function was recently developed by a biology student).

As in many other scientific research areas, visualization is really critical in biology. For this reason, several visualization functionalities are already pre-defined. There is a general *tag* framework to let the user define relations

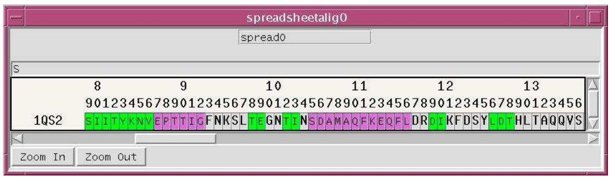

Figure 2: A spreadsheet specialized in editing and visualizing sequences: a secondary structure is highlighted.

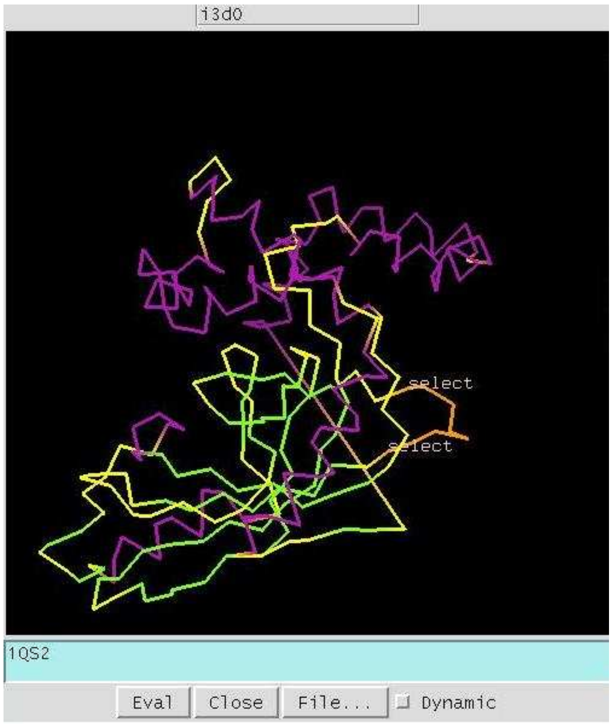

Figure 3: Molecule viewer: the secondary structure is highlighted accordingly.

between domain values, as well as graphical attributes and positions in the different visualization tools. Tags are edited in a specialized editor, where the user has to define a method to associate tag values to data positions in a script. This method is typically either a small script for simple tags, or an invocation of more complex operations that run analyzes, for instance from a Web Server [14].

A set of tags is already defined at the spreadsheet level (with tags for columns, rows, or cells) and at the biology level (particularly a tag to highlight parts of sequences). The user can create sub-classes of tags: for instance in Figure 4 a user wants to highlight specific patterns in front of another visualized tag showing transmembrane segments. The latter tag has been implemented by a biology student, who had only a programming experience of (rather unsuccessfully) learning Python for one month.

6 Dynamic Introspection and Interception in XOTcl

Biok uses a language called XOTcl [22]. XOTcl is an object-oriented scripting language. As a Tcl extension, it is a full-fledged programming language. Scripting languages, such as Tcl, Python, or Perl are often used in the context of end-user programming and programmability to rapidly handle changes that are hard to anticipate. XOTcl specifically adds some high-level, object-oriented functionalities to enable these tasks. In the remainder of this section, we explain some of these functionalities of XOTcl with simple examples. The described language functionalities

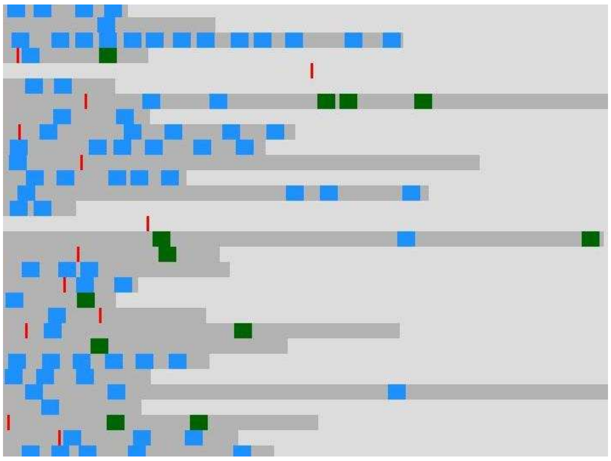

Figure 4: Tag sub-classing: a tag showing transmembrane segments (in blue and green) is augmented by a sub-tag highlighting small patterns around them (in red).

are used for enabling end-user programmability in biok and are also used in the internal implementation of biok.

