# Making GUIs less messy: a preliminary report on the Take Back Control library

Theodore S. Norvell
*Dept. Electrical and Computer Engineering*
*Memorial University*
*theo@mun.ca*

***Abstract*— Implementing user interfaces can be messy. The implementation is typically based on a state machine and looks nothing like the nice structured use cases that you probably wrote to specify the UI and reviewed with the customer. The implementation is divorced from the specification by at least two levels of abstraction: translation to a state machine and implementation of that state machine This makes user interfaces hard to change, hard to validate, hard to understand.**

**This paper presents a library of combinators that allows us to write user interfaces in a way that is easy to read and that matches use case based specifications. This makes implementation easier, validation easier, and the code easier to understand. It replaces a messy divorce with a harmonious union between the specification and implementation.**

***Index Terms*— User interfaces, inversion of control, monadic programming**

## I. Introduction

My first job after graduating from university was to create a prototype programming environment and an interpreter for the graphical programming language ProGraph. [3] The language, with a different implementation, later became a successful commercial product. One thing that was interesting about my implementation is that it was not event driven. When my program needed input, it would wait (by polling) for the user to do something, which in this case was either to click on a graphics tablet with a special stylus or to press a key on the keyboard. This was a program written without inversion of control. This was a pleasant way to write a user interface.

The second thing that is relevant about the user interface (UI) of the ProGraph project is the way that I documented it. For this I used a context free grammar (CFG) [1], [2] in which the symbols were either actions of the user or actions of the system. I probably did this because, as a new graduate in 1985, I didn't know much about UI specification, but I sure knew a lot about CFGs. Also the use of a CFG was a good fit for the project because the IDE had two modes: interpretation and editing. While editing you could start the interpreter, but if it stopped —perhaps because of a bug or an incomplete subroutine— you could then enter the editor, change the code and resume interpreting. Thus editing and interpreting sessions could be nested to an arbitrary depth. The implementation of this was easy because the UI didn't use inversion of control, so starting a new editing or interpretation session was just a matter of calling a (recursive) subroutine. I don't think this was actually good UI design, but the point is that it required more than a finite state machine to describe. Although the use of CFGs to document or specify UIs never caught on (well not yet, anyway), the use of Use Case models is very popular and Use Cases resemble extended context free grammars if you don't look too carefully at the details.

For some time it has seemed to me that we should not only be able to document UIs using context-free grammars, but to write them in a way that resembles CFGs. An analogy —if you are familiar with parsing— may be found in the parsing of complex textual input specified by a CFG. We have a choice: we can derive a complex push-down automaton [1] and implement it as such or we can write our parser using recursive descent [13].[1] The recursive descent parser will correspond clearly to the grammar and hence be far more maintainable as the grammar changes. By using parsing combinators [9], we can make the correspondence between the grammar and the parser quite transparent.

What do we mean by "inversion of control". Really it means that instead of high-level code calling low-level code, we do it the other way around. [5] Low-level code calls the high-level code. In the case of UIs the high-level code is the application-specific code, while the low-level is the UI framework, which is a reusable library. To use the UI framework we (the application programmers) register various event handling routines with an instance of the framework. Then the high-level code hands control off to a routine called the "event loop"; that's typically the last thing the main program does. The event loop waits for an event to happen then it calls any registered handlers —this is where the low-level code calls the high-level code— then it waits again, and so on. Inversion of control is sometimes called the Hollywood Principle as in "don't call us, we'll call you". Another name for it is "event driven" UIs.

In the rest of this paper I will advocate for taking back control. I'll call this style "upright control." We might even call this the Matthew 7:7 principle: "Ask, and it shall be given you."

Furthermore, I will present a simple library called Take Back Control (TBC) that provides a way to write UIs in

[1]There are other choices. One is to use a parser generator to convert the CFG to a push down automaton automatically. I'll ignore this choice for now, although when I started this project, I was thinking along the lines of creating a parser generator suitable for UIs. As it developed, a simpler solution emerged.

an upright style on top of some underlying framework using inversion of control.

## II. Specification of UIs

It has become common to specify UIs using use cases. [10] Use Cases provide a structured approach to designing and documenting system behaviour. They can be read or written by non-software-professionals.

Here is an example of a set of use cases for a simple web-based integrated development environment. The UI consists of two radio buttons used to select a language to edit, a button to run the current program, a button to resume editing, two editor controls (one for Java and one for C++) where the user can edit the text of a C++ or Java program, and an execution control (called the Teaching Machine), where the user can step through the program they've written. The editor controls and the execution control are library components and we will treat them as black-boxes; we will only concern ourselves with making them appear and disappear at the right times. So we will only worry about the 4 buttons. Initially only the two radio buttons are shown.

