# SubScript: Extending Scala with the Algebra of Communicating Processes

André van Delft

andre dot vandelft at gmail dot com

## Abstract

Most programming languages offer relatively little or no support for parallelism and non-determinism. Support would in particular be very useful for specifying event handling and background processing in applications with graphical user interfaces, and for specifying grammars of input data. To improve the situation, programming languages may be extended with constructs adopted from the theory named Algebra of Communicating Processes (ACP). This has been done in SubScript, which is an extension to Scala. Examples show how SubScript is useful for programming GUI controllers in the MVC paradigm.

ACP-like constructs and some syntactic sugar suddenly enable process descriptions with an expressive power comparable to the power of parser generators, logic and Unix pipes. This applied math allows for modeling the behavior of GUI controllers very clearly and concisely. We claim that this will lead to higher reliability against less cost, but foremost: more fun.

Currently SubScript is implemented as a DSL, in about 2000 lines of Scala code. This maintains a call graph, that grows and shrinks as a SubScript program runs. The semantics of the language is mainly determined in terms of the rules for this graph manipulation. SubScript implementations have some freedom in these rules, so that they can specialize, e.g. for real time constraints, probabilities, timed simulations and parallel processing.

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming

***General Terms*** Languages, Theory

***Keywords*** Process Algebra, Algebra of Communicating Processes, parallel programming, non-determinism, GUI programming, MVC, Scala

## 1. Introduction

*Our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

These wise words are from the famous paper *Goto statement considered harmful* that Edsger Dijkstra wrote in 1968 [2]. In those days many programmers tended to use the *goto* statement, which too often obscured the control flow. The resulting programs were described as *spaghetti code* . In the years after this paper, the use of goto statements decreased, also by improving choice and iteration constructs in programming languages. By 1980, *structured programming* had become widely accepted.

But spaghetti code returned, although this was not immediately recognized. Programs got graphical user interfaces, and program behavior started to be driven by user generated events. At first, a *main event loop* would handle these events. Programming such an event loop was tedious, but the code was still understandable. As the user interfaces became more advanced, the coding complexity increased. For instance, time consuming tasks required special care so that they would not block user input.

Around the year 2000 programming interactive applications had become really hard. Spaghetti code was needed to handle all kinds of input events; the screen needed to be updated in a specific thread, and background threads had to keep the application responsive. As a result, too many applications, even professional ones, nowadays freeze the GUI time and again.

The main cause for the trouble is that the applied programming languages offer inappropriate support for event-driven and parallel programming. Event handling and multithreading are usually implemented using dynamically created objects. Manipulating these data items largely determines the flow of control. This is much less clear than the use of explicit control flow constructs, that programming languages offer for concepts such as choice and iteration.

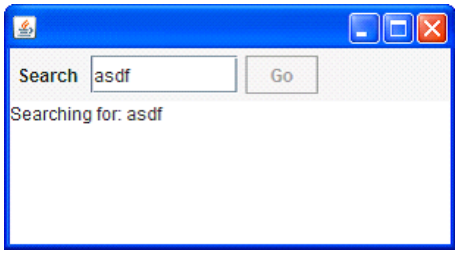
This paper presents SubScript: a Scala extension with such constructs, inspired by the Algebra of Communicating Processes (ACP). Sections 2 and 3 present examples that gives a global impression of the language. Next ACP is discussed, and how it can extend Scala.

## 2. Example: A Simple GUI Application

Suppose we need a simple program to look up items in a database, based on a search string.

The user can enter a search string in the text field and then press the Go button. This will at first put a ”Searching” message in the text area at the lower part. Then the actual search will be performed at a database, which may take a few seconds (simulated by a call to Thread.sleep). Finally the results from the database are shown in the text area.

If you would program this functionality in plain Scala, the resulting code would be like:




```
1  val searchButton = new Button("Go")    {
2    reactions.+= {
3      case ButtonClicked(b) =>
4        enabled = false
5        outputTA.text = "Starting search.."
6        new Thread(new Runnable {
7         def run() {
8          Thread.sleep(3000)
9          SwingUtilities.invokeLater(new Runnable{
10           def run() {outputTA.text="Search ready"
11                      enabled = true
12         }})
13       }}).start
14   }
15 }
```

Here `outputTA` denotes the output text area. A solution in Java would be similar. This code looks very technical: lots of indentations and braces. The control flow is hidden in nested functions. Parallelism is done by calling a method `start` on a Thread object. This looks like a usual method call, but something magic happens inside. Parallelism does not get a similar basic treatment as statement sequences do.

The order in which the lines are executed is spaghetti-like:

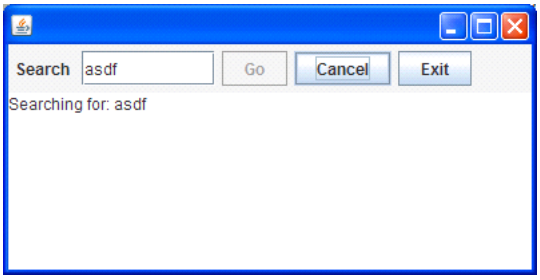
- Lines 1 and 2 are done during initialization, in the main thread.
- Line 3 to 6 and 13 are call back statements, executed when the button is pressed. Line 4 and 5 must be executed in the Swing thread; this happens to be the case with the call back, so no special provision needs to be taken.
- On the execution of line 13, a background thread starts that will execute line 8
- When the background thread has finished line 8, it schedules line 10 and 11 for execution in the Swing thread.

To paraphrase Dijkstra's earlier quote: the conceptual gap between the static program text and the dynamic process is far from trivial here. The programming task is hard and boring. The result: many applications fail to appropriately enable and disable their GUI widgets, or they are not responsive, or they even hang every now and then. This situation is unnecessary.
The SubScript notation is much more concise and intuitive:

```
1  live = searchButton
2         @gui: {outputTA.text="Starting search.."}
3         {* Thread.sleep(3000) *}
4         @gui: {outputTA.text="Search ready"}
5         ...
```

Whitespace acts here as a hidden sequential composition operator.

- Line 1: `live` is a method like refinement called "script" for the controller behavior. It first silently uses an implicit script named `clicked` that gets the `searchButton` as a parameter. This `clicked` script "happens" when the user presses the search button. It is defined in a utility object `SubScript.swing.Scripts`. As a bonus, the script makes sure the button is exactly enabled when applicable, i.e. when the program is ready to handle a button click.
- Lines 2 and 4 each write a message in the text area. An annotation, `@gui:`, makes sure this happens in the Swing thread, as needed. `gui` is also defined in `SubScript.swing.Scripts`.
- Line 3 simulates the lasting database search using a sleep call. The asterisks next to the braces specify that this is done in a background thread, so that neither the GUI nor the main thread will be blocked meanwhile.
- Line 5 turns the foregoing into an "eternal" sequential loop (. . . , "etcetera") of search sequences




You may write the code using more refinements; e.g. starting with:

```
1  live              = searchSequence...
2  searchSequence    = searchCommand
3                      showSearchingText
4                      searchInDatabase
5                      showSearchResults
```

### 2.1 Extending the program

Let us add some realistic requirements to the program.