In XOTcl an automatic type conversion system is used to let programmers only see one type (strings) in the scripts, and they do not have to care for further type conversion issues. XOTcl scripts do only use strings and do not expose the necessary conversion code. All XOTcl objects and classes are addressable at runtime with string-based IDs. The IDs are also be converted automatically to the respective class implementing the functionality.

This internal architecture of XOTcl can be used for incremental program manipulation. Classes and objects can be incrementally defined and modified at runtime because code is treated as data and can be evaluated by XOTcl dynamically. For instance, we can define a new class at any time by evaluating the following script:

```
Class C1
```

To this new class `C1` (and to all existing classes) we can dynamically add method definitions at any time, for instance:

```
C1 instproc calc {a b} {
  expr $a + $b
}
```

We have added a simple instance method that calculates a sum. Upon a change, we can dynamically change the method implementation, e.g.:

```
C1 instproc calc {a b} {
  set r [expr $a + $b]
  expr [$r - 0.25 * $r]
}
```

We calculate the same sum but lessen it by 25 per-cent. Dynamically, we have redefined the method, and whenever an instance of `C1` calls the method, the new implementation is invoked.

User-defined scripts (e.g. in the biology domain) are often of similar simplicity as the examples above. In a tool such as biok we need to connect such scripts to the environment, and the user should not have to understand the whole biok system in order to provide customizations. Introspection options provide a solution. These enable us to directly see and dynamically manipulate each of the elements known to the runtime environment (here: XOTcl's interpreter). A programmer can not only inspect all language elements, but also manipulate the language and customize it to the current requirements dynamically. In XOTcl, each introspection option is offered in the `info` method. `info` accepts a number of options. Another method (most often directly corresponding to the option's name) allows the programmer to change the option dynamically.

For instance, we can query a class `C1` for its superclasses:

```
C1 info superclass
```

We can also change this setting at runtime. For instance we can let `C1` have an additional superclass:

```
C1 superclass [concat [C1 info superclass] Plotter]
```

Now we have added a `Plotter` class to the existing superclass list of `C1`. With introspection options a tool like biok can connect user scripts to existing classes of the system.

Another common example of incremental programming is to adapt the algorithm of a given method. For instance, we can use the `info instbody` introspection option to retrieve the current body of a method and then dynamically enhance it with logging functionality, if only this one method should be logged:

```
C1 instproc calc {a b} [concat \
   {puts "C1->calc $a $b"; } \
   [C1 info instbody calc]]
```

Dynamic manipulation together with introspection is important for enabling rapid changeability in scripts without knowing internal details of the surrounding framework that is to be enhanced.

Another important feature for incremental programming is the invocation context. The invocation context is built from the interpreter's callstack and allows us to find out the calling context of the current object. Thus for dealing with rapid changes the current context can be queried and the script can deal, for instance, with different invocation sources in a context-sensitive way. In XOTcl the invocation context is given by the `self` command. In particular, `self` without arguments returns the current object ID. `self class` returns the class executing the current method, and `self proc` returns the current method name. `self callingobject` returns the calling object, `self callingclass` returns the calling class, and `self callingproc` returns the calling method. For message interceptors (see below) there are also options returning the name of the object, method, and class to be called by the Interceptor.

Customizations performed by a tool should be transparent for the user. This can be reached using message interceptors. A message interceptor is dynamically added to a computational entity (like an object, a class, a class hierarchy), and it intercepts all (specified) messages that are sent to this computational entity before, after, or instead-of the original message dispatch. User-defined functionality can be added to the existing system by using introspection options and invocation contexts together with message interceptors. In XOTcl there are two kinds of message interceptors:

- *Per-class/per-object mixins* are classes that are dynamically attached to or detached from a class or object. They intercept every message sent to a class or object and can handle the message before/after the original receiver.
- *Per-class/per-object filters* are special instance methods which are dynamically registered or de-registered for a class hierarchy or object. Every time an instance of this class hierarchy or object receives a message, the filter is invoked automatically and intercepts this message.

