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. This has been done in Subscript, which is a extension to Scala. Examples show how Subscript is useful for programming GUI controllers in the MVC paradigm.
Currently Subscript is implemented as a DSL, using a run-time library named the Subscript VM. This VM has been written in about 2000 lines of Scala code. It 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 VMs 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 Controller, simulation

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. 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, freeze these days 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, taken from the Algebra of Communicating Processes.

2. The Algebra of Communicating Processes

The Algebra of Communicating Processes (ACP)[2] 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 and CSP), 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 δ and the empty process ϵ. 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 alternative operator (+), which provides a choice between actions, and the sequencing operator ($\cdot$), which

[1] This description of ACP has been based on its item in Wikipedia

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 $(a \cdot b) \| (c \cdot d)$ 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; they can then 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 special primitives δ and ϵ behave much like the 0 and 1 that are neutral elements for addition and multiplication in usual algebra:

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

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

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

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

$$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x|y$$

$x\lfloor\!\lfloor y$ - "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.

A different way to define the operators is by means of a technique called Structural Operational Semantics. However, for describing the semantics of SubScript it seems to be less applicable.

In 1990, Henk Goeman unified Lambda Calculus with process expressions, but that work has remained largely unknown. Shortly thereafter, Robin Milner started Pi-calculus, which also combines the two theories.

Many extensions to ACP have been developed, e.g. interrupt and disrupt operators, 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.

3. From ACP and Scala to SubScript

It is well possible to add ACP-like expressions to imperative programming languages. This has for instance been done in Scriptic, a sequence of extensions to C, C++ and Java. Subscript is a follow up, extending Scala while staying closer to ACP. However, there are still many small and big differences between ACP and the SubScript extension.

To describe the Subscript language constructs, various more or less vague phrases are used rather loosely below, such as: "activation", "happen", "success", "failure", "suspend", "resume", "process", "parent", and "ancestor". These relate to a "Call Graph" model for executing SubScript programs. A call graph is an acyclic directed graph; 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.

Inside this call graph, several kinds of messages are sent along the edges. E.g. when an atomic action starts, this is reported upwards in the call graph. If that report arrives from a certain child node at a node representing a "+" operator, then that node sends "exclude" messages to all its other children, so that any active atomic actions in those branches will be deactivated.

After completion, an atomic action will report a $success$ message upwards; thereafter it deactivates. The same does an ϵ-like node. A sequence node that receives a success message from a child node, will activate a next child, if applicable given the parse tree; otherwise the sequence node will send a success messages to its parent.

Section 7 discusses the call graph semantics in more detail.

3.1 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. On the other hand it would as well be nice if SubScript would have some mathematical look and feel, using symbols like δ, ϵ and ν for deadlock, an empty process and a neutral process. Since these symbols are not easy to type, predefined symbols (-), (+) and (+-) exist for these processes. The variants with the Greek symbols are defined in a 'Predef' object.

ACP symbols for choice, sequence parallelism are +, ; 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. 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 $\cdot$.

3.2 Scripts

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

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

Like methods, scripts may be implicit, abstract, or overriding. The script parameters are a different from method parameters: each script parameter is of kind input, output or constrained, whereas method parameters are of kind input. Output parameters are marked by a single question mark; constrained parameters have two question marks, as in:

```
script read(i: Integer?)
script readForcable(i: Integer??)
```

The first script should be called with an actual output parameter, as in $read(j?)$. The second script may be called that way, but also with a specific ”forcing” value, or with a less forcing constraint, as in

```
readForcable(10)
readForcable(j? if(j<=10))
```

3.3 Native language interoperability

Scripts may contain Scala code in various flavors of code fragments, in actual parameter lists of script calls, and in several other constructs. Such Scala code may refer to a special value named $here$, which refers to the ”current operand” of the active process expression. $here$ is much like $this$, the current object. Through the value ”here” various useful run-time features are accessible, such as a state. $here$ may also be used as an implicit value.

To call a script from Scala, it is useful to have a bridge method that adds a so called Script executer as a parameter, and returns that executer. 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 δ:

```
def hello: ScriptExecuter = {
  ScriptExecuter se=new BasicScriptExecuter
  hello(se); se
}
def test =  {println(
    if (hello.succeeded) "OK" else "NOK"
)}
```

Such a bridge will also be generated using an annotation:

```
@BasicScriptExecuter
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 executers than the BasicScriptExecuter may offer support for specific application requirements, such as real time, simulation time, randomization, and parallel processing.

3.4 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 executers can handle such operators. In order to learn such requirements, the esoteric operators remain for the time being in the language definition.

3.4.1 Arity

The ACP operators for sequence, choice and parallelism are in principle binary, but as they are associative, the operators may also be considered to be n-ary. In SubScript they they 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. For instance, consider the following two specifications:

(..A)B - a sequence of one or more A’s, and finally B
..(AB) - a sequence of one or more sequences of A and B

Here the two dots denote an iteration and at the same time an optional exit point.

3.4.2 Commutativity

Operators such as + that are commutative in ACP are even not 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 the conditionals in if-expressions) 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 executer to determine this prioritization; the executer may even randomize.

3.4.3 Sequences

Having multiple ways of expressing sequences occasionally allows for smaller code, with less parentheses. We use the fact that the semicolon binds weakly, whereas the ”space operator” binds strongly. For instance, a sequence of A’s terminated by a B would initially be