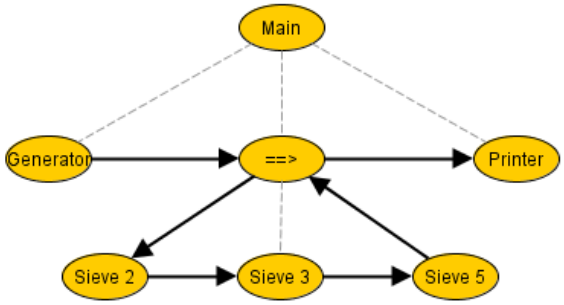
- The search action may also be triggered by the user pressing the Enter key in the search text field.
- The search action requires that the input text field is not empty; only then should the search button be enabled
- The user should be able to cancel an ongoing search, by clicking a Cancel button, or pressing the Escape key.
- As long as the database search is ongoing, the progress should be indicated: 4 times per second a number is appended to the output text area
- The user can exit the application by clicking an Exit button, or by clicking in the close box at the window's upper right corner. But exiting should first be confirmed in a dialog box.

In a Java or Scala version the application state would need to be kept in variables; updating these would be nontrivial. The progress indicator would be cumbersome and error-prone to program (and that is why it is rarely present). It is easier to grow the SubScript version.

```
1  live            = searchSequence... || exit
2
3  searchCommand   = searchButton + Key.Enter
4  cancelCommand   = cancelButton + Key.Escape
5  exitCommand     = exitButton   + windowClosing
6  exit            = exitCommand
7                      @gui: while(!confirmExit)
8  cancelSearch    = cancelCommand
9                      @gui: showCanceledText
10
```

```
11  searchGuard     = if(!searchTF.text.isEmpty) .
12                    anyEvent(searchTF)
13                    ...
14
15  searchSequence  = searchGuard
16                    searchCommand
17                  ; showSearchingText
18                    searchInDatabase
19                    showSearchResults
20                  / cancelSearch
21
22  showSearchingText = @gui: {outputTA.text =...}
23  showSearchResults = @gui: {outputTA.text =...}
24
25  searchInDatabase  = {*Thread.sleep(3000)*}
26                    || progressMonitor
27
28  progressMonitor   = {*Thread.sleep(250)*}
29                      @gui:{searchTF.text
30                                  += here.pass}
31                      ...
```

- Line 1, 6 and 7: the `||` denotes or-parallelism: both its operands become active, but the `live` script is ready when either one of its operands ends successfully.
  In this case, the left hand operand is an eternal loop of search sequences; the right hand operand is a (probably) finite loop of exit tries.
  One try consists of an exit command and then a confirmation dialog; this loop goes on while the dialog is canceled. The dialog is correctly executed in the Swing thread
- Lines 3 and 4: the `+` denotes exclusive choice. `Key.Enter` and `Key.Escape` cause calls to the implicit script `vkey`
- Line 5: `windowClosing` acts on window closing events; it is defined in `SubScript.swing.Scripts`.
- Line 11 to 16: `searchGuard` is an "active guard" containing a sequential loop.
  It first checks whether the text field contains some text. If so, there is an "optional break", specified by the dot. This means that the sequence and thus also the guard may end successfully, so that `searchCommand` in line 15 becomes active.
  However maybe an event happens at the text field before the user issues this search command; then the check needs to be redone, etc (`...`).
- Line 15 to 20: after the search command has been issued, the search starts, and meanwhile it should be possible to cancel the search.
  The slash operator (`/`) denotes optional disruption of its left hand operator by its right hand operator. Its left hand operator is the sequence from `showSearchingText` to `showSearchResults`. This is because the reach of the slash operator starts immediately after the semicolon at line 17.
  Both the semicolon and whitespace act as sequential operators: the former with low priority; the latter with high priority. The combination these two operators makes it possible to do without parentheses, and that makes the text better readable.
- Line 25 to 31: the database search has now an or-parallel construct with a `progressMonitor` process.
  The latter is an eternal loop of: wait 1/4 second; then append a loop counter to the output text field.
  The pseudo-value `here` denotes "the current operand"; it is comparable to `this`, the "current object". Its field `pass` yields 0, 1, 2, ... in subsequent passes of the loop.




## 3. Example: The Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm to compute prime numbers, named after a Greek who invented it thousands of years ago. The algorithm starts with the first prime number, 2. From there up to a maximum value the algorithm wipes out all multiples of this prime. The next remaining number, 3 must then also be prime. Now all multiples of 3 are erased. This way prime numbers are discovered one by one, and each acts as a sieve to find more primes.
It is fun to program this using tiny sieves as processes that run concurrently. Think of a pipeline with a simple number generator, a list of sieves and a printer. There is a sieve for each recognized prime number; sieve 2 filters out all multiples of 2, etc. After 3 tiny sieves have been generated, the processes would be like:

```
1  main(args:Array[String]) =
2     generator(2,1000000) ==> (..==>sieve)
3                ==>{toPrint}==> printer
4
5  generator(s:Int,e:Int) = for(i<-s to e) <=i
6
7  printer      = ..=>i:Int? println,i
8
9  sieve        =   =>p:Int?    @toPrint:<=p;
10                 ..=>i:Int? if (i%p!=0) <=i
11
12 <==>(i:Int)  = {}
```

- Lines 1 to 3 specify the main processes and their connections, as in the diagram.
  `..==>sieve` is a parallel loop of connected sieves; the ellipsis (`..`) denotes both a loop and an optional exit point. This loop starts with creating one sieve process; as soon as an action starts in therein, a next a new sieve process is created, etc.
- Line 5: The `generator` script generates numbers and sends these over the network: `<=i` sends `i` over an unnamed channel.
- Line 7: `printer` is a loop (`..`) of receiving and printing a number. `=>i:Int?` receives a number `i` from an unnamed channel.
- Lines 9 and 10: a sieve starts by receiving its own prime number `p`, and forwards this to the printer. Then, separated using a semicolon, a loop starts, of receiving numbers and sending these on when not divisible by `p`.
- Line 12 defines the unnamed channel that transmits numbers

Both `generator` and `printer` are quite generic and reusable; in principle they may be moved to a trait.

The pipe in `main` towards the printer, and script `sieve` contain an annotation `toPrint`. This item would be an instance of class `NetworkConnection`. This ensures that the sending of a prime by `<= p` will be redirected towards the printer. Without this provision, the prime would be forwarded to the brand new next sieve.

## 4. The Algebra of Communicating Processes

A good understanding of SubScript requires some knowledge of the Algebra of Communicating Processes (ACP)[1]. This 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[1]. More so than the other seminal process calculi (CCS [7] and CSP [5]), the development of ACP focused on the algebra of processes, and sought to create an abstract, generalized axiomatic system for processes and in fact the term process algebra was coined during the research that led to ACP.

