Reactive Programming using the Algebra of Communicating Processes

André van Delft
Independent Researcher
andre dot vandelft at gmail dot com

Anatoliy Kmetyuk
Student, Odessa National Academy of Law
anatoliykmetyuk at gmail dot com

ABSTRACT

R&D on reactive programming is growing and this has delivered quite many language constructs, libraries and tools. Scala programmers can use threads, timers, actors, futures, promises, observables, the async construct, and others. Still it seems to us that the state of the art is not mature: reactive programming is relatively hard, and confidence in correct operation depends mainly on extensive testing. Better support for reasoning about correctness would be useful to address timing dependent issues.

The Algebra of Communicating Processes (ACP) has a potential to improve this: it lets one concisely specify event handling and concurrency, and it facilitates reasoning about parallel systems. There is an ACP-based extension to Scala named SubScript. We investigate how it helps describing the internal behavior of Akka actors, and how it combines with futures.

There is a considerable performance penalty for actors that are mainly sending messages to one another. Therefore SubScript is not well applicable to actor systems for which efficient message handling is critical. However, for such systems SubScript may still be useful as a tool for prototyping, performance prediction and test specifications.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Dataflow languages

General Terms

Languages, Theory

Keywords

Algebra of Communicating Processes, ACP, data flow, concurrency, non-determinism, reactive programming, actors, futures, Akka

1. INTRODUCTION

As argued in the Reactive Manifesto[1], the need is growing for applications that are event-driven, scalable, resilient and responsive. Reactive programming gains attention, particularly in the Scala world. Developers can use the Akka toolkit[2], which offers abstractions such as actors [**?**], futures [**?**] and state transition machines (STMs). Akka has become a solid programming platform. The message throughput of Akka actors suits many applications: in the order of magnitude of one million per second, on a modern CPU.

Yet we think that the way internal behavior of Akka actors is expressed, should be improved. Actors need to process incoming messages; send messages on to other actors and be ready for their responses; have timeouts for such expectations; and all this concurrently. It is possible to express such tasks in both object-oriented and functional languages, and certainly in Scala. Yet we think that the current Akka programs are a bit obscure:

- Actors may have a *receive* method for processing incoming messages; state is then usually maintained using instance variables. The receive method is essentially a list of incoming message kinds and the associated handlers. The typical order for such messages has no clear relation with the program text.
- Alternatively actors may contain an STM: a `become` method allows to change to a parameterized state. This overcomes part of the unclarity of the *receive* method, but the code may seem a bit spaghetti-like. `become` is a kind of jump, a higher level reminiscent of the goto instruction in the doghouse.
- Futures and async constructs are available for concurrency and time management inside a single actor. Futures are high level abstractions for convenient expressing data dependencies while allowing for concurrency; however in our opinion these constructs obfuscate the control flow somewhat.

For us this was illustrated during the Coursera course "Principles on Reactive Programming" that we both took in November and December 2013[3]. Going from the natural language specification to a working program seemed very hard. We did not feel well about our solutions to the main actor programming assignment: these passed both our tests and the tests created by the course leaders, but the program text was not clear enough to reason easily about correctness, e.g. with respect to time dependencies and race conditions.

One of us was helped by designing the actor behavior for this assignment in the formalism of the Algebra of Communicating Processes (ACP). This is a concurrency theory that allows for concise specifications of event-driven and concurrent processes. It also helps formal reasoning about process behavior. ACP is good at describing processes that communicate synchronously, and less at describing asynchronously communicating processes, such as actors.

[1]http://www.reactivemanifesto.org/

[2]http://akka.io/

[3]https://www.coursera.org/course/reactive

Thus it seems well suited for describing internal actor behavior, and, for the time being at least, not for complete actor systems.
It is also possible to program applications using ACP. We are developing an ACP based extension to Scala by the name of SubScript. This paper is a follow up to a paper presented at the Scala Workshop 2013[?] about dataflow programming support in SubScript, with applications to actor systems. We present in this paper the ideas that have evolved since then, and a more complete example actor application. Futures and ACP processes have some common properties. Apparently they may be mixed in SubScript programs, by virtue of the implicit features in Scala.
Since the previous paper there have been some changes in the syntax of SubScript: all fat arrows related to dataflow, such as => and ==> are now curly arrows such as > and   >. Script closure notation now uses rectangular brackets ([. . . .]) instead of triangular brackets (< . . . . >).
SubScript comes with a compiler, which is a derivative of the regular Scala compiler. It transforms ACP-like specifications into calls to an API of a SubScript Virtual Machine. At the time of writing the compiler & virtual machine understand code that closely matches the earlier draft specifications for the Coursera assignment. Under the hood the same Akka functions are called as in a plain Scala version. This is desirable since the Akka API is mature and solid, and the SubScript layer on top it is rather small.
However the process management by the SubScript virtual machine requires processing power. On a modern CPU the throughput of SubScript actions is typically between 10,000 and 100,000 per second. This means that message processing in a SubScript-actor program is 10 to 100 times slower than in a plain Scala version. This gap is smaller for applications where the internal processing of a single message requires more time than the message handling by the Akka library.

The rest of this paper is structured as follows: Four sections introduce ACP, SubScript, the "call graph" semantics, and the basic implementation. These sections contain text fragments literally copied or adapted from the predecessor paper. The latter has more details; there are some minor syntactical differences.
Then sections with example applications highlight SubScript's symbiosis with actors, futures and observables. Thereafter the implementation is discussed. Next we report the measured performance of SubScript actors. We then refer to some related work, and make concluding remarks.
The work is in progress. Our actor examples compile and run well, but the compiler does not process all new syntax yet, at the time of writing. The interoperation of processes and futures is not yet working.