Both filters and mixin classes intercept messages sent to an object or a class hierarchy before they reach the original receiver. The interceptor can adapt the message to another receiver, handle it directly, or decorate it with arbitrary behavior before/after the original receiver gets it. Filters are used to implement entities and concerns cutting across an entity as a whole, whereas mixins only intercept certain message calls. Filter and mixin classes can be used to *transparently* observe the user's actions, and perhaps manipulate them. A simple filter example is a logging filter:

```
Class Logger
Logger instproc loggingFilter args {
  puts "called: [self]->[self calledproc]"
  next
}
```

This simple filter uses the invocation context to write the current object and called method to the standard output. `next` means that the next filter in the filter chain and finally the original receiver is invoked. A filter method can be registered for any class or specific object to log the class' or object's actions:

```
C1 instfilter Logger
```

Now all instances of `C1` are logged. Filters handle all methods of a specific class or object. Mixin classes handle only those methods that are specified on the mixin class. Consider we want to handle all user-defined methods (then we have to intercept all calls to the method `proc`):

```
Class ProcHandler
ProcHanlder instproc proc args {
  puts "called: [self]->[self proc]"
  # handle proc
  # ....
  next
}
```

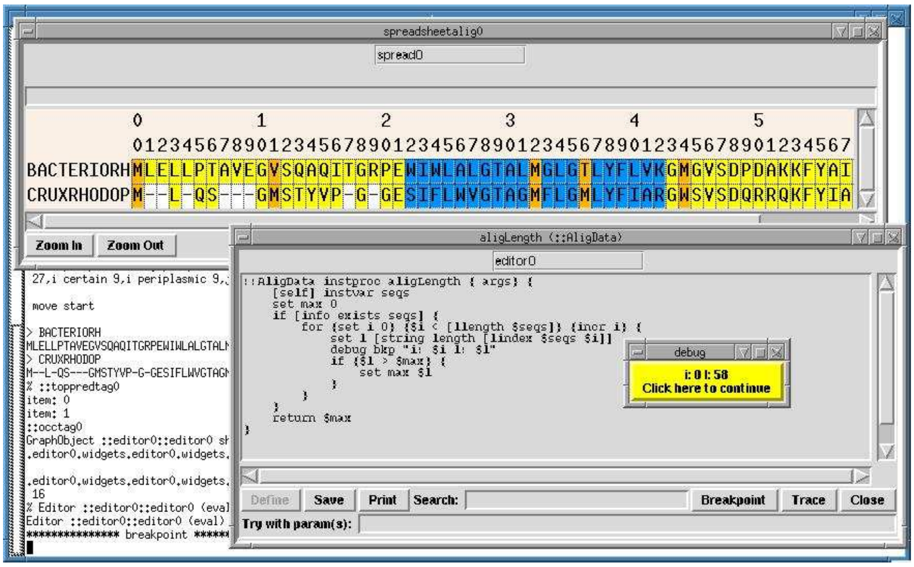

Figure 5: Programming in biok: the method displayed in the editor computes the length of the alignment. A breakpoint, set from within the source code, has occurred (small debug window).

This mixin only intercepts the `proc` method calls of the class or object it is registered for, e.g. only instances of `C1`:

```
C1 instmixin ProcHandler
```

XOTcl has been chosen to be used in biok for different reasons. First, the scripting capabilities of Tcl (such as text processing, integration tools, simple syntax) allow users to rapidly learn the language. Many tools, required for biok's internal implementation, are available and well-integrated with the language at the script level, including Tk (as the graphical toolkit), network, and system libraries. XOTcl combines these features with high-level object-oriented functionalities, such as introspection, language dynamics, and interceptors. In biok these functionalities are used for incremental program evolution at runtime.