ACP uses instantaneous, atomic actions (a,b,c,...) as its main primitives. Two special primitives are the deadlock process $\delta$ and the empty process $\epsilon$. The primitives may be combined to form processes using a variety of operators. These operators can be roughly categorized as providing a basic process algebra, concurrency, and communication:

- Choice and sequencing - the most fundamental of algebraic operators are the weakly binding alternative operator ( $+$ ), which provides a choice between actions, and the strongly binding 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 a 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 so they are performed on their own, but together, when active in two parallel processes. Then such 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 special primitives $\delta$ and $\epsilon$ behave much like the 0 and 1 that are usually the neutral elements for addition and multiplication:

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

There is no axiom for $x \cdot \delta$. It just means: x and then deadlock.
$x + \epsilon$ means: *optionally* $x$. This is illustrated by using the given axioms as rewrite rules:

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

[1] This description of ACP has largely been copied from its item in Wikipedia

Using the axioms as rewrite rules it may be shown that any closed process expression is equivalent to one of the following forms:

$$\begin{aligned} x + y \\ a \cdot x \\ \delta \\ \epsilon \end{aligned}$$

This makes it easy to define new operators. For instance, an operator `x/y` that does `x` optionally disrupted by `y`, would be defined with

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

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

$$x \parallel y = x \between y + y \between x + x | y$$

- $x \between y$ - "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 other defining axioms for the parallel merge are quite technical, and they are not replicated here.

Many other extensions to ACP have been developed, e.g. for process launching, interruption, 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, but that work has remained largely unknown [3]. Shortly thereafter, Robin Milner et al developed Pi-calculus [8], which also combines the two theories.

## 5. From ACP and Scala to SubScript

An earlier ACP based family of language extensions was Scriptic [11], a sequel of extended Modula-2, C, C++ and Java. SubScript is a follow up, extending Scala[9] while staying closer to ACP. However, there are still many small and big differences between ACP and the SubScript extension. This section describes the most used half of the language constructs; the rest will be described in a forthcoming technical report.

To describe the SubScript language constructs, various more or less vague phrases are used rather loosely below, such as: "activation", "deactivation", "success", "failure", "process", "parent", and "ancestor". These relate to a "Call Graph" model for executing SubScript programs. A call graph is an acyclic directed graph representing the structure of the activated scripts of an executing SubScript program. It grows downwards by activating nodes in accordance to the static parse trees of called scripts. Process operators will be represented by parent nodes; atomic actions by leave nodes. Section 7 discusses the call graph semantics in more detail.

### 5.1 Lexical differences

ACP specifications apply quite some mathematical symbols. For a programming language, it is in principle desirable that all characters are easy accessible on the keyboard. Therefore SubScript has its own symbols for neutral elements, rather than $\delta$, $\epsilon$.

The ACP symbols for choice, sequence and parallelism are $+$, $\cdot$ and $\|$, for which SubScript has `+`, `;` and `&`. As a courtesy to the ACP scientists, the multiplicative dot $\cdot$ is available for multiplication too in SubScript. Just like in Math and ACP the operator symbol for multiplication (here denoting sequence) may be left out, so that effectively whitespace may act as a sequential operator.

SubScript has a wider range of process operator symbols; their operator precedences follow Scala rules, except for the strongly binding $\cdot$. The spacing without operator binds stronger than $\cdot$.

### 5.2 Script definitions

SubScript adds to Scala a new kind of class members, next to variables and methods: so called "scripts". These are much like methods, but they are refinements of ACP process expressions, rather than statement sequences. A script definition starts with the word `script` rather than `def`. The following script is defined as a sequence of two actions: Scala code fragments that print "Hello" and "World!":

```
script hello = {println("Hello")}
               {println("World!")}
```

Like methods, scripts may be implicit, abstract, or overriding. The script parameters may specify input values, like method parameters. However, script parameter may also specify output values, see the subsection describing script calls.

### 5.3 Executing scripts from Scala

To call a script from Scala, it is useful to have a bridge method that creates a so called Script executor; then calls the script with this executor as a parameter; and finally returns the executor. The caller of the bridge method can query the returned result about the state in which the script had ended: success or failure. Failure occurs when in ACP terms the script ends essentially as the deadlock process $\delta$:

```
def hello: ScriptExecutor = { // bridge
  ScriptExecutor se=new CommonScriptExecutor
  hello(se); // call the script
  se // return the executor
}
def test =  {println(
    if (hello.succeeded) "OK" else "NOK"
)}
```

Such a bridge will also be generated using an annotation:

```
@CommonScriptExecutor
script hello = {println("Hello")}
```

A "main(args: Array[String])" script in a declared object has such an implicit annotation. This creates a bridge method that effectively replaces the "main" method of the object. For instance, the following SubScript program would print "Hello":

```
object Hello {
  script
    main(args: Array[String]) = {println("Hello")}
}
```

Other executors than the CommonScriptExecutor may offer support for specific application requirements, such as real time, simulation time, randomization, and parallel processing.

### 5.4 Executing Scala from scripts

Each execution of Scala code in scripts goes though a so called Code Executor. Using annotations one may assign special code executors, e.g. to enforce execution in the GUI thread or on a special processor. A code executor may cooperate with the script executor to simulate probabilistic behavior, or a notion of time.

Some kinds of Scala code are executed when the management of the call graph requires so: if-conditions and while-conditions, parameter lists that must be evaluated, and annotations. While such execution is ongoing, no other graph management is done. Even so, the code executor may have the code executed in the GUI thread, using a call to `SwingUtilities.invokeAndWait`.

Other kinds of Scala code are in code fragments. These correspond to ACP atomic actions, though not one-to-one. An ACP atomic action is instantaneous, i.e. nothing else happens concurrently. Variations of ACP model longer lasting actions as 2 atomic actions: the action start and end; in between something happens as well, but that is abstracted from. Similarly for SubScript code fragments, one may interpret the opening brace as the atomic start action, and the closing brace as the end action. What happens in between is not really of interest to the script executor; it is up to the code executor, and that may also determine when the action start and end occur.

For most plain code fragments, action start, the "real" code execution and action end are normally done in 1 go, without the script executor doing anything meanwhile.

The code may also be executed in a different thread, e.g. a new one, or in a database access thread, or in the GUI thread using a call to `SwingUtilities.invokeLater`. Then after the execution starts the script executor proceeds meanwhile with its other work. After the code fragment has ended, the script executor will notice that and interpret that as an end action. A code executor that supports time based simulations may similarly postpone the end notification after a given amount of simulation time has elapsed.

Finally there are event-driven code fragments, often listeners to input events. These fragments are executed as call backs, asynchronously from the script manager; afterwards the script manager normally notices such executions, and only then the start and end action are supposed to happen. However, there is obligation for the script manager to do so: the associated atomic actions may have been excluded meanwhile due to other developments in the call graph. Also the script executor may just not be able to keep with all the call back executions.