**Select initial language.**

- Precondition: Initial state
- Scenario
  1) The user selects either the C++ or Java radio button.
  2) The system shows the editor for C++ or the editor for Java, as appropriate and also the Run button.
- Postcondition: Editing

**Make edits**

- Precondition: Editing
- Scenario
  1) The user interacts with the editor for the current language
- Postcondition: Editing

**Change Language**

- Precondition: Editing
- Scenario
  1) The user select the radio button of the other language
  2) The system hides the current editor and shows the other
- Postcondition: Editing

**Run**

- Precondition: Editing
- Scenario
  1) The user selects the Run button
  2) The system disables both radio buttons and hides the editor for the current language
  3) The system shows the Teaching Machine and the Edit button.
  4) The user interacts with the Teaching Machine.
  5) The user selects the Edit button
  6) The system enables both radio buttons, hides the Teaching Machine and the Edit button, and shows the editor for the current language
- Postcondition: Editing

We might supplement the use-case model with a table showing the state (enabled, disabled, hidden) of each control for each of the five conditions. This would mean there is no need to specify hiding, enabling, and showing in the use case model

The ordinary methodology to go from a set of use cases to code would go through some sort of state machine design. For the simple UI specified above, the state machine apple would not fall far from the use case model tree, but as the use case model gets more complicated, especially as scenarios become longer and more complex, the number of states quickly goes up. Next we turn the state machine into code. The connection between the code and the original set of use-cases is now very tenuous. The logic of the UI is now distributed across an amorphous set of event handling routines that communicate using shared data. The original hierarchical structure of the use cases is lost. There is no subroutining of state machines, so we can not bring abstraction to the rescue.

Our intention is to be able to write code that closely matches our use-case model. In particular we would like to be able to write something like this.

```
function initialState() : Process⟨Triv⟩ { return
  await( click(langButton[CPP]) && unit(CPP)
       || click(langButton[JAVA]) && unit(JAVA) ) >=
  function( lang : Int ) { return
    showEditor(lang) >
    editingState(lang) ; } ; }

function editingState( lang : Int) : Process⟨Triv⟩ { return
  await( change(lang) || run(lang) ) ; }

function change( lang : Int) : GuardedProcess⟨Triv⟩ { return
  click(langButton[1-lang]) &&
  hideEditor(lang) >
  showEditor(1-lang) >
  editingState(1-lang) ; }

function run( lang : Int) : GuardedProcess⟨Triv⟩ { return
  click(runButton) &&
  hideEditorPane() >
  showTM() >
  await( click(editButton) && skip() ) >
  showEditorPane() > editingState(lang) ; }
```

## III. Take back control

My approach to taking back control is to use a domain-specific language (DSL) for UIs. In this case the (DSL) is embedded in another language. The host language is Haxe [4], which was chosen for several reasons: In contrast to JavaScript, it is strongly typed; this aids software design by automatically spotting superficial errors, thus allowing the

engineer to concentrate on avoiding deeper errors. It supports generic classes [11]. It compiles to JavaScript, which allows us to use it for developing the client side of web-applications. It can be compiled to other languages too including Java and C++, which allows the library to be used for applications where JavaScript is not appropriate. It supports operator overloading. It supports lambda expressions with lexical scoping.[2]

The library or DSL —take your pick— is called Take Back Control, TBC for short.

### A. Processes

A fundamental type in TBC is that of a $\mathsf{Process}\langle A\rangle$. This type represents computations that compute values of type $A$. For example if $p$ is a $\mathsf{Process}\langle\mathsf{Int}\rangle$ it is an object that can compute a value of type Int. Now to initiate execution of our process, we can use its go method, which takes a function as an argument. For example we could write

$$p.\mathsf{go}(\lambda(x:\mathsf{Int})\{\mathsf{trace}(x);\}) \tag{1}$$

Once the process has finished executing, the function is called with its result, so in this case the trace function[3] will be called with the result of executing process $p$.

A simple example of a $\mathsf{Process}\langle A\rangle$ object is $\mathsf{unit}(x)$, where $x$ is of type $A$. This process immediately computes $x$. Thus

$$\mathsf{unit}(42).\mathsf{go}(\lambda(x:\mathsf{Int})\{\mathsf{trace}(x);\})$$

is the same as $\mathsf{trace}(42)$.

At this point you may be wondering how a process differs substantively from a function. I.e., how does (1) differ from $\mathsf{trace}(f())$ where $f$ is a function that computes the same value as $p$ produces? The difference is that a process may require time and interaction to produce its result, but the go method returns immediately. For example, as we will see soon,

$$\mathbf{var}\ p = \mathsf{await}(\mathsf{click}(\mathsf{b})\ \&\&\ \mathsf{skip}()) > \mathsf{pause}(1000) > \mathsf{output};$$
$$p.\mathsf{go}(\lambda(x:\mathsf{Triv});\{\})$$

completes immediately, but it has the side effect of enabling a button b such that, when it is clicked, the process output is executed 1s later. In the next sections we will look at the meaning of constructs such as $>$, $\&\&$, await, click, and pause.