2. ACP

The Algebra of Communicating Processes [?] is an algebraic approach to reasoning about concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras or process calculi[4]. More so than the other seminal process calculi (CCS [?] and CSP [?]), the development of ACP focused on the algebra of processes, and sought to create an abstract, generalized axiomatic system for processes.
ACP uses instantaneous, atomic actions (a,b,c,...) as its main primitives. Two special primitives are the deadlock process δ and the empty process ϵ. Expressions of primitives and operators represent processes. The main operators can be roughly categorized as providing a basic process algebra, concurrency, and communication:

[4]This description of ACP has largely been taken from Wikipedia

- *Choice and sequencing* - the most fundamental of algebraic operators are the alternative operator $(+)$, which provides a choice between actions, and the sequencing operator $(\cdot)$, which specifies an ordering on actions. So, for example, the process $(a + b) \cdot c$ first chooses to perform either a or b, and then performs action c. How the choice between a and b is made does not matter and is left unspecified. Note that alternative composition is commutative but sequential composition is not (because time flows forward).
- *Concurrency* - to allow the description of concurrency, ACP provides the merge operator $\|$. This represents the parallel composition of two processes, the individual actions of which are interleaved. As an example, the process $(a \cdot b) \parallel (c \cdot d)$ may perform the actions a, b, c, d in any of the sequences abcd, acbd, acdb, cabd, cadb, cdab.
- *Communication* - pairs of atomic actions may be defined as communicating actions, implying they can not be performed on their own, but only together, when active in two parallel processes. This way, the two processes synchronize, and they may exchange data.

ACP fundamentally adopts an axiomatic, algebraic approach to the formal definition of its various operators. Using the alternative and sequential composition operators, ACP defines a basic process algebra which satisfies the following axioms:

$$\begin{aligned}
x + y &= y + x \\
(x + y) + z &= x + (y + z) \\
x + x &= x \\
(x + y) \cdot z &= x \cdot z + y \cdot z \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z)
\end{aligned}$$

The primitives 0 and 1, also known as δ and ϵ, behave much like the 0 and 1 that are usually neutral elements for addition and multiplication in algebra:

$$\begin{aligned}
0 + x &= x \\
0 \cdot x &= \delta \\
1 \cdot x &= x \\
x \cdot 1 &= x
\end{aligned}$$

There is no axiom for $x \cdot 0$.

$x + 1$ means: *optionally* x. This is illustrated by rewriting $(x + 1) \cdot y$ using the given axioms:

$$\begin{aligned}
(x + 1) \cdot y &= x \cdot y + 1 \cdot y \\
&= x \cdot y + y
\end{aligned}$$

The parallel merge operator $\|$ is defined in terms of the alternative and sequential composition operators. This definition also requires two auxiliary operators:

$$x \parallel y = x \mathbin{\|\!\!\_} y + y \mathbin{\|\!\!\_} x + x | y$$

- $x \mathbin{\|\!\!\_} y$ - "left-merge": x starts with an action, and then the rest of x is done in parallel with y.
- $x|y$ - "communication merge": x and y start with a communication (as a pair of atomic actions), and then the rest of x is done in parallel with the rest of y.

The definitions of many new operators such as the left merge operator use a special property of closed process expressions with $\cdot$ and

$+$: with the axioms as term rewrite rules from left to right (except for the commutativity axiom for $+$), each such expression reduces into one of the following normal forms: $(x + y)$, $a \cdot x$, 1, 0. E.g. the axioms for the left merge operator are:

$$\begin{aligned}(x + y)\llfloor z &= x\llfloor z + y\llfloor z\\ a \cdot x\llfloor y &= a \cdot (x\llfloor y)\\ 1\llfloor x &= 0\\ 0\llfloor x &= 0\end{aligned}$$

Again these axioms may be applied as term rewrite rules so that each closed expression with the parallel merge operator $\|$ reduces to one of the four normal forms. This way it has been possible to extend ACP with many new operators that are defined precisely in terms of sequence and choice, e.g. interrupt and disrupt operators, process launching, and notions of time and priorities.
Since its inception in 1982, ACP has successfully been applied to the specification and verification of among others, communication protocols, traffic systems and manufacturing plants.
In 1989, Henk Goeman unified Lambda Calculus with process expressions [**?**]. Shortly thereafter, Robin Milner et al developed Pi-calculus [**?**], which also combines the two theories.

3. SUBSCRIPT

SubScript extends Scala with a construct named "script". This is a counterpart of ACP process refinements, that coexists with variables and methods in classes. The body of a script is an expression like the ACP process expressions.

3.1 Notation

Esthetically, ACP processes are preferably notated with the mathematical expression syntax. However, ACP symbols $\cdot$, $\|$ are hard to type; for a programming language ASCII based alternatives are preferred. SubScript therefore applies a semicolon (`;`) and ampersand (`&`) for sequential and parallel composition. The special ACP processes 0 and 1 would clash with the usual notation for numbers; therefore these are replaced by symbols: `(-)` and `(+)`.
As with multiplication in math, the symbol for sequence may also be omitted, but then some white space should separate the operands. As usual, the semicolon should have low precedence, and the white space operator should have high precedence. This way one can get rid of parentheses. Instead of `(a;b)+c; d` and `(a b + c) d` one could write `a b + c; d`.
Scripts are usually defined together in a section, e.g.,

```
script..
  hello =          print("Hello,")
  test  = hello & print("world!")
```

From here on the section header `script..` is mostly omitted for brevity.