### 5.5 Here

Scala code in Scripts may refer to a special value named `here`, which refers to the "current operand" of the active process expression. The value `here` is much like the value`this`. Through the value "here" various useful run-time features are accessible, such as a state. `here` may also be used as an implicit value.

### 5.6 Annotations

Annotations in SubScript are a bit different from the ones in Scala. They start with `@`, but then they contain some code instead of a class name, and they are terminated by a colon, as in

```
@code: term.
```

Any syntax conflicts with possible colons in the Scala code of the annotation should be avoided by enclosing the code in braces.

The annotation code executes when its operand is about to become activated. There is often a need to refer to that operand in the code. That is done using the the field `here.there`. That has also an shorthand value: `there`, which is also implicit, instead of `here`.

### 5.7 Flavors of code fragments

| | |
|---|---|
| { ... } | plain code fragment |
| {* ... *} | threaded code fragment: runs in its own thread. The code executor supports interruption from the script executor; the boolean method `here.interrupted` flags such cases. |
| {? ... ?} | unsure code fragment. Action start and end only happen after execution, but that may be prevented by calling `here.fail` or `here.ignore`; after `here.ignore` the code remains eligible for another execution. |
| {! ... !} | immediate code fragment. Executed immediately upon activation, so no action start and end are done. In case `here.fail` has been called, the fragment gets the ACP meaning of $\delta$, else $\epsilon$ |
| {. ... .} | event handling code fragment; meant to be executed by an installed event handler, e.g. for handling keyboard or mouse input. After execution as a call back, the script manager will normally interpret that as an action start and end. `here.fail` or `here.ignore` are available, for instance done when the event did not fit a given condition. |
| {... ... ...} | looping event handling code fragment: after executing the code, the fragment normally remains active to listen to events; this may prevent event loss. The code may also trigger an optional break from the loop or a mandatory break, as by calling `here.optionalBreak` and `here.break`. |

### 5.8 N-Ary Operators

SubScript offers the main process operators of ACP, but with some syntactic and semantic differences. There are also many other operators in SubScript; some of these are very useful and intuitive; the presence of more esoteric ones in the language specification may be less justified. Fortunately these operators are all quite similar so they do not impose a heavy syntax burden.

Still, a better option could be to have these as user defined operators, more or less standardized in a library. But in that case, the language definition should ensure that script executors can handle such operators. In order to learn such requirements, the esoteric operators remain for the time being in the language definition.

#### 5.8.1 Commutativity

Operators such as + that are commutative in ACP, are not exactly commutative in SubScript: when an expression `x+y` is activated, then first the `x` part is activated and then the `y` part. The programmer should know this activation order; it implies that evaluations (e.g. for actual parameters) in `x` precede the ones in `y`.

In a sense commutativity still holds: when writing an expression with the + operator, the programmer normally expresses the intent that there is a choice of the atomic actions sequences between the operands, and he does not care which operand will get priority. It is up to the script executor to determine this prioritization; the executor may even randomize.

#### 5.8.2 Associativity

The ACP operators for sequence, choice and parallelism are binary: they have 2 operands. They are also associative, so that we may often leave out parentheses, and the operators may be considered to be n-ary, i.e. having 2 or more operands.

In SubScript such operators are not associative any more, and therefore they are defined as n-ary operators, not as binary ones. This non-associativity is caused by the existence of some special types of operands, that turn expressions into iterations, or that break away from an expression.

#### 5.8.3 Sequences

Sequences are very similar to their counterparts in usual programming languages. In Scala a new line may be used as an alternative for a semicolon that separate sequence elements. This inspired something similar in SubScript: the operator symbol may be left out completely, not just at the end of the line.

With these multiple ways of expressing sequences code may occasionally become smaller, with less parentheses. We use the fact that the semicolon binds weakly, whereas the "space operator" binds strongly.

#### 5.8.4 Choices

The `+` operator in SubScript is meant for choices between input actions, like in ACP. When applied to other actions, the choice is up to the script executor. Default behavior will be to execute the first activated action. An executor may also randomize by itself, or take assigned probabilities and priorities into account.

#### 5.8.5 Parallelism

The ACP parallelism operator, `parallel`, is a simple kind of "and-parallelism". It succeeds when each of its operands succeeds. Other forms of parallelism would occasionally be useful as well, such as simple or-parallelism. SubScript supports both flavors of parallelism, with symbols `&` and `|`. Analogous to operators on boolean expressions in C and other languages, SubScript has also stronger versions for and- and or- parallelism:

| | |
|---|---|
| `&` | "and parallelism" or "normal parallelism": succeeds when each operand succeeds |
| `\|` | "or parallelism": succeeds as soon as any operand does so |
| `&&` | "strong and": the whole ends as deadlock ($\delta$) as soon as one operand does so |
| `\|\|` | "strong or": the whole ends successfully as soon as any operand does so |

#### 5.8.6 Networks

Another special n-ary operator is `<<==>>`. This is much like "normal" parallelism (using `&`), but it also describes a network that restricts certain kinds of communication actions. These communication actions are either send or receive actions, and their names should end in `<=` (for send) or `=>` (for receive). The `<<==>>` defines a topology that interconnects every subset of operands (not just every pair, since communication is n=-ary in general).

Variations of the network operator symbol may impose restrictions on the topology. The following variations are possible:

```
<==   <<==   ==>   ==>>   <==>   <<==>   <==>>   <<==>>
```

If the arrow is only one-sided, then communication can only go in that direction. A single arrow head (< or >) instead of a double (<< or >>) denotes that communication can only be to the adjacent operand in the corresponding direction. E.g., in

```
p1<==p2<==>>p3<==>p4==>p5
```

process p2 may send to p1 and p3 and p4, etc.
`==>` corresponds with pipes in Unix shell languages.

Inside the network arrows, special annotations may be placed between braces, that further control the topology. For instance,

```
==>{myPipe}==>
```

would give this part of the network annotation `myPipe`. A similar annotation for a send action in the left operand of the arrow could effectively restrict that action to this pipe. SubScript does not enforce such behavior; it should be defined in the class of `myPipe`.

#### 5.8.7 Disrupt and Interrupt Operators

ACP has an extension with "modal transfer" operators for disruption and interruption. SubScript has similar operators:

| | |
|---|---|
| `x/y` | `x` happens, possibly disrupted by `y` |
| `x%/y` | `x` happens with 1 interruption by `y` |
| `x%/%/y` | `x` sequentially interrupted zero or more times by `y` |

The interrupt definition is different from the official one in ACP. The latter denotes optional interruption; it cannot model mandatory interruption, which is an unnecessary limitation. In SubScript the interruption is mandatory; it becomes effectively optional when the right hand side is made optional.