```
(..;A);B
```

or

```
(..A)B
```

The parentheses are not needed if we us just one semicolon:

```
..A;B
```

3.4.4 Parallelism

The ACP parallelism operator, ||, 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 boolean expressions in C and other languages, SubScript has also stronger versions for and- and or-parallelism, and even for ”equal-parallelism”:

&	”and parallelism” or ”normal parallelism”: succeeds when each operand succeeds
|	”or parallelism”: the succeeds as soon as any operand does so
&&	”strong and”: the whole ends as deadlock (δ) as soon as one operand does so
||	”strong or”: the whole ends successfully as soon as any operand does so
==	”equal parallelism”: this succeeds like & when each operand succeeds, but it also succeeds when each operand has ended as deadlock

The latter operator is only experimental, for the time being. It makes the set of parallel operators logically more complete. There is no ”not-equals-parallelism”, since that may be counterintuitive for an n-ary operator. However, there are also experimental unary negation operators which combined with $==$ may accomplish ”not-equals-parallelism”.

3.4.5 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 $myPipe$.

The network arrows may also be marked with value tuples, as in $<<== \ (i,j) \ ==>>$. This can be picked up by a topology controller that is specified using an annotation. E.g.,

```
@myTopology: (
 for (i<-0 to m; j<-0 to n) <<=(i,j)=>> p(i,j)
)
```

$myTopology$ could for instance impose a torus topology by allowing only connections between ”adjacent” i,j pairs.

3.4.6 Left-merge Operators

For each parallel operator there is a left-merge version, with a symbol equal to the original symbol with ”·” appended:

$\& \cdot$ $\&\& \cdot$ $| \cdot$ $||\cdot$ $== \cdot$ $<<==>> \cdot$

The right-hand side only becomes active when an action at the left hand side occurs for the first time. The ”·” postfix expresses that there is sequential dependency.

3.4.7 Disrupt, Interrupt and other Suspend/Resume Operators

ACP has an extension with ”modal transfer” operators for disruption and interruption. Subscript has similar operators:

$/$	disrupt: x/y means that x happens, possibly disrupted by y
$\%/$	interrupt: x%/y means that x happens with 1 interruption 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.

The interrupt operator symbol has two characters, one of which is %. It belongs to an experimental family of suspend/resume operators, each starting with a % character in its symbol. Two of these don’t have a neutral element:

x%&y	x and y in any order. As soon as x starts, y is suspended, and vice versa. As soon as x has success, y is resumed, and vice versa. Similar, but not equal, to $xy + yx$
x%;y	first x and then y, with x and y both optional, but when x does not happen, y must happen. Similar, but not equal, to $x(y + \epsilon) + y$
x%%y	x and y in any order and both optional, but at least one of the two must happen. Similar, but not equal, to $x(y + \epsilon) + y(x + \epsilon)$
x%/y	x interrupted by y; x happens, but y as well; x is suspended as soon as y starts to happen. When y is (or may be) ready, x is (or may be) resumed
x%/%/y	x sequentially interrupted zero or more times by y

3.4.8 Deadlock continuation operator

In normal sequences of the form $x; y$, y may start when x succeeds. A dual kind of sequence is $x!; y$, meaning: y may start when x ends in deadlock. This is useful for text parsers: as soon as deadlock occurs because the input text does not match a specified grammar, then the execution falls through this operator so that an error message may be given.

3.5 Neutral elements

Most of the introduced n-ary operators have a neutral element, which is in ACP terms either δ or ϵ. We will call these operators or-like and and-like respectively.

Or-like	+	|	||	/	%;	%%	!;
And-like	;	&	&&	%/	%&		
Neither	==	%/%/					

A special operand is ν, named the neutral process. If this operand belongs to an or-like operator, then ν behaves like δ. For and-like operators and in ”unclear” circumstances, the neutral process behaves like ϵ.

The neutral process is implicit in the definition of if-expression without an else-part, and also for operands such as iterators.

Since working with Greek symbols is at times problematic, Subscript defines $(-)$, $(+)$ and $(+-)$ for the deadlock, empty process and neutral process. The Greek symbols are defined as scripts in the subscript.Predef object.

3.6 Unary operators

There are some unary operators on expressions:

!x	negation; ends in deadlock when x ends successfully, and vice versa
-x	strong negation; ends in deadlock when x ends successfully, and vice versa. -x also succeeds when an action in x happens without x succeeding
$\sim x$	action tracing; succeeds when an action in x happens
*x	process spawning. After activation, x executes in parallel with its parent process p, as if p had become p&x. This parent process is by default the highest level script that had been called from the base language
**x	marking of a anchor place for spawned processes

Spawned process starts to run in parallel to its nearest by parent process as seen in the call hierarchy

The latter two are not allowed by Scala rules on unary prefix operators. This may turn out to be undesired; maybe another way will be found to express launching and anchor places.

3.7 If-else, Match and Ternary Operator

Just like Scala, SubScript offers if-else and match constructs; the difference being that operands such as the then part and else part, are script expressions rather than pieces of regular Scala code.