3.2 Parallelism

Between `hello` and `print("world!")` is a parallel operator. Each operand essentially contains a simple code fragment rather than code to be run in a separate thread. Therefore one operand will be executed before the other; the result is either "Hello, world!" or "world!Hello,". In general the atomic actions in concurrent processes are shuffle merged, like one can shuffle card decks.
In the "hello world" example the most straightforward execution strategy will deterministically apply a left-to-right precedence for the code fragments that are operands to the parallel operator `&`. However, alternative strategies are possible, e.g. for random simulations.
SubScript offers more forms of parallelism. Most notably the operator `||` denotes *strong or-parallelism*: it succeeds when any operand has success; it terminates when any operand terminates successfully.

3.3 Deterministic control and Iteration

SubScript has `if-else` and `match` constructs like the ones in Scala. Six operand types support iterations and breaking:

`while`	marks a loop and a conditional mandatory break
`for`	much like `while` and like the Scala for-comprehension
`...`	marks a loop; no break point, at least not here
`..`	marks a loop, and at the same time an optional break
`.`	an optional break point
`break`	a mandatory break point

3.4 Method Calls and Script Calls

Both scripts call the method `print`. Actually this is a shorthand notation for a fragment of Scala code that normally appears between braces, as in `{print("Hello,")}`. The body of the script `test` contains also a script call to `hello`.
A SubScript implementation will translate each script into a method. This way most Scala language features for methods also apply to scripts: scripts may have both type parameters and data parameters; each parameter may be named or implicit. Variable length parameters and even script currying are possible.

3.5 Result Values

Scripts may also have result values, which are comparable to method return values. A difference is that a method returns only once, whereas the script result value is available to the caller each time that the script has a success; this may be more than once, due to the 1-element of ACP.
The following scripts each have result type Int:

```
s1:Int = {5}
s2 = s1
s3 = s2; {5}^
s4 = {5}^v; {v}
```

The first script has its result type explicitly stated; for the others the type is inferred. The following type inference rules apply: `Script[T]`

- If the script expression is just a code fragment or a script call or a method call then the result type thereof is used as the inferred type
- If some code fragments and calls to methods and scripts in the script expression are suffixed by a caret (`^`), then the type is inferred from those fragments and calls.

At the time of writing it was not yet clear how the result type will be determined in other cases.
In `{5}^v` the result of the code fragment is stored in a local variable named `v`. As no declaration for this name was in scope, the fragment also implicitly declares the variable here.

3.6 Interoperation with Scala code

Scripts interoperate with Scala code in several ways:

- Any fragment of Scala code placed between pairs of braces, and variations thereof, may serve as an operand in a process expression. The start and end of such fragments will correspond with atomic actions in ACP. This way the code fragments may overlap, which is useful when they are run in separate threads, or when they denote actions that take some simulation time in a discrete event simulation context.

- In such code fragments and in calls to methods, Scala expressions may use a special value named `here`. It refers to the current node in the call graph, like `this` refers to the current object. `here` is in particular useful for implementing event handling scripts. For convenience it is an implicit value so that it may be left out of parameter lists containing an implicit formal parameter having the node type.
- Script expressions placed between rectangular brackets, such as `[{5}]` and `[(a;b)+c; d]`, are values of type `Script[T]` for an inferred type T. Such values are essentially closures; they may act inside a script expression as script calls.
- From Scala code a so called *script executor* may execute a script closure. The executor may be tailored for the type of application, e.g. discrete event simulations. After the execution ends, the executor may provide information on the execution, e.g. on whether the script ended successfully or as deadlock (δ).

The following would be a bridge method:

```
def testBridge = subscript.DSL._execute([test])
```

The DSL method `_execute` creates a fresh CommonExecutor (the default executor type) and then calls its run method with the script closure. Other types of executors could be more suited for specific application domains, such as discrete event simulations and multicore parallelism. With any given executor a script may be run as

```
  executor.run([test])
```

4. CALL GRAPH SEMANTICS

A SubScript program is executed by an executor, which will maintain a call graph, which is at run time derived from the static structure of the invoked scripts.

This static structure may be represented by so-called "Template trees". For instance, consider the following process which prints optionally "Hello", and then "world!":

```
Main = . print("Hello ");
         print("world!")
```

Figure 1 gives the template tree. At run time this tree is used to grow and prune the call graph, as shown in figure 2. The predecessor paper contains a more detailed explanation.

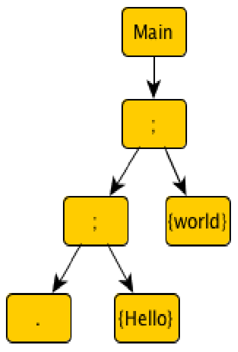

Figure 1: Template Tree

5. BASIC IMPLEMENTATION

At first SubScript had been implemented as a domain specific language (DSL); the so called SubScript Virtual Machine executes scripts by internally doing graph manipulation. The VM has been programmed using 2000 lines of Scala code. This is not a complete

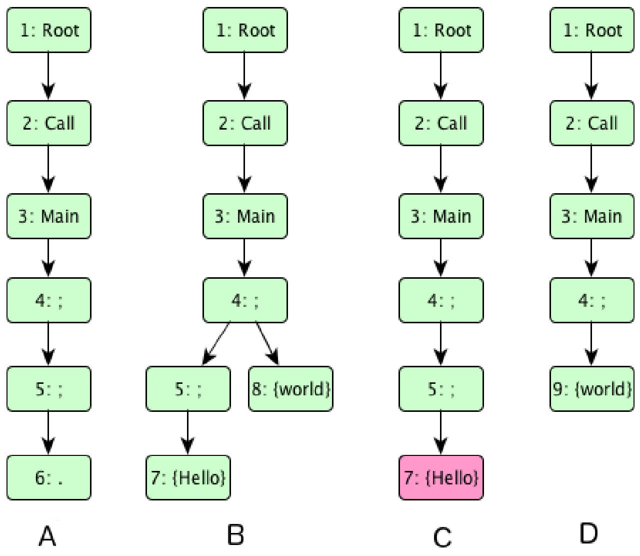

Figure 2: Call Graphs

implementation; most notably support for ACP style communication is still to be done. When complete the VM may contain about 4000 lines.