Note that XOTcl is just a technical solution for the concepts presented in this paper. Many other languages (and language extensions) provide similar means. Often, however, some functionalities of XOTcl would have to be re-implemented in the other languages, before they could be used for a tool like biok. Aspect-oriented languages, for instance, provide aspects as a language construct that could be used instead of message interceptors. For the use in biok a dynamic aspect weaver would be required so that biok can combine (and remove) aspects for the existing systems dynamically. Languages, such as Smalltalk or Lisp variants, provide language dynamics but high-level interceptors would have to be implemented and added to these languages (which is even in these languages a non-trivial task as transparent interceptors require complex callstack manipulations). However, even though some effort might be required to provide similar functionalities in other languages, we do not see a principal problem in doing so.

7 Biok Programming Environment

Biok uses the XOTcl features discussed in the previous section to enable programmability for end-users. At any moment, the user can ask a graphical object for its source code by a menu. In Figure 5, a method editor is shown for a method of the spreadsheet. The code of the method is found by XOTcl introspection options, and can be redefined at any time. It can be saved at the user level, in a separate file, that the user can edit independently. This file is loaded at the start-up of the biok environment, after the system itself is loaded. According to the users of biok, this simple mechanism helps them to feel more secure: they can make errors and try new things. For instance, a user can change the code of a method to compute the alignment length differently (there is indeed a variation on this computation in current tools).

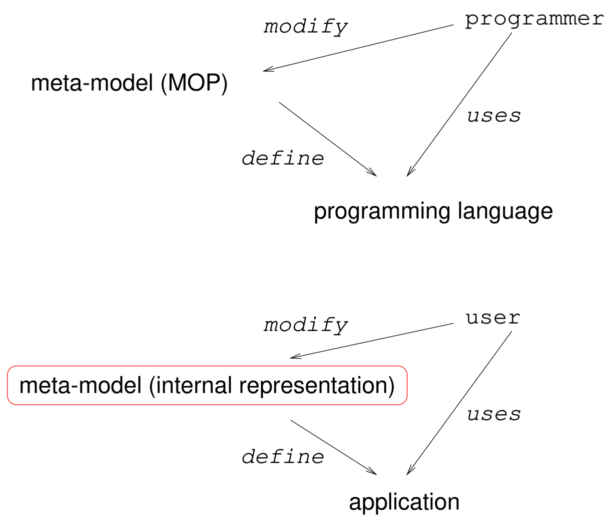

Figure 6: MAP: Meta-application Protocols

In Figure 5 sequences are what biologists call 'aligned', meaning that after comparison, corresponding letters are put together, and letters without any correspondence are aligned to 'gaps', represented here as hyphens. The well-known algorithm, based on dynamic programming, to compute this alignment has been implemented by a biology student having learned programming for only one year.

The biok editor provides tools for debugging, such as breakpoints or traces on method calls. It also allows you to run the edited method with parameters. You can also put breakpoints in the code. These display a small box with a custom message. There is also a tool to spy all the execution (implemented as an XOTcl filter): you can observe all the methods that are called during a certain time, for instance, to find out which methods are called when clicking on a specific button. This way, debugging (what is really important in incremental programming), is available directly in the programming environment (and accessible to end-users) as a first class feature.

8 Algorithmic Evolutions

We describe two variants of what we call "algorithmic flexibility." The concept draws from Wegner's concept [25] that computer algorithms should be more clever and more interactive to take the users' knowledge into account. First we describe a meta-protocol to let the biologist modify an algorithm behaviour by adapting an internal structure. Secondly we describe how to enable user interactions with the graphical interface on top of a meta-protocol.

In both variants, we can use introspection options available in XOTcl to query the current setting of this internal structure and use interception techniques, particularly *mixins*, for dealing non-intrusively with unexpected changes of the internal structures and algorithms. That is, the behaviour can be modified for some parts of the program and data, not all, and the modification can be dynamically removed by de-registering the mixin.