### 5.9 Neutral elements

Most of the introduced n-ary operators have a neutral element, depending on whether they are "Or-like" or "And-like":

| | |
|---|---|
| Or-like | `+  \|  \|\|  /` |
| And-like | `;  &  &&  %/` |
| Neither | `%/%/` |

We may add a neutral element $\nu$ to ACP. If this operand belongs to an or-like operator, then $\nu$ behaves like $\delta$. For and-like operators and in "unclear" circumstances, the neutral process behaves like $\epsilon$. The neutral process is implicit in the definition of an if-expression without an else-part, and also for operands such as iterators.

Since working with Greek symbols is sometimes impractical, SubScript defines `(+-)` for $\nu$, `(-)` for $\delta$ and `(+)` for $\epsilon$.

### 5.10 Process Launching

`*x` means process launching: after activation, `x` is activated in parallel with an ancestor process p, as if p had become p&x. The parent of p the anchor process of x. By default, the anchor is the root process. Additionally the launching process `*x` behaves like $\nu$.

### 5.11 If-else

Just like Scala, SubScript offers if-else constructs; the difference being that operands such as the then part and else part, are script expressions rather than pieces of regular Scala code. An if without an else behaves like $\nu$ when the condition fails, so the following are equivalent:

```
if (b) x
if (b) x else (+-)
```

### 5.12 Iterating and Breaking Operands

ACP applies mathematical symbols $\Sigma$ and $\Pi$ to create fixed size alternative and sequential iterations. SubScript supports iterations with operands; these may be used in combination with any kind of n-ary operator[2]. There are also operands for breaking out of n-ary operator constructs:

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

An activated iterator operand (`while`, `for`, `...` and `..`) acts on its n-ary operator as if the operand list in its specification text is repeated an infinite number of times.

When `break` is activated the related n-ary operator starts acting as if its specification text has no more operands. This is another reason why commutativity is strictly broken for `+` and parallel operators. `while(b)` behaves much like `if(b)... else break`. For the `for` operand something similar holds.

`..` behaves much like a combination of `.` and `....`

The effect of an optional break depends on its related n-ary operator. Most operators will on activation activate their operands from left to right until an optional break (or normal break) is encountered. Activation will continue in a next batch just after an atomic action has started that had been activated in a previous batch.

A sequential operator that is activated, activates only its leftmost operand. When an operand has success, the next operand of the sequence is activated. An optional break that is activated in a sequential operator has such a success, but it also makes the sequential operator itself succeed. For example,

`.x; y` behaves much like `x+(+); y`.
`x . y` behaves much like `x; (+)+y`.

Apart from these effects, all iterating and breaking operands behave like the neutral process $\nu$.

### 5.13 Local Variables and Values

Local variables and values are written down as in Scala using the keywords var and val. A variable should be an operand of a sequential operator, a value may be an operand of any kind of n-ary operator. Both variables and values can be used only in subsequent operands.

[2] At the time of writing it was unclear whether "for" may be implemented

### 5.14 Script calls

Script calls are operands that may look like method calls, but they have extra support for output parameters and matching constraints. Output parameters are neither present in ACP refinements, nor in Scala methods.

#### 5.14.1 Output Parameters

A script definition with an output parameter is:

```
r(i: Int?) = {i=computed}
```

The question mark suffix denotes that parameter i is an output parameter. Then the following call would be allowed:

```
var i: Int r(i?) a(i)
```

#### 5.14.2 Constrained Parameters

The previous definition of the r script allows it to yield any number of type Integer. We may want to restrict the received values to the range 0..9, as in the ACP example. This would be possible if the parameter i in the script definition gets a double question mark suffix:

```
script r(i: Int??) = {i=computed}
```

This makes the parameter `i` a constrained output parameter. The caller of such a script may specify a normal output parameter, but it may also add some constraints. This is done Scala style using the keyword `if`, as in

```
var i: Int r(i? if(i>=0&&i<=9)) a(i)
```

Such a single-parameter constraint condition is evaluated with the formal value of the corresponding parameter of the called script. Formal parameter values are copied onto the actual output parameters only when a script call succeeds.

Normally the definition of script `r` should ensure that its atomic actions may only happen if the constraints evaluate to true. This is possible for instance using:

```
script r(i: Integer??)
  = {? i=computed; if (!(_i.matches) here.fail ?}
```

So a parameter `i` may be referred to by `_i`; this returns an object that has a method named `matches`. Other available features are `value`, `originalValue`, and `kind` (which returns the kind of the corresponding actual parameter).

A convenience method doing the same check for all parameters in one go, is

```
script r(i: Int??)
  = {? i=computed; here.matchParameters ?}
```

#### 5.14.3 Forcing Parameters

A special kind of constraint is calling the script with a value parameter without a question mark suffix. Such a parameter is called a forcing parameter:

```
r(1)
```

#### 5.14.4 Adapting Parameters

Formal constrained parameters may be transparently passed through script calls, as ”adapting parameters”, the following way:

```
script rr(i: Int??) = r(i??)
```

Optionally a postfix test may be added, as in:

```
script rr(i: Int??) = r(i?? if(i%2==0))
```

In both calls `i` is said to be an ”adapting parameter”. Inside script `r`, accessing `_i` has the same effects as inside `rr`

### 5.15 Communication: multi-calls

In ACP atoms $a$, $b$, $c$ denote normally atomic actions, but they may alternatively be partners of pairs of communicating actions. For instance, it may be defined that atoms $a$, $b$ and $c$ communicate in the possible pairs $(a, b)$ and $(a, c)$, yielding some atomic actions $d$ and $e$. At the top level of an ACP program, single occurrences of $a$, $b$ and $c$ are hidden so that these can not be mistaken as autonomous atomic actions.

In SubScript, no such hiding is needed, as it has special kinds of communicating scripts that will not act on their own. Combinations of such scripts refine into processes. This is more general than the refinement into atomic actions, which is prescribed by ACP.
Communication is a generalization of a script call: a *multi-call*. Alternatively: a normal script call is a special case of communication, involving only one party. For instance,

```
script a,b = {println("hello")} {println("world")}
```

When `a` and `b` have been activated in parallel to one another, their shared body may start. In case only `a` is active, no action would follow; an active `a` would just have to wait for a partner `b`; maybe it will be deactivated before that would happen.

When script `a` should also be able to communicate with a partner `c`, then these alternatives must be marked, by writing `+=` instead of `=` in the definition. For the same communicating script `a` , no parter may even be needed:

```
script a,b += {println("hello")}
script a,c += {println("world")}
script a   += {println("Alone")}
```

This is a bit similar to marking overridden methods in Scala with the keyword ”override”.

#### 5.15.1 Multiple Communication Partners

Unlike in standard ACP, SubScript communication may involve more than 2 partners:

```
script a,b,c = {println("hello")}
```

Any number of partners with a given name and signature may be allowed to communicate, and a partner may be optional:

```
script a..  = {println("hello")}
script b,.c = {println("world")}
```

So one or more `a`'s together yield ”hello”. One `b` with or without a `c` yields ”world”. Unless specified otherwise, a script executor attempts to establish communications with a maximum number of partners: a set of partners is allowed to communicate if it cannot be extended any more with other activated partner scripts.