In principle the DSL suffices for writing the essence of SubScript programs. However, with the special syntax, e.g. for parameter lists, n-ary infix operators, various flavors of code fragments, specifications become considerably smaller and these require much less parentheses and braces (which is also important for clarity).

A special branch of the Scala compiler was modified so that it translates the genuine SubScript syntax to the DSL. This took about 2000 lines of Scala code, mainly in the scanner, the parser and the typer.

6. EXAMPLE: SUBSCRIPT-ACTORS

We build a SubScript-Actor system, in which one actor, a front processor, performs computational tasks on requests, by splitting these up and deferring them to other processor actors. After the front processor has received all results from the processor actors, it aggregates these results and sends those back to the original task requester. From now, we will use `Di, Df, Rf, Ri` data types to represent input data, forked data fragments, forked result fragments and output result respectively. At any time during processing a new task may arrive; this disrupts ongoing data processing, if any.

The communication between the front processor and the other processors is unreliable. For each of such processors there is a proxy actor, which is supposed to communicate more reliably with the front processor. On a new task, each processor first sends a receipt acknowledgement back; then it does the potentially time consuming operation; and finally it replies the result thereof.

Things may go wrong:

- If a proxy does not get receipt acknowledgement within a second, it will resend the task to the processor; up to at most 3 times.
- If there is still no acknowledgement after these tries, the proxy sends a failure message back to the front processor.
- The front processor will also reply a failure to the original requester when it receives such a failure message
- It will also do so when it has not received all processor results within 10 seconds.

This is not necessarily a good design for a task processor system; we aim to show how to specify a non-trivial SubScript actor system.

6.1 Processor

The Processor class has the following `live` script:

```
live = task/.. ; ...
```

As in Scala, the semicolon denotes sequence but it is here an operator rather than a statement separator. SubScript operator priorities are the same as in Scala, with the semicolon having the lowest priority. The following expression with parameters would be equivalent: `(task/..)  ; ....`
Behind the semicolon there are three dots (`...`), denoting an iterator: it turns the sequence into a loop, without specifying a possible exit point. So `live` is an eternal loop of `task/..` .
The slash there is a disruption operator: the left hand side happens, possibly interrupted by the right hand side. This right hand side (`..`) is an iterator which also denotes an optional break. It has the following effect in combination with the disruption construct:

- first a `task` is activated
- then the optional break is activated. This turns the disruption construct into a loop, but the next round of the loop is not yet activated
- as soon as an atomic action in `task` happens, another instance of `task` is activated, so that this may disrupt the ongoing one

So during its life time a processor will do tasks; if it is busy with a task that will be disrupted in case a new task has to start. A new task starts when a `Task` message arrives:

```
task = << Task(data: Df, id)
       => val tasker = sender
          tasker!ConfirmReceipt(id)
      ==> {*process(data)*}
      ~~> {tasker ! Result(id, _)}
       >>
```

The brackets `<<....>>` denote a message handler; on the inside there is essentially a partial function, like in the usual `receive` methods of actors. Multiple of such message handlers may be active at the same time; these handlers are considered when Akka gives an message to the actor for handling. If any such active message handler has a partial function that is defined for the message, it will handle it, and the actor's `receive` method will not be called. In this example the partial function for the message handler has only one component; therefore the message handler syntax allows the `case` keyword to be omitted.
The part to the left of the long fat arrow (==>) is in ACP terms an atomic action corresponding the initial handling of the message. The script expression to the right, `{*process(data)*}    > {tasker!Result(id, _)}`, denotes what happens next.
This fragment contains a long curly arrow, meaning data flow. The asterisks in `{*process(data)*}` mark a *threaded code fragment*: this is executed in a separate thread. The return value of `process(data)` becomes the result of the code fragment. Then this result value is taken by the arrow to serve as input parameter for operand at the right: `{tasker!Result(id, _)}`. This is essentially a script closure like
`(p:T) => [{tasker !  Result(id, p)}]`.

6.2 Proxy

A high level version of the live script for Proxy actors is:

```
live = receive_task
        ( ( times(3)
              {target!currentMessage} sleep(1 second)
          ) fail
        / receive_confirmation
          ( handle_result || after(7 second) fail )
        )
      / ..
```

`times(i:Int)` is an iterator script; it is defined in object `subscript.Predef` as `while(pass<i)`. `pass` is a method that implicitly accepts the `here` value; it returns the "pass number" of the nearest ancestor operator that may be an iteration.
The `after` script is inherited from SubScript actor; it performs an empty atomic action after the duration specified by its parameter. It is placed in an or-parallel composition with `handle_result` using the operator `||`; this means that whichever of `after` and `handle_result` terminates successfully will cause this composition to terminate successfully.
Other scripts that `live` calls, such as `receive_task` would set and use instance variables such as

```
var currentMessage: AnyRef  = null
val taskRequester: ActorRef = null
```

It may be worthwhile to get rid of such instance variables by inlining the other scripts such as `receive_task`, into a lower level version of such as `live`:

```
live = << currentTask @ Task(data, id)
       => val requester = sender
          val TaskId = id // upper case for below

      ==> ( ( times(3)
                {target!currentTask} sleep(1 second)
              ) {requester ! Failure(TaskId)}
            /
              <<ConfirmReceipt(TaskId)>>
              ( <<r@Result(TaskId,Some(data))
                 => requester ! r; reset(None)>>
              /
                 after(7 second)
                 {requester ! Failure(TaskId)} )
              )
          >>
          / ..
```

Note that the message handlers (`<<....>>`) are now nested. For `<<ReceiptConfirmation(TaskId)>>` there was nothing to do in the partial function; for sake of brevity the arrow => could be omitted.