8.1 MAP: Meta-application Protocols

As illustrated in Figure 6, the idea of a MAP (Meta-application Protocol) is based on the MOP concept. However, we feel more comfortable with the meta-*application* term, as "MOP" comes from the specific field of object-oriented languages – hence "meta-*object*." In a meta-application protocol, the meta-model of the application (the internal representation) corresponds to the meta-object.

Similarly, the application exposes the language visible to the end-user (in biok based on graphic objects). Finally, the user corresponds to the programmer. As in MOPs, an explicit part of the internal structure that has a key role in the computation is documented as a *meta-level interface* to change the system behaviour.

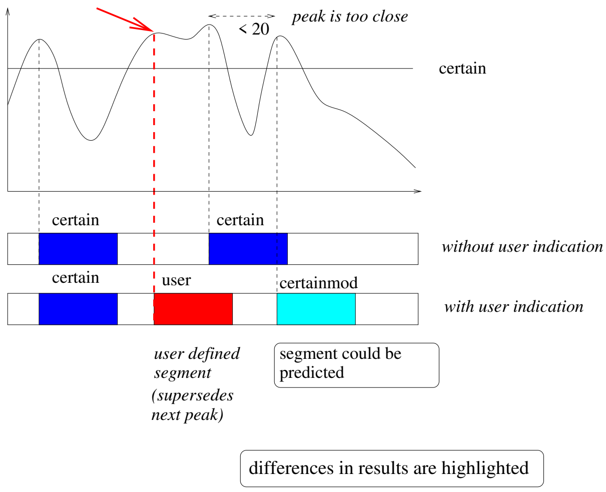

Figure 7: Interactive protocol to help detect transmembrane segments. Some segments are automatically predicted. Others (displayed in another color) are indicated by the biologist. Some segments cannot be predicted without user indications.

Recently a student applied this idea to a dynamic programming algorithm for *aligning molecular sequences* (Figure 5). Here the *protocol* of the MAP is: The heuristic of the algorithm can be expressed in a definite state automaton. It is both easy to extend (by adding states corresponding to external knowledge) and editable by a *knowledgeable* user. Compared to [1] that also expressed dynamic programming algorithms with a definite state automaton (DFA), the strategy chosen here just adds external knowledge, whereas [1] computes everything internally, which is more costly.

Although it could seem awkward to let a biologist edit the data structure of a DFA, we have conducted an interview with a scientist in Pasteur having programmed a graphical simulation environment for living cells. In this system users can enter differential equations in C++ in a specific part of the code [10]. Having such feature provided in a software written by a biologist shows that the effort for editing the data structure should not be too huge.

8.2 Interactive MAP protocols

A promising extension of the MAP concept is to find a way to integrate user's knowledge in computation *neither as a parameter nor as program* but just by interacting. In the example shown in Figure 7, we had noticed that the heuristic of the algorithm *detecting transmembrane segments* by their hydrophobicity [9] could fail in situations difficult to specify in the algorithm, but easy to show by the user. As a solution, we implemented a way for the user to indicate different starts for the segments. The implementation just adds a message interceptor (mixin) to two methods:

1. `segments`: selects the transmembrane segments;
2. `result_segments`: displays the predicted transmembrane segments (user defined segments are displayed in a different color).

Let us take the *sequence alignment* example again. We set an interactive mechanism where the user is able to indicate, by selecting areas in the spreadsheet, which segments should be aligned. This is then taken as input for the MAP described in 8.1. Constraints specified by the user, acting as external knowledge, just lead to new states in the DFA.

This way, by interfacing the MAP feature with a small set of interactive actions, the protocol is made available to biologists not having any knowledge about meta-protocols.

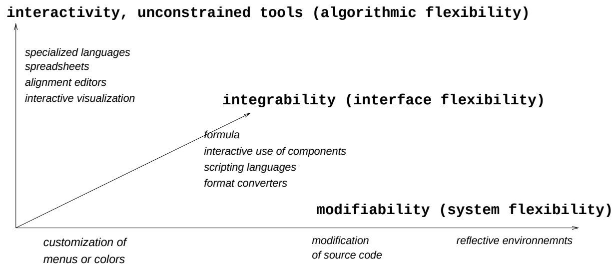

Figure 8: Dimensions of flexibility

9 End-User Programming and Participatory Design: a Complementary Approach

The considerations in this section rely heavily on [24] which describes how MOP and participatory design can be combined as a way to design software for flexibility.