### B. Composing processes

We can compose processes by piping the result of one into another. For example, suppose $p$ is of type $\mathsf{Process}\langle A\rangle$ and $f$ is a function of type $A \to \mathsf{Process}\langle B\rangle$, then $p >= f$ is a $\mathsf{Process}\langle B\rangle$. Of course it executes by first executing $p$ and then executing $f(x)$ where $x$ is the value computed by $p$. The implementation is simple: $p >= f$ is an object $q$ whose go method is implemented by

$$q.\mathsf{go}(k) = p.\mathsf{go}(\lambda(x:A)\{f(x).\mathsf{go}(k);\})$$

[2]An example of a Haxe lambda expression can be found on the last three lines of function initalState at the end of the previous section. At times, to save space, I will write $\lambda$ in place of the Haxe keyword function.

[3]trace is a Haxe library function that outputs its argument.

If you know about monads [12], you might recognize that $>=$ this is a "bind" operation, also known as "flatMap". Together $>=$ and unit form a monad meaning they obey all the laws one would expect a monad to obey.[4] I won't discuss monads more in this paper.

Often the result of $p$ is not needed. When it is not, we can abbreviate $p >= \lambda(x:A)\{\mathbf{return}\ q;\}$ by $p > q$. This is essentially the sequential composition operator: do $p$ and then do $q$.

To achieve side-effects the process $\mathsf{exec}(f)$ calls function $f$ with no arguments. Thus $\mathsf{exec}(f).\mathsf{go}(k) = k(f())$.

We can also combine processes in parallel;

$$\mathsf{par}(p,q)$$

is a process that computes a pair of results. The actions of $p$'s and $q$'s executions are interleaved in an arbitrary fashion. It should be noted that this form of parallelism involves no true concurrency; all actions are executed by the same thread. Actions are bounded by points where the processes wait, as discussed in the next section.

Another way to achieve parallelism is to use the fork method, which starts a new process execution, but does not wait for it to finish. We have

$$\mathsf{fork}(p) = \mathsf{exec}(\lambda()\{p.\mathsf{go}(\lambda(a:A)\{\});\mathbf{return}\ \mathsf{null};\})$$

Again there is no true concurrency. The forked processes actions are interleaved with the actions of other active processes. There is no join mechanism, but it can be simulated using a channel.

### C. Guards, guarded processes, and waiting

A guard is an object that represents a category of events. For example, if $b$ represents a button on a web page, $\mathsf{click}(b)$ is an object of type $\mathsf{Guard}\langle\mathsf{Event}\rangle$ where Event is the type of HTML event objects. The $\mathsf{click}(b)$ object represents clicks on button $b$.

We can combine a guard $g : \mathsf{Guard}\langle E\rangle$ with a function $f : E \to \mathsf{Process}\langle A\rangle$ to make guarded process $g >> f$ of type $\mathsf{GuardedProcess}\langle A\rangle$. This guarded process can only execute when an event occurs.

If $m$ is a guarded processes of type $\mathsf{GuardedProcess}\langle A\rangle$, then $\mathsf{await}(m)$ is a process of type $\mathsf{Process}\langle A\rangle$. This process executes by waiting until an appropriate event occurs and then executing the guarded process. For example, $\mathsf{await}(\mathsf{click}(b) >> f)$ is a process that waits until button $b$ is clicked and then executes process $f(e)$, where $e$ is the event object for the click.

Guarded processes can be combined so that $m \parallel n$ represents a choice between the two guarded processes. For example, if $b0$ and $b1$ are two different buttons, we can

[4]Haskell programmers might wonder why I used $>=$ for bind rather than $>>=$ as in Haskell. The reason is that Haxe only allows existing operators to be overloaded.

TABLE I
SUMMARY OF OPERATIONS

| Meaning | Type | Syntax | Alternative Syntax |
|---|---|---|---|
| Launch process $p$. | Void | $p$.go($h$) | |
| Do $p$ and then do $q$. | Process$\langle B\rangle$ | $p > q$ | $p$.sc($q$) |
| Do $p$, then do $f(x)$, where $x$ is the result of $p$. | Process$\langle B\rangle$ | $p >= f$ | $p$.bind($f$) |
| Do $p$ and $q$ in parallel. | Process$\langle Pair\langle A, B\rangle\rangle$ | par($p, q$) | |
| Do $p$ in parallel. | Process$\langle Triv\rangle$ | fork($p$) | |
| Make a guarded process that does $p$ when fired. | GuardedProcess$\langle A\rangle$ | $g$ && $p$ | $g$.andThen($p$) |
| Make a guarded process that does $m(e)$ when fired, where $e$ represents information about the event. | GuardedProcess$\langle A\rangle$ | $g >> m$ | $g$.guarding($m$) |
| Make a choice between guarded processes. | GuardedProcess$\langle A\rangle$ | $gp_0 \parallel gp_1$ | choose($gp_0, gp_1$) |
| Wait for an event, then execute a guarded process. | Process$\langle A\rangle$ | await($gp$) | |
| Output on channel $c$. | Guard$\langle A\rangle$ | $c$.out($x$) | |
| Input from channel $c$. | Guard$\langle A\rangle$ | $c$.in() | |