6.3 Front Processor

The Front Processor receives tasks and splits these up; then it sends the partial tasks to the proxies, using Akka's `ask` method. This yields a collection of Futures that will produce the replied data. Then a SubScript process starts an and-parallel loop over all these futures.

```
live = (task / ..; ...) || config...

task = << Task(data: Di, taskId)
        => val taskRequester = sender
           var responses    : List[Rf] = Nil
           var taskRequester: ActorRef = null
           val fd:Seq[Df]=fork(data,processors.size)
           val futureAnswers =
              (proxies zip fd) map{
                (p, d) => p ? Task(d, taskId))
              }

      ==>
        ( for (fa<-futureAnswers) &&
          ( fa
           ~~> _.result match (
                case s@Success[(Int,Rf)]
                     => responses :+= s._2
                case f@Failure
                     => taskRequester ! Failure(f)}
                    ==> (-)
              )
          )
        )
        {taskRequester
               ! Success((taskId, join(responses)))}
       / after(10 second)
         {taskRequester ! Failure(TaskId)}
       >>

config = << Configuration(processors)
          => setProxies(proxies); sender!Ready
          >>
```

Scripts have some features in common with futures: they carry data and handle concurrency; they support fork-join paradigm, and they have notions of success and failure. Here a failure of a script means that it terminates without having success. However, while a future completes once, with either a success or a failure, a script may have success more than once; and a failure may occur after a success, like in the ACP process $1 + x \cdot 0$.
It is possible to specify a future as a kind of script call in a script expression, using an implicit conversion method. Likewise a script closure may be implicitly be converted to a future. success and failures. But more than once success possible.

7. GOOGLE SEARCHES

In this example, we will demonstrate application of SubScript to the problem of simple Google Search application. This is an application which provides user with capabilities to search over the Internet with the help of Google. Also, an application provides user with a list of suggestion as he or she types the search query. Suggestions shouldn't be confused with search results: suggestions try to complete the incomplete search query as user types; search results are links returned by a search engine on a particular query.

7.1 GUI and Search API

Our sample application will have a simple GUI that contains:

- A text field for the user to type his or her search query
- A suggestions box with a list of suggestions. The items in this box are clickable; when a user clicks on some suggestion an event is generated.

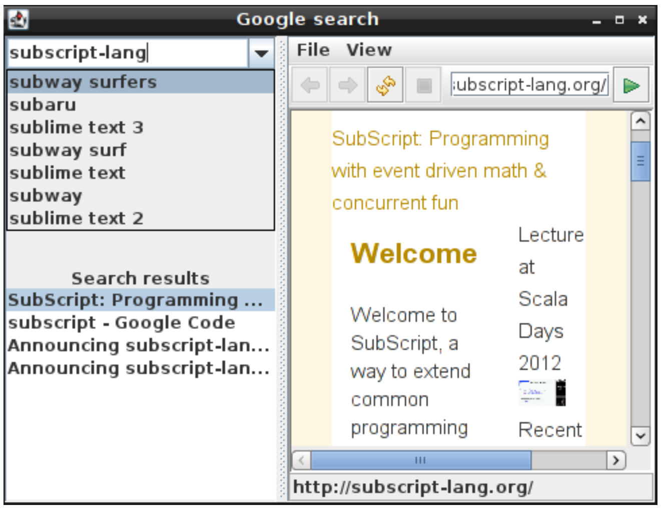
Google search
subscript-lang
subway surfers
subaru
sublime text 3
subway surf
sublime text
subway
sublime text 2
Search results
SubScript: Programming ...
subscript - Google Code
Announcing subscript-lan...
Announcing subscript-lan...
File View
ubscript-lang.org/
SubScript: Programming with event driven math & concurrent fun
Welcome
Welcome to SubScript, a way to extend common programming
Lecture at Scala Days 2012
Recent
http://subscript-lang.org/

Figure 3: Google Search GUI

- A search box with a list of search results. The items in this box are also clickable.
- An embedded browser window that will display a website to user after he/she clicks on some search result.

Most of modern GUI libraries harness the Observer pattern: other objects, called Observers, can register themselves at GUI objects to receive various events via callbacks.

More complex systems can be built using this pattern. From a callback-registering method a `Future[T]` may be produced, where `T` is the type of some event-specific data.

Thus we can safely put the following methods into the GUI trait with the guarantee that they can be implemented in vast majority of GUI libraries:

```
def textChange         : Future[String]
def suggestionsClick   : Future[String]
def searchResultsClick: Future[String]
```

Here, `textChange` represents a `Future` that will be completed once the input string in the search text field experiences some changes. Meaning, a new character inputed by user will complete this `Future`. This `Future` will be completed with the new input string as its result.

`suggestionsClick` represents a `Future` that will be completed on mouse click event that happened on some suggestion from the suggestion list box. This `Future` will be compteted with the text of the selected suggestion.

`searchResultsClick` is similar to `suggestionsClick`, but it responds to clicks on search results rather then suggestions. It is completed by the clicked URL address.