Let us first see what participatory design contributes when applied without end-user programming. When adopting user-centered and participatory design, the chance to get an unadapted system really drops down. This was clearly the case for the spreadsheet tool and the programmable tag system in our prototype. We have conducted design workshops, and the subjects that are approached during these workshops are generally selected through either interviews and brainstorming sessions. Not surprisingly, among the 9 participatory workshops organized over the last 5 years, about 6 of them were focused on pattern searching and sequence features visualization, both handled by these tools.

The property that is gained is thus *design stability*, in other words, less need for re-design. However, as we have observed, evolution is an intrinsic need of biology research software, and participatory design does not necessarily lead to flexibility: it leads to the properties that are useful in a specific context, and there are contexts where flexibility is rather prohibited.

But there are benefits of using both programmability and participatory design together. As stated in Section 3.1, above, end-user programming generally requires a strong *anticipation*: the software must be instrumented to let the user program with the user interface. This results in taking heavy design decisions about what should be open to programming within the tool. On the other hand, we were able to state that *general programmability* through introspection, structured in a good programming environment, could help in enabling *any* part of the software to be changed. As a general property of the system, you do not have to specify programming features for various system parts. But there is still a risk that programming remains too difficult for the user, just because the environment does not provide the required help. We observed that by using the participatory approach:

1. *Dimensions of flexibility*, i.e. potential unexpected changes, are generally better anticipated. A user-centred approach let simultaneously show up spots of stability and spots of variability (see [24]), which was confirmed by our experiences. In scientific software, for instance, during the interviews and workshops, three dimensions of flexibility appeared as important (see Figure 8):
   - *system flexibility*: capacity of the system to support change,
   - *algorithmic flexibility* (or interactivity): capacity of the system to be controlled by the user, as illustrated by the interactive MAP described in Section 8.2, that emerged almost as such during a participatory workshop, and
   - *interface flexibility* (or integrability): capacity of the system to be combined with others.
2. *Programming features and tools are better designed and adapted to the user needs*: just having any programming environment does not help as such. This environment's features must be correctly anticipated. For instance, we observed that "creating new classes" rarely occurs in the tasks of the biologist, whereas method creation or redefinition is very often used. Actually, users, able to design a class hierarchy, can also do it outside of biok, as happened with one of our student, who designed the MAP and the DFA (with a little help from us).

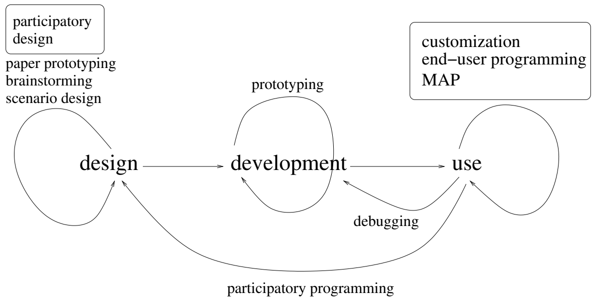

Figure 9: Development cycles and anticipated co-evolution

Similarly, we had not identified the tag creation as being that important for the tool, and, during a prototyping workshop, it appeared how critical it is to have a well-adapted tool for this. So, we organized an additional workshop with a pre-built mockup (both a storyboard and a mockup in fact) which is very close to the final tool, and we played with the data in it.

Thus, it is the combination of general programming capabilities and the user-centered approach which enables to design for unanticipated evolution by supporting general lines of changes.

10 Summary of the General Approach

In summary, it is important to enable the user to adapt the system or to evolve it over time, and to have a system properly designed and already adapted to the user's tasks by using user-centred methods and a participatory design approach. Both aspects actually belong to the same methodological approach relying on the idea of co-evolution [2] of design, development, and use of a software system. This leads to software development cycles with evolution anticipated at every step (see Figure 9):

- involving users at the design step (participatory design),
- including development at the use step (end-user programming), and
- developing a *prototyping* approach: as augmented in [3], prototyping, i.e. using rapid prototyping tools to program some of the critical parts of a system can economize a lot of time; in this aspect, the flexibility of the programming language and tools is important.