#### 5.15.2 Communication Parameters

Communicating scripts may share parameters. Only the first occurrence of a shared parameter in the formal parameter lists specifies its type. For instance, a communication with a send action `s` and a receive action `r` would be like:

```
script s(i:Int),r(i:_??) = {print(i)}

script test1 = s(1) & j:Int r(j?)
script test2 = s(1) &       r(1)
script test3 = s(1) &       r(2)
```

`test1` and `test2` would result in a communication; `test3` would not, since the forcing parameter value handed to `r` does not match the input parameter value handed to `s`.

### 5.15.3 Communication over Channels

There is a more convenient notation for common send and receive pairs, so that parameter lists need not be copied. First, a script name may end with an arrow symbol. There need not be a text part before the arrow. Such scripts denote send or receive actions over a channel. Second, there is a short hand notation for such send/receive pairs with equal channel names and almost equal parameter lists. The send actions should have input parameters, and the receive actions should have either output or constrained parameters. So the following two definitions are equivalent:

```
a<-(i:Int), a->(i?)            a<-->(i:Int?)
b<-(i:Int),.b->(i??)          b<-.->(i:Int??)
 <-(i:Int),..->(i??)          <-..->(i:Int??)
```

Successful send and receive actions could be:

```
script test1 = a<-(1) & var j:Int a->(j?)
script test2 = b<-(1) & b->(1)
script test3 =  <-(1) & var j:Int ->(j?) & ->(2)
```

### 5.15.4 Communication over networks

Communication may be restricted to a network topology defined by the `<==>` operator. Then specify thick arrows, as in

```
a<==>(i:Int?)
```

This network should also be defined in scripts belonging to the same object or class instance as the send and receive actions.

### 5.15.5 Asynchronous Communication

To do a send action asynchronously over a channel, just launch it as a process using the unary prefix operator *:

```
*a<-(1)
```

An equivalent notation for a channel is:

```
a<-*(1)
```

For an asynchronous send over a network channel, the first form `*a<=(1)` may lead to unintended results. The reason is that normally processes are launched to a too high level, directly under the top of the call hierarchy. If a launched network send action would not be subordinate to the aimed network operator, it cannot fulfill the networking constraint. By using the form `a<=*(1)`, the send action would be launched as a subordinate of the nearest network operator ancestor.

## 5.16 Syntactic sugar

With some syntactic sugar SubScript programs may become even more concise, and lots of braces and parentheses may be ditched.

### 5.16.1 script.. Sections

Often classes will contain sequences of multiple scripts. The repeated word `script` can be factored out like this:

```
script..
  a = {println("hello")}
  b = {println("world")}
```

Some rules to make this work:

- The indent level for each of the defined scripts should be larger than the one of the leading phrase `script..`
- The indent level for the script body should be larger than the indent level of the script name
- Script definitions in such a section should start at the same indent level

Similar sections could be allowed for regular Scala constructs: `class..`, `val..`, `var..` and `def...` From here on in this paper, we will also leave the phrase `script..` when it suits.

### 5.16.2 Omitting Braces

Braces may be omitted for normal code fragments that merely consist of a method call with a simple instance access path (empty or only identifiers and `this`). So the following script is valid:

```
a = println("hello")
```

### 5.16.3 Omitting Parameter List Parentheses

We may leave out the parentheses of a parameter list if each parameter is either a literal or a simple access path. A comma needs then to be added as a separator after the name of the script or method:

```
a = println,"hello"
```

### 5.16.4 Omitting Names of Implicit Scripts

We may leave out the names of implicit scripts, so that calls will be resolved based on actual parameter lists:

```
implicit _println(s: String*) = {println(s)}

a = "hello"  "world" // 2 calls
b = "hello", "world" // 1 call
```

### 5.16.5 Local Value Declarations in Calls

There is a shorthand notation for a sequence of a value declaration and a call that initializes that value in the position of an output parameter:

```
val i:Int r(i?)            r(i:Int?)
val i:Int r,i?             r,i:Int?
val n:Int n?               n:Int?
val n:Int n?if(n<10)       n:Int?if(n<10)
val n,m:Int n?,m?          n:Int?,m:Int?
```

### 5.16.6 Prefix Notation

To avoid annoying and error-prone repetitions of n-ary operators, a prefix notation is allowed. The following specifications are equivalent:

```
unaryOperator =  "!" + "-" + "~" + "*"

unaryOperator =+ "!"   "-"   "~"   "*"
```

This latter line is part of the SubScript syntax definition, that is written in SubScript itself. A common pattern in syntax definitions is a choice between sequences. This is done in SubScript as:

```
simpleTerm =;+ simpleValueLedTerm
               specialTerm
               throwTerm
               whileTerm
               forTerm
               codeFragment
               "(" scriptExpression ")"
               arrow . actualParameters
```

The choice alternatives appears on separate lines, while each line denotes a sequence. For that purpose, the start of the definition starts with `simpleTerm =;+`. The first symbol after `=;+` defines the meaning of white space within a line (here a sequence), and the second defines the meaning of white space between lines (here a choice).

## 6. Use case: text parsing

SubScript's predecessor Scriptic is in operational use since 2010 to analyze technical documentation on a communication protocol. The application parses documents containing chapter headers, free text, tables and footnotes. The elements had to be captured and stored in a more structured fashion.
This real world experience also indicates how SubScript may be used: grammar rules combined with more conventional constructs such as local variables, parameters, `if` and `for`. The code fragments below are given in the Subscript style.

### 6.1 Low level scripts

The input is a set of Open Document Format (ODF) documents, available as an XML stream. Eight low level scripts recognize table structures, line end and file end:

```
startTable  startRow startCell   endOfLine
  endTable    endRow   endCell   endOfFile
```

Two implicit scripts recognize text and numbers on a line:

```
implicit text(s: String??) =
  @expect(here,  TextToken(_s): {? accept(here) ?}
implicit number(n:  Int??) =
  @expect(here,NumberToken(_n): {? accept(here) ?}
```

The `expect` calls in the annotations communicate expectations to the scanner. The latter uses the expectations to decide on what token to recognize from the input stream. It applies a higher priority for a NumberToken than for a TextToken. For a specific expected number or text (i.e. in case of a forcing parameter to `text` or `number`), the priority is higher than for any number or text. The scanner checks at run time for ambiguities in the expectations. In case of unmatched input it lists all expectations.

```
comma          = ","
anyText        = s: String?
someEmptyLines = ..endOfLine
someLines      = ..anyText endOfLine
```

These scripts parse a table cell with a text or number:

```
cell(s: String?) = startCell s? endOfLine endCell
cell(n: Int   ?) = startCell n? endOfLine endCell
```