Also, the application requires means to provide output to the user. Specifically, methods for setting current list of suggestions, search results and navigating the embedded browser to a given URL are required:

```
def setInputString   (str:     String ): Unit
def setSuggestions   (ls : Seq[String]): Unit
def setSearchResults(ls : Seq[String]): Unit
def setBrowserUrl    (url:     String ): Unit
```

This comprises a GUI interface of the application. It is possible to create concrete implementations of such an interface in the majority of GUI libraries for Java/Scala, such as Swing.

Also, an API for doing actual Google search and retrieving suggestions will be required:

```
def suggestions(req: String): Future[Seq[String]]
def search     (req: String): Future[Seq[String]]
```

They will accept a `String` request as an argument and will return a `Future` of the required result.

7.2 Reactive flow specification

Given the API mentioned above, it is easy to describe the lifecycle of the application in SubScript:

```
live = textChange ~~> (req: String) => [
          suggestions(req)~~>setSuggestions(_);
          search     (req)~~>setSearchResults(_)
       ]
    || suggestionsClick    ~~>setInputString(_)
    || searchResultsClick  ~~>setBrowserUrl(_)
    ; ...
```

This is an eternal sequential loop (`; ...`) of event handling expressions. There are three kinds of events, handled in an or-parallel composition (`||`).

The first operand defines the handling of changes in the input text field. `textChange` returns a `Future[String]`. This `Future` is converted using an implicit conversion. Once it has success, the right-hand operand of the dataflow operator > is activated. This operand is a script closure, that receives the result of `textChange` as actual parameter value. Its retrieves suggestions and updates the GUI correspondingly; then it retrieves search results and again updates the GUI. `suggestions` and `search` are also `Futures` (this time from the Search API), converted into scripts using implicit conversions.

The other two operands of `||` respond to clicks on the presented suggestions and on search results, by updating the GUI. Again, futures are implicitly converted into scripts; and the results are transmitted using the dataflow operator into the GUI update code. These handlers take little time. Meanwhile a longer lasting text change handling may have been ongoing; then this would be disrupted because of the or-parallel composition.

8. IMPLEMENTATION

8.1 SubScript Actors Implementation

SubScript Actors are implemented in a separate trait, named `SubScriptActor`. This extends `akka.actor.Actor` and overrides some methods.

First such overriden method is `aroundPreStart`. It is called by Akka before actor starts and its main job is to register and start the actor within the SubScript virtual machine (SVM) that is responsible for running this actor. Precisely, it starts the actor's lifecycle from the SVM:

```
override def aroundPreStart() {
  runner.launch([lifecycle])
  super.aroundPreStart()
}
```

The `lifecycle` script is also defined in this trait and represents the lifecycle of the actor:

```
lifecycle = live || terminate; die
```

The `lifecycle` starts two processes in parallel. `live` process is supposed to be overridden by the end user and is abstract in this trait. It specifies the logic of the actor, such as what messages can it handle and how. `terminate` process is a mean that allows to terminate the actor on demand. The actor can be terminated simply by completing `terminate` process: in this case, `live` will be excluded. After either `live` or `terminate` has completed, the `die` process starts. Its job is to do the final clean-up, it unloads the actor from the `ActorSystem` and the memory.

Another aspect that requires explanation is the `runner` that was used in `aroundPreStart` to launch the `lifecycle`. This object is of type `SubScriptActorRunner`. That is a trait that contains methods to launch and execute scripts; the purpose of this trait is very close to the `ScriptExecutor` ones. Its working implementation is an object `SSARunnerV1` - this is the runner that executes the SubScript scripts using Akka's scheduler rather then simple `while`-loop.

`SubScriptActor` uses multiple message handlers to handle its messages, as opposed to `akka.actor.Actor` that uses only one handler, `receive`. It contains the collection of such handlers:

```
private val callHandlers = ListBuffer[
    PartialFunction[Any, Unit]]()
```

Actor messages specified in message handlers need to be interpreted as SubScript atomic actions. Akka Actor class has hook functions such as aroundReceive; usual receive processing replaced by SubScript action handling.

Normally SVM executes a script in a loop that handles call graph message synchronously sequentially. There are alternatives, as the body of the loop is a method, `tryHandleMessage`, that may as well be called from outside. For the purpose of running SubScript Actors, there are 3 options for the coexistence of Akka and the SVM:

- each in their own threads; use `wait` and `notify` for synchronization
- describe the coordination as a separate SubScript process, run in its own SVM
- `tryHandleMessage` in the SVM is invoked by timer callbacks and `aroundReceive`

For SubScript actors the third option is taken; the appendix lists the main source code.

The SubScript compiler translates a simple message handler `<<partialFn>>` (which has no `==> scriptExpr` parts) into

```
@{initForReceive(there, partialFn)}: {.  .}
```

`doScriptSteps_loop` does at most 1 background action (having a low priority) at a time. Other strategies are in principle also possible. Message processing will have higher priority.

Akka employs aspect-oriented programming to allow developers to extend default Actor's behavior. This is done by around-aspect methods such as `aroundReceive`, which defines the logic of Akka message handling by a particular actor.

In SubScript Actors the message handling logic needs to be different from usual Akka Actors, so class `SubScriptActor` overrides this method. First, `aroundReceive` lets the SVM do its handling using calls `runner.doScriptSteps`; then it does the actual Akka message handling. It does not use the standard `receive` method for this.
A SubScriptActor contains a collection of message handler functions of type `PartialFunction[Any, Unit]`, that is man-

aged by the SVM. aroundReceive looks for a handler capable to handle the message:

```
runner.doScriptSteps
callHandlers.collectFirst {
  case handler if handler isDefinedAt msg
    => handler(msg) }
match {
  case None     => super.aroundReceive(receive, msg)
  case Some(_)  => super.aroundReceive(
                            {case _: Any =>}, msg)
                  runner.doScriptSteps
}
```

If a message had been handled here, Akka should not cause it to be handled again; yet super.aroundReceive needs to be called. This is done by supplying as "receive" parameter a partial function that can handle any message and that does nothing when the message has been handled.
If the message had not been handled, the actor can handle the message in its regular receive method. This is a fall back option with lightweight message handling that is supposed not to interfere with the handling in scripts.