We can also situate this framework by comparing it to the USE framework as described in [12]:

- *Time of change:* design, prototyping, development and software use are supported.
- *Type of evolution (sequential or parallel):* both sequential evolution, where users developments are re-integrated in the system (which happened several times), and parallel evolution, where users are allowed to explore independently, are supported.
- *Incrementality:* the main granularity of change are object-oriented methods. XOTcl provides ready-to-use features for incremental method re-definition; furthermore, most of the changes we were able to observe were either done or to be done at the method level (adapting a functionality). Mixins can be used to bundle and reuse method re-definitions.
- *Automation:* none; the integration of the user's code into the system should be a subject of discussion between the software engineer and the biologist. Sometimes, the user's code stands as a kind of "active" specification.

- *Change effort:* the conceptual framework of cognitive dimensions in [6] provides some tools for checking this aspect. *Viscosity* is the term to describe the effort necessary to perform a change, and it is related to other variables described in this framework. For instance, *closeness of mapping* describes how the program or the environment correspond to the problem. *Hidden dependencies* hinder incremental changes. *Premature commitment* may require for the user to anticipate too many things in order to begin a change. In biok, we decided to map general classes of biological objects (sequences, alignments, 3D molecules) to graphical objects. Then the user can localise the code more easily. Efforts to *undo a change* must also be evaluated: in biok, the user just has to remove the files containing his or her code in order to go back to the system definitions.

- *Openness:* basically, the biok system is open for change, and all the source code is available to the user.

The benefits of our approach have been discussed widely in this paper. In summary, the main benefits are that we provide an approach for co-evolution of the aspects design, development, and use of (scientific) software. We provided a conceptual integration of technical flexibility functionalities and design methods to enable the user to easily evolve the software. The user does not have to understand advanced programming concepts or large parts of the biok framework, but only the graphical objects of the familiar work task. Internally, however, we can use the MOP, language dynamics, and reflection capabilities to support the desired software evolution non-intrusively.

Of course, in some situations our approach may also incur some liabilities. The approach cannot be used in any software, but requires a preparation of the scientific software tool to be open for end-user evolution. This requires some efforts to integrate the MOP, dynamic language, and reflection with the GUI that is used for the scientific tasks. User-centered and participatory design require some organizational effort, and, as in any quality process, an approach based on co-evolution has an overhead compared to ad hoc approaches.

11 Conclusion

We have described an approach and a prototype designed for incremental evolution, from the design step to the use step. The objective of this research is to explore the dimensions of software flexibility that has been observed as being critical in the field of biology research and is likely to be critical in other scientific research areas as well.

Thus we believe that the approach could be generalized to other software development areas. We have already applied similar ideas for domain-specific customizations by content editors in the field of interactive television (see [5]). In particular, user-centered design proved to be a very rich tool to better capture and anticipate software evolution.

The prototype we have described has been used by several students during a few months. The use by students both plays a proof-of-concept and a methodological role in this research. We also aimed at exploring the problem space, more than tried to evaluate a unique technical solution to a well-defined problem.

The flexibility on the technological side (provided by the language XOTcl) played a key role in our approach, and we hope that our approach illustrates how important flexibility is on the technological side. Technical flexibility and advanced programming functionalities, however, can also be counter-productive in the end-user realm (if they only add complexity). Thus we have provided a technical solution for letting users only deal with the graphical objects associated with the particular work task. These technical solutions are tightly integrated with user-centred design and participatory design methods to help the end-user deal with the programming task conceptually.

Acknowledgements

We would like to thank Albane Leroch for having implemented the algorithm in [9] and having, during her project, made several suggestions regarding the programming environment in biok. We would like to thank Alexis Gambis and Alexandre Dehne-Garcia for their conceptual work on the design of meta-applications protocols. We would also like to thank Pierre Tuffery, Fabienne Dulin, and Julie Mauro for their hard work in the design and implementation of the molecule viewer.