### 6.2 Using if-expressions and for-operands

The ACP based constructs mix well with if-expressions and for-operands. E.g., to parse a header row with predefined texts:

```
tableRow(ss: String*) =
   startRow; for(s<-ss) cell(s); endRow
```

To parse a string out of a set an alternative for-loop is used:

```
text_oneOf(r: String?, ss: String*) =
      for(s<-ss) + s {! r=s !}
```

The parsed documents refer to footnotes as "(1)" etc. Footnotes often start like "(1) -", but in some chapters like "1.":

```
footnoteRef(n: Int?) = "(" n? ")"

footnote(n: Int?, s: String?) =
  if (fnFormat==NUMBER_DOT)        (n? ".")
  else                 (footnoteRef,n? "-")
  s? endOfLine
```

#### 6.2.1 Challenge: disambiguation

Consider this choice between a text cell and a number cell:

```
cell,s? + cell,n?
```

This would be ambiguous because it comes down to

```
   startCell s? endCell
+ startCell n? endCell
```

The ambiguity may be solved using or-parallelism:

```
cell,s? || cell,n?
```

This happens to work: if the call to `text` in the left hand side succeeds, the call to `number` in the right hand side fails, and vice versa. That is due to the inner working of those scripts, that call the method `accept`. It would be preferable if the two calls to `startCell` were treated as one, and when `text` and `number` were treated as exclusive alternatives.
This challenge seems to be solvable using a choice operator, say `|+|`, that is non-exclusive for actions that communicate with one another. Assume `startCell` calls a script `xmlTag`, then the latter would allow for multicalls:

```
xmlTag(t: XMLTag),.. = @expect(here, t)
                       {? accept(here) ?}
```

A dedicated script executor could understand from an annotation on a process expression that it should disambiguate choices there by giving `+` the effect of `|+|`. This would also apply to implicit choices in optional break from a sequence:

```
@disambiguate: (  cell,s? + cell,n?)
@disambiguate: (. cell,s? ; cell,n?)
```

### 6.3 Challenge: avoid cluttering with data

The following script parses and collects comma separated numbers:

```
csn(nn:Queue[Int]) = i:Int?{!nn+=i!}..comma
```

This code is a bit cluttered. YACC [6] does a much better job in this respect; it lets refinements return values, which are accessible using variables named like `$1`, `$2`. It is still a challenge to enable such an approach for SubScript; to deal with iterations, then phrases like `$$1` would mean sequences of return values. The definition of the comma separated numbers would become something like

```
csn =  number..comma; {! $ = $$1 !}
```

### 6.4 Similarity: GUI controllers

SubScript GUI controllers are quite similar to text parsers:

- Using process expressions instead of imperative constructs reduces the need for state variables.
- Activated input scripts register acceptable input options. GUI controllers may use this information to enable and disable buttons as needed. Text parsers can give error messages stating expected tokens when actual input is not acceptable; they can also flag ambiguous choices.

This agrees with a quote of Stephen Johnson [4], creator of YACC: *One application I think is promising is using compiler-design techniques to design GUIs - I think GUI designers are still writing GUIs in the equivalent of assembly language, and interfaces have become too complicated for that to work any more.*

### 6.5 Performance

The Scriptic-Java application expects about 120,000 tokens per second, of which about a quarter is accepted: 30,000. This has been recorded on Fedora Linux with Java 1.6, on Intel Quad Core 2.66 GHz. SubScript performance will likely be similar. When used for GUI controllers, the overhead of expecting and accepting input actions will be neglectable.

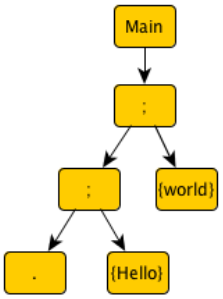
## 7. Call Graph Semantics

SubScript programs are executed by script executors. The semantics therefore depends on the applied executor. Normally the executor is an instance of class CommonScriptExecutor. Its specifications should preferably be a reference for other executors. However, no exact specification is available at the time of writing this article. Yet we can present informally how it should operate.

The static structure of scripts may be represented by trees, also called "Template trees". For instance, consider the following script prints optionally "Hello", and then "world!":

```
main(args: Array[String]) = ."Hello "; "world!"
```

The corresponding template tree looks like:




A script being called from Scala implies that its template tree is handed to a script executor. The latter starts building a so called call graph, an acyclic graph with a single root node. Below this root node there is a "script call node", with in turn parents a "callee node" for the executed script. Under that, other nodes will be added and removed according to the template tree as the program evolves; these nodes represent the process expression constructs, such as n-ary operators and code fragments. This is done by handling simple messages of various kinds.

Call graph management (node activation to deactivation) precedes over executing code for atomic actions. Graph operations below a unary or n-ary operator precede over the operation at such an operator. This is achieved by collecting messages arriving at such operators in so called `Continuation` messages. This way the response by at the n-ary operator can take into account all messages that have arrived.

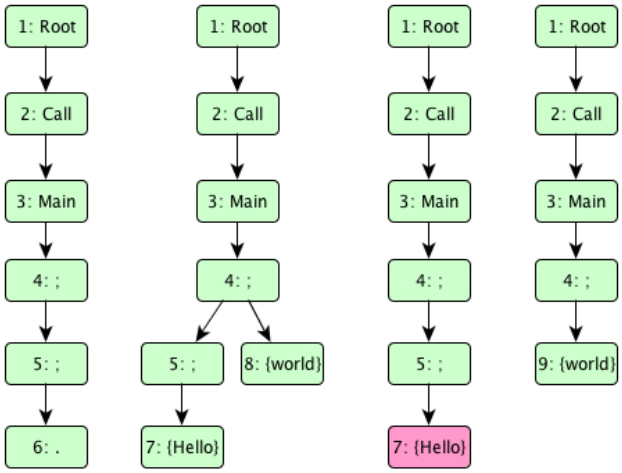
The message types are in descending priority order:

- `AAActivated, AAStarted, AAEnded` - an atomic action has been activated, started or ended
- `CAActivated` - an communicator action has been activated
- `Break` - a break or optional break has been encountered
- `Success` - a success has been encountered
- `Exclude, Suspend, Resume` - atomic actions in descendants must be excluded, suspended or resumed
- `Activation` - a node is added to the call graph, according to the template tree. This may also involve executing Scala code for annotations, script parameter evaluation, if- and while conditions, etc. Moreover, one or more messages may be inserted in the message queue.
- `Deactivation` - a node is removed from the graph
- `Continuation1` - Collected messages for an unary operator
- `Continuation` - Collected messages for an n-ary operator
- `CommunicationMatching` - Communications be determined from activated communicator actions
- `AAExecutionFinished` - a code executor reports that the code related to an atomic action has finished
- `AAToBeExecuted` - Atomic Action to be executed in the main thread
- `AAToBeReexecuted` - Atomic Action for which another `AAToBeExecuted` message should be inserted

`Exclude`, `Resume` and `Suspend` messages are propagated downwards in the call graph. `AAStarted`, `AAEnded`, `CAActivated`, `AAActivated`, `Break` and `Success` messages are propagated upwards in the graph; the latter two stop at n-ary operator nodes. They may cause something to happen; e.g., when an `AAStarted` message arrives a child node at `+` and `;`, `Exclude` messages for the siblings are inserted.