A general message handler is of the form

```
<< case p_1 => scalaCode_1 ==> subScriptExpr_1
   case p_2 => scalaCode_2 ==> subScriptExpr_2
   ....
   case p_n => scalaCode_n ==> subScriptExpr_n >>
```

On lines without scalaCode or a subScriptExpr the corresponding arrows may be omitted.

```
var s:Script[T]=null
<< case p_1 => scalaCode_1; s=[subScriptExpr_1]
   case p_2 => scalaCode_2; s=[subScriptExpr_2]
   ....
   case p_n => scalaCode_n; s=[subScriptExpr_n] >>
if (s!=null) s
```

Here T is the most specific super type of the result types of the subScriptExprs.

8.2 Data Flow

In {*process*} ~~> {tasker!Result(id, _)} the arrow is a data flow operator: a kind of sequential composition that captures the result of the left hand side, and passes that on to the right hand side as a parameter. Like with normal Scala code, the underscore turns a script expression into a script lambda with a parameter.
The following translations are done, with T as the return type of process:

```
{*process*} ~~> {tasker!Result(id, _)}

{*process*} ~~> (p:T => [{tasker!Result(id, p)}])

do {*process*}^v
then (p:T => [{tasker!Result(id, p)}])(v)

var v:T=null;
do @{there.onSuccess{v=there.result}}: {*process*}
then (p:T => [{tasker!Result(id, p)}])(v)
```

The construct ^v captures a result value into a variable v, for which implicitly a declaration is made, in case a declaration was not yet in scope.

There is ternary construct do x then y else z which means: do x; after x has success y is activated. In case x deactivates without success then z is activated. Both the then-part and the else part are optional, but not at the same time. If the else-part is absent, do-then is a binary sequential operator; it cannot iterate, so if it has an iterator operand then that affects an n-ary operator higher in the call graph.

In the final translated version, the ; sequential operator is used to declare a local variable v as its first operand and then proceed to the actual script execution. The second operand of ; is a then sequential operator application.

The left-hand side operand of the do-then construct is an annotation applied to left-hand side operand, {*process*}, of the original > operator. The annotation registers an onSuccess callback in this operand, so that when it finishes execution its result value becomes stored at the v variable.

The right-hand side operand of the do-then construct is a script closure applied to the v variable. The v variable contains the result of the original {*process*} script by the time of the application due to the sequential nature of ; and do-then operators.

8.3 Interoperation with Futures

Scala's Futures and SubScript processes are fundamentally similar. They both represent a potentially time-consuming operation that results in some value (success) or in an exception (failure).

The following implicit conversion from Futures to processes would allow one to use Futures in SubScript expressions just like any other call:

```
implicit def script conversion[T](f: Future[T]):
    Script[T]
  = @{
       f.onSuccess {case d =>
         here.script.result = d
         there.execute
       }
       f.onFailure{case e =>
         here.script.failure = e
         there.fail
       }
     }: {. .}
```

Here, the Future is transformed into an annotated event-handling code fragment ({.   .}). Such a code fragment is not automatically executed by the SubScript Virtual Machine, but instead by external code, through a call to the method execute.

The code inside the annotation registers two callbacks within the converted future:

- when the future succeeds, the corresponding callback sets the result value of the current script (conversion) to the result of the Future; then it executes the event-handling code fragment.
- when the future fails, the corresponding callback sets the failure value of the current script; then it makes the event-handling code fragment fail.

The once activated such a conversion script might be deactivated because of another process, before the future had been completed. One might expect that the future then would be cancelled; however such an operation is currently not supported in the Scala Futures.

Conversely it is possible to convert a script closure to a future:

```
implicit def convert[T](s: Script[T]): Future[T] =
{
  val p = new Promise[T]
  val s1 =
    [ @{there.onSuccess{p success there.result}
       {there.onDeactivation{ if(!there.hasSuccess)
                         p failure there.failure}}:
       s
    ]
  AkkaScriptRunner.execute(s1)
  p.future
}
```

Here `AkkaScriptRunner` is much like the earlier discussed `SSARunnerV1`: it may execute a given script in cooperation with Akka's actor system. During this execution it adds an annotation the script that registers handlers for success and failure: on such occasions that the corresponding values are transferred from the script to the future.

A problem with this conversion is that the script may have multiple times success, so that the promise's success method may be called multiple times. Some extra logic may prevent this, but there is anyway a kind of impedance mismatch. Anyway we don't know yet whether the conversion from scripts to futures will be useful or not.

9. PERFORMANCE

We measured performance of using a Sieve of Eratosthenes application, in which each prime sieve was implemented by an actor process. On an iMac with a 2.7GHz Intel Core i5, the plain Scala version got a throughput of about 1,000,000 messages per second, whereas the SubScript version only got about 10,000 messages per second. The performance penalty factor is therefore about 100. We expect to improve performance considerably by relatively small modifications, but we estimate that this way the gap will remain at least a factor 25.

One reason for this can be inefficient mechanics of script processing by the virtual machine. Currently, all the processing relies heavily on message passing within the virtual machine. It is possible, that the a good fraction of such messages are "noise", meaning they don't carry any useful information and just throttle execution speed.