References

[1] E. Birney and R. Durbin. Dynamite: a flexible code generating language for dynamic programming methods used in sequence comparison. *Proc Int Conf Intell Syst Mol Biol*, 5:56–64, 1997.

[2] G. Bourguin, A. Derycke, and J. Tarby. Beyond the interface: Co-evolution inside interactive sytems a proposal founded on activity theory. In *Proceedings of IHM-HCI 2001*, Sept. 2001.

[3] F. P. Brooks. No silver bullet,essence and accidents of software engineering. *Computer Magazine*, Apr. 1987.

[4] A. DiSessa. *Changing Minds: Computers, Learning, and Literacy*. MIT Press, 1999.

[5] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architectures. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS 2002)*, Montpellier, France, September 2002.

[6] T. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *Conference on People and Computers V*. Cambridge University Press, 1989.

[7] J. Greenbaum and M. Kyng. *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, New Jersey Lawrence Erlbaum Associates, 1991.

[8] D. C. Halbert. Smallstar: Programming by demonstration in the desktop metaphor. In *Watch What I Do. Programming by Demonstration, Part I Systems.*, pages 238–269. MIT Press, 1993.

[9] G. V. Heijne. Membrane protein structure prediction. hydrophobicity analysis and the positive-inside rule. *J. Mol. Biol*, 225(2):487–494, 1992.

[10] M. Kerszberg and J.-P. Changeux. A simple molecular model of neurulation. *BioEssays*, 20:758–770, 1998.

[11] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[12] G. Kniesel, J. Noppen, T. Mens, and J. Buckley. The first workshop on unanticipated software evolution (USE 2002). In *ECOOP 2002 Workshop Reader, Springer Verlag, LNCS 2548*, 2002.

[13] C. Letondal. *Programmation et interaction*. PhD thesis, Université de Paris XI, Orsay, 2001.

[14] C. Letondal. A web interface generator for molecular biology programs in Unix. *Bioinformatics*, 17(1):73–82, 2001.

[15] H. Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2000.

[16] H. Lieberman, B. A. Nardi, and D. Wright. Grammex: Defining grammars by example. In *ACM conference on Human Factors in Computing Systems (Summary, Demonstrations) (CHI '98), Los angeles, California, USA*, pages 11–12. ACM Press, Apr. 1998.

[17] W. E. Mackay. Triggers and barriers to customizing software. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 153–160. ACM Press, 1991.

[18] P. Maes. Computational reflection. In *Technical report 87-2, Free University of Brussels, AI Lab*, 1987.

[19] A. Morch. Evolving a generic application into a domain-oriented design environment. *Scandinavian Journal of Information Systems*, 8(2):63–89, 1997.

[20] A. Morch. Three levels of end-user tailoring: Customization, integration, and extension. In M. Kyng and L. Mathiassen, editors, *Computers and Design in Context.*, pages 51–76. The MIT Press, Cambridge, MA, 1997.

[21] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT Press, 1995. 162 pages.

[22] G. Neumann and U. Zdun. Xotcl, an object-oriented scripting language. In *Proceedings of 7th Usenix Tcl/Tk Conference (Tcl2k), Austin, Texas, Feb 14-18*, 2000.

[23] J. E. Stajich, D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigian, G. Fuellen, J. G. Gilbert, I. Korf, H. Lapp, H. Lehvslaiho, C. Matsalla, C. J. Mungall, B. I. Osborne, M. R. Pocock, P. Schattner, M. Senger, L. D. Stein, E. Stupka, M. D. Wilkinson, and E. Birney. The bioperl toolkit: Perl modules for the life sciences. *Genome Research*, 12(10):1611–1618, 2002.

[24] R. H. Trigg. Participatory design meets the mop: Informing the design of tailorable computer systems. In *Proceedings of the 15th IRIS (Information systems Research seminar In Scandinavia) Gro Bjerknes, Tone Bratteteig, Karlheinz Kautz (eds.), August 1992, Larkollen, Norway.*, 1992.

[25] P. Wegner. Why interaction is more powerful than algorithms. *CACM*, 40(5):80–91, May 1997.