Variables above have the following types: $p$ : Process$\langle A\rangle, h : A \rightarrow$ Void, $q$ : Process$\langle B\rangle, f : A \rightarrow$ Process$\langle B\rangle, g$ : Guard$\langle E\rangle, m : E \rightarrow$ Process$\langle A\rangle, gp$ : GuardedProcess$\langle A\rangle, c$ : Channel$\langle A\rangle, x : A$

represent a choice between them by.

$$\begin{aligned}&\text{await(}\ \text{click}(b0) >> f0\\ &\quad \parallel \text{click}(b1) >> f1\\ &\quad \parallel \text{timeout}(500) >> f2\ \text{)}\end{aligned}$$

The final guard represents 500ms passing and always produces null. So, if no button is clicked within 500ms, $f2$(null) will be executed.

Often the information produced by an event is not needed. We can write $g$ && $q$ to mean $g >> \lambda(e : E)\{\mathbf{return}\ q;\}$. The $pause(n)$ process used above is simply a shorthand for

$$\text{await}(\text{timeout}(n)\ \&\&\ \text{skip}())$$

where skip() is a shorthand for unit(null).

At this point the code at the end of Section II should make sense. One thing you might wonder about is what happened to the "Make edits" use case and the line in the "Run" use case that says "the user interacts with the Teaching Machine". Since these components are reused rather than being written using TBC, they are written in an event driven manner and the user can use them whenever they are visible. This is an example of TBC playing well with legacy components.

### D. Communication

Processes can participate in communication events patterned after CSP [8]. Given a channel $c$ : Channel$\langle A\rangle$ and a value $a$ : $A$, $c$.in() is a Guard$\langle A\rangle$ that receives a communication, and $c$.out($a$) is a Guard$\langle A\rangle$ that outputs value $a$.

Communication is synchronous in that both input and output guards need to be enabled at the same time in order for either one to happen. There is no buffering. Communication involves two executing processes, that is it is point to point and not broadcast.

More complex communication mechanisms can be build on top of this simple one.

## IV. Conclusions, related and future work

You can see from the example at the end of Section II that the combinators presented above allow the control portion of UIs to be written in an "upright" rather than "inverted" style. The consequence of this is that we can use familiar structured control constructs to create a structured program that tells a story, just a use cases tell a story. Perhaps most importantly we can name parts of that story, allowing us to use procedural abstraction, i.e. subroutines, to structure the program.

Table I summarizes the operations. As the table shows, where operator overloading has been used, an alternative syntax is available; this allows the library to be used even when the application layer is written in a language other than Haxe, but which Haxe can be translated into, for example JavaScript or Java.

The ideas in this paper come from many sources: Monadic parser combinators [9], process algebras such as CSP [8], Scala's and Aka's actors [7], continuation passing style [6], and so on.

This is only a preliminary report. There is much work to be done yet. Channels need to be implemented. There needs to be a way to filter events based on predicates. Perhaps there should be a way to disable guarded commands. Communication with servers (e.g. XmlHttpRequests) and other asynchronous IO needs to be implemented.

## References

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice Hall, 1972.

[2] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[3] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pages 150–156, 1989.

[4] Benjamin Dasnois. *haXe 2*. Packt, 2011.

[5] Martin Fowler. Inversion of control. `http://martinfowler.com/bliki/InversionOfControl.html` accessed October 2015, June 2005.

[6] Jr. Guy Lewis Steele and Gerald Jay Sussman. Lambda the ultimate imperative. A. I. Memo 353, M.I.T., 1976.

[7] Philipp Haller and Martin Oderski. Event-based programming without inversion of control. In *Proceedings of the 7th Joint Conference on Modular Programming Languages*, number 4228 in LNCS, pages 4–22. Springer-Verlag, 2006.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1994.

[9] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.

[10] Ivar Jacobsen, Grady Booch, and James Rumbaugh. *The Unified Software Development Process: The Complete Guide to*. Addison-Wesley Professional, 1999.

[11] Martin Oderski and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159. ACM, ACM, 1997.

[12] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM, ACM, 1992.

[13] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.