Many language constructs behave quite straightforward. E.g., upon activation, the deadlock process `(-)` inserts a `Deactivation` message. The empty process `(+)` inserts a `Success` and a `Deactivation` message. The neutral process determines whether its nary operator ancestor is and-like: in that case it acts as `(+)`, else as `(-)`. When an iterator is activated (`.. ... while for`) it first sets an iteration flag at its n-ary ancestor node, if any. Then it may execute some test (`while for`), insert a `Break` message; from then the iterator acts neutrally, i.e. like `(+-)`.

Executing the previous Hello Wold script would yield the following call graph development:




Nodes 1 is the root; node 2 is an anchor point for the executed script, node 3. This activates the outer sequential operator as node 4. Activating any n-ary operator activates its leftmost operand, and then inserts a `Continuation` message. Thus nodes 5 and 6 are activated.

The latter inserts `Break`, `Success` and `Deactivate` messages. These three arrive at sequential node 5, where they are added to the `Continuation`; the `Deactivation` also removes node 6. The `Continuation` at node 5 activates the next operand (node 7, Hello), because of the `Success`. It also sends a `Success` onward to node 4, because of the optional `Break`. There it is added to the `Continuation`. Handling the `Succes`-holding `Continuation` at node 4 activates the next operand (node 8, `World`). During the activation of nodes 7 and 8, `AAActivated` and `AAToBeExecuted` messages are inserted.

Now the `AAToBeExecuted` for node 7 is handled; it precedes over the one for node 8 because it had been inserted earlier. The code executor for node 7 executes the code and inserts `AAStarted`, `AAEnded`, `Success` and `Deactivation` messages. The `AAStarted` has no effect at node 5. There another `AAStarted` is inserted propagating to node 4. Handling this one causes an `Exclude` message for node 8 to be inserted. Handling that message inserts a `Deactivation` for node 8.

Then another `Succes`-holding `Continuation` at node 5 is handled; this leads to such a `Continuation` at node 4, which activates the node 9, again for `World`. After that has been executed and processed, all nodes are deactivated and the script execution ends.

An Unsure code fragment would be executed first; if it succeeds, messages are inserted like for a normal code fragment. If it fails, a Deactivation is inserted. If it got an undetermined state then a `AAToBeReexecuted` would be inserted.

For a threaded code fragment, or one for the GUI thread, an `AAStarted` messages is inserted, and thereafter the code fragment is executed asynchronously; at the end thereof an `AAExecution-Finished` message is inserted. This is picked up in the message handling loop, and then, normally, `AAEnded`, `Success` and `Deactivation` messages are inserted.

An event handling code executor works in general asynchronously. Upon an event notification it executes the code, and then inserts an `AAExecutionFinished` message. Handling the latter inserts an `AAStarted` message and the three others.
Note that handling an `AAExecutionFinished` may result in only a `Deactivation` message, when the atomic action had been excluded meanwhile; that may have happened e.g. because of another atomic action in another branch of a disrupt operator.

Dedicated script executors may support concepts such as real time constraints simulation time constraints, probability, parallel execution and disambiguation. These executors may come with special classes for use in script annotations, and user defined operators.

## 8. Implementation

At the time of writing, SubScript has only been implemented as a Scala DSL, that accesses a CommonScriptExecutor; full syntactic support by modifying the Scala compiler is only done after the DSL implementation is finished and stable. This approach has some advantages:

- easy to develop: we can concentrate on semantics rather than on the trivial syntax that would still require fairly complicated compiler modifications
- language features such as variable argument lists and named parameters are inherited for free
- later only relatively simple changes to the Scala compiler will be needed

A "Hello" "World" main script is programmed in the DSL as:

```
def _main(_args:FormalInputParameter[Array[String]])
 = _script('main, _args~'args) {
     _seq({print("Hello ")}, {println("world!")})
   }
def main(args: Array[String]): Unit
  = _execute(_main(new ActualInputParameter(args)))
```

The current implementation is publicly available at Google Code [10]. The size is about 2000 lines (file headers excluded). This may grow to an estimated 3000 lines, as some language features are not yet implemented.

Thread synchronization is scarcely needed in the source code; mainly for accessing the message queue. The GUI Lookup example applications run responsively.

Using Scala as a base language and implementation language was very useful:

- ease of making a DSL that comes quite close to a full SubScript implementation
- concise class definitions, and handy case classes and function types
- a similar style of defining methods and variables with "="
- inspiring handy features such as implicits

## 9. Conclusion

Adding ACP plus some extra features and syntactic sugar to Scala yields support for various programming paradigms, such as: event-drivenness, concurrency, parallelism, non-determinism, dataflow, logic and time awareness. Despite this expressive power, an implementation may be rather simple and small.

SubScript raises some challenges for ACP theorists:

- ACP-like algebra may specify various kinds of structures, not just processes. E.g. in the GUI example program
  `searchCommand = searchButton + Key.Enter`
  a button and a key code are added; this happened to yield an input process, but an output process would have been possible as well.
- new operators such as or-parallelism, sequential continuation after deadlock, disambiguating choice and negation
- n-way communication and communication that refines into process expressions instead of atomic actions

The most useful and typical parts of the language have been discussed. Additional language elements will be: exception handling, match expressions, process lambdas, various unary and n-ary operators, and some provisions for local data and parameters. There is also still much other work to be done: a complete language definition and implementation, tool support and practical experiments. The current work will hopefully raise interest at programming language enthusiasts, so that some of them will join further development.

## References


[1] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335:131–146, May 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2004.07.036.

[2] E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[3] H. Goeman. Towards a theory of (self) applicative communicating processes: A short note. *Inf. Process. Lett.*, 34(3):139–142, 1990.

[4] N. Hamilton. The A-Z of Programming Languages: YACC. *Computerworld*, July 2008.

[5] C. Hoare. Communicating sequential processes. *ACM Computing Surveys*, 7(1):80–112, 1985.

[6] S. Johnson. Yacc: Yet another compiler-compiler. Technical report, Bell Laboratories, 1979.

[7] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

[8] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i. *I AND II. INFORMATION AND COMPUTATION*, 100, 1989.

[9] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. Artima Incorporation, USA, 2nd edition, 2011. ISBN 0981531644, 9780981531649.

[10] SubScript development site, 2012. URL `http://code.google.com/p/subscript/`.

[11] A. van Delft. The scriptic programming language. In *Proceedings on Parallel architectures and languages Europe : volume II: parallel languages*, PARLE '91, pages 220–237, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54152-7.