In principle it seems possible that a SubScript virtual machine follows a different execution strategy: the compiler or VM would analyze a behavior specification and transform it into a finite state machine representation. Then the SubScript actors performance could much be much closer to the one of plain Scala actors.

10. RELATED WORK

The predecessor paper contains an overview of other languages that show some resemblance to SubScript. SubScript seems to be unique in allowing for an process algebra approach to actor programming.

There is some noticeable work with process algebra as a theoretical underpinning to actors. E.g., the following papers apply Pi-calculus: [?], [?]. [?] applies ACP to define actor semantics.

11. CONCLUSION

SubScript offers constructs from the Algebra of Communicating Processes that apply well to reactive programming. It helps to concisely specify internal actor behavior.
There is a performance penalty. Whether that is a problem depends on performance requirements, and on the proportion of CPU time required for internal actions to the total CPU time.
Futures may conveniently placed in SubScript process expressions. Likewise SubScript processes may be converted into futures, but there is an "impedance mismatch". A variant of Futures that supports a kind of 1-element from ACP, could be interesting.
Further R&D on ACP, SubScript and Actor systems could focus on design, prototyping, performance analysis, testing and formal reasoning.
SubScript is an open source project[5]. It is currently implemented as a branch of the regular Scala compiler, bundled with a virtual machine and a library for interfacing with AKKA actors and Swing GUIs. We are considering replacing the regular Scala compiler with the compiler front end named SugarScala [?].

12. ACKNOWLEDGEMENT

Anatoliy Kmetyuk receives a Google Summer of Code stipend in 2014 for his work on the project.

APPENDIX

CommonScriptExecutor Main Loop

```
def run(s: CallGraphNode._scriptType[_]) = {
  initializeExecution(s)
  while (hasActiveProcesses) { // main execution loop
    if (tryHandleMessage(Int.MinValue)==null)
     awaitMessages
  }
}
// minAAPriority => what atomic actions are handled
// Int.MinValue — allow for background task processing
// 0 — allow only foreground task processing
def tryHandleMessage(minAAPriority:Int): CallGraphMsg = {
  val m = dequeueCallGraphMsg(minAAPriority)
  if (m == null) return null
  messageHandled(m); handle(m); m
}
```

Akka-SubScript Bridge Code

```
trait SubScriptActorRunner {
  def system: ActorSystem
  def executor: CommonScriptExecutor
  def launch(s: Script[_])
  def execute(debugger: ScriptDebugger)
  def doScriptSteps
}

object SSARunnerV1 extends SubScriptActorRunner {
  lazy val system = ActorSystem()
  lazy val executor = ScriptExecutorFactory.
     createScriptExecutor(true)
  var launch_anchor: N_launch_anchor = null

  def scheduledTaskDelay = 0.01 milliseconds
  def launch(s: Script[_]) = launch_anchor.launch(s)

  script live =@{launch_anchor=there}: (** {. .} **)
```

[5]Subscript web site: http://subscript-lang.org

```
  def execute(debugger: ScriptDebugger) {
    if (debugger!=null) debugger.attach(executor)
    executor.initializeExecution(_live())
    doScriptSteps_loop
  }

  def doScriptSteps_loop: Unit = {
    doScriptSteps
    if (executor.hasActiveProcesses) {
      system.scheduler.scheduleOnce(scheduledTaskDelay)(
     doScriptSteps_loop)
    }
  }

  var doScriptSteps_running = 0
  def doScriptSteps = {
    synchronized {
      if (doScriptSteps_running==0) {
        doScriptSteps_running += 1
        try {
          var handledMsg =
                executor.tryHandleMessage(Int.MinValue)
          while (handledMsg!=null) {
            handledMsg = executor.tryHandleMessage(0)
          }
          executor.messageAwaiting
        }
        finally {doScriptSteps_running -= 1}
      }
    }
  }
}
```

```
trait SubScriptActor extends Actor {

  val runner: SubScriptActorRunner = SSARunnerV1

  private object Terminator {
    var executor: EventHandlingCodeFragmentExecutor[
                                  N_atomic_action] = null

    script block
    = @{executor=new EventHandlingCodeFragmentExecutor(
                  there,there.scriptExecutor)}: {. .}
    def release = executor.executeMatching(isMatching=
     true)
  }

  private val callHandlers =
      ListBuffer[PartialFunction[Any, Unit]]()

  script..
    live
    after(d: Duration) =
        @{akka.pattern.after(d,system.scheduler)
                          (Future{there.execute})}:
         {..}

  private script..
     die       = {if (context ne null) context stop self}
     lifecycle = live || terminate; die
     terminate = .... /* when aroundPostStop is called */

  override def aroundPreStart() {
      runner.launch([lifecycle]); super.aroundPreStart()}

  override def aroundPostStop() {
    .... /* make script terminate happen*/
    super.aroundPostStop()
  }
```

```
  override def aroundReceive(receive:Receive, msg:Any) {
    runner.doScriptSteps
    callHandlers.collectFirst {
      case handler if handler isDefinedAt msg
        => handler(msg) }
    match {
      case None     => super.aroundReceive(receive, msg)
      case Some(_) => super.aroundReceive(
                              {case _: Any =>}, msg)
                    runner.doScriptSteps
  }

  // SubScript actor convenience methods
  def initForReceive(node: N_code_eventhandling,
                 _handler: PartialFunction[Any, Unit])
  {
    node.codeExecutor = EventHandlingCodeFragmentExecutor
                          (node, node.scriptExecutor)
    val handler = _handler andThen
                    {_ => node.codeExecutor.executeAA}
    synchronized {callHandlers += handler}
    node.onDeactivate {
      synchronized {callHandlers -= handler}
    }
  }
}
```