```
if (b) x
```

behaves as if the else part is neutral, so it is shorthand for

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

There is also a ternary operator, that has 3 processes as operands $x?y : z$ does x; when that has success, y may start happening. In case x ends in deadlock, z starts. x? y: (-) behaves much like x;y. A difference is that such a sequence cannot become an iteration. x? (+): y behaves much like x!;y. Again it cannot become an iteration. Binary usage is also allowed: x? y is shorthand for x? y: (+-). It is a good means to express a either precondition or a postcondition.

3.8 Iterating and Breaking Operands

There are 6 operand for supporting iterations and breaking:

$while$	marks a loop and an conditional mandatory break point
for	a for-comprehension, like while marking a loop and a break point [2]
...	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

Note that these are operands; they often belong to a sequential operator, but the iterations may as well be alternative or parallel. The relation ”belongs to an n-ary operator” shines through unary operators, script calls, if expressions, etc. An activated iterator operands (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.

$while(b)$ behaves much like $if(b)...else\ break$. For the for operand something similar holds. 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.

.. behaves much like a combination of . and .... The optional break lets its related n-ary operator optionally stop activating its operands. What exactly happens depends on the kind of the sequential operator:

- A sequential operator will succeed and also activate the next operand. E.g. $x.y$ does x; thereafter it may stop or continue with y. It behaves much like $x; (+) + y$, if we disregard the effects of iterating and breaking operands in y.
- Most operators will on activation activate its operands from left to right until a break is encountered. If the break is a hard break, then no more operands are activated any more. If the break is a soft break, then activation will continue after an atomic action has started in one of the just activated operands.

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

3.9 Local Variables and Values

Local variables and values are written down as in Scala using the keywords var and val. A for comprehension may also imply a new local value (not a variable).

- A variable should be a direct operand of a sequential operator
- A value should be a direct operand of any kind of n-ary operator.

Both variables and values can be used only in subsequent operands.

3.9.1 Looping Local Variables and Constants

A local variable or value may be initialized using a ”looping” expression. E.g.,

```
val i=0...(i+1)
```

Like ”...”, such a declaration turns its related n-ary operator into an iteration. The value i becomes 0 during the first iteration pass; in subsequent passes it becomes the value of the previous pass. So the following fragment has two iterators; it will print 0 to 9:

```
val i=0...(i+1) while(i<10) {println(i)}
```

A parallel variation will do the same; each parallel pass will get its own copy of the value i:

```
val i=0...(i+1) & while(i<10) & {println(i)}
```

3.9.2 Private Local Variables

Sometimes different operands of an n-ary operator need their own copy of a variable. Then a ”private” declaration would be useful. For instance consider a variation of the previous example:

```
var i=0; while(i<10) & {println(i)} & {! i+=1 !}
```

This will print 10 times the number 10: the execution of all println actions take place after all have been activated, and after each activation the variable i is incremented. A ”private” declaration will then ensure that each println action gets a private copy of i.

```
var i=0;
while(i<10) & {!i+=1!} & private i: {println(i)}
```

This prints numbers 1 up to 10.

3.10 Code fragments

The atomic actions of ACP have their SubScript counterparts in code fragments. These are operands with pieces of Scala code, enclosed in braces. However, it is often more accurate to say that the start and end of the code fragment execution are like ACP atomic actions. This allows code fragments to model longer lasting actions, such as code running in a separate thread, or code that simulates to take a nonzero duration.

A special kind of code fragment does not behave like an atomic action, but as δ or ϵ.

The Script Executer may determine the way code fragments are executed. Examples of such executor types are:

- A discrete event simulation engine
- A probabilistic engine doing Monte Carlo execution
- A scheduler for parallel hardware

Code fragments are always enclosed in braces. Symbols next to the braces denote different flavors:

{ code }	plain code fragment. Normally, no other actions take place between the start and end of this fragment. However, a simulation engine may attribute a positive duration to this fragment, so that other actions may come in between. The same happens when an executor defers the code asynchronously to the GUI thread
{* code *}	threaded code fragment. This code fragment normally runs in its own thread, but the script executer may assign it to a thread pool
{? code ?}	unsure code fragment. When executed, the code fragment may not reflect a happening atomic action, but instead δ or ϵ. It may even get state undetermined, meaning that it remains eligible for another execution
{! code !}	immediate code fragment. Executed immediately upon activation. This normally gets the ACP meaning of ϵ, but that may be overridden by the code to become δ or ν
{. code .}	event handling code fragment; meant to be executed by an installed event handler, e.g. for handling keyboard or mouse input. Normally it becomes an atomic action shortly after the code execution, but the code may set it to behave like ? or undetermined
{... code ...}	looping event handling code fragment. The code may also trigger an optional break from the loop or a mandatory break, as by calling $here.optionalBreak$ and $here.break$

3.11 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.

3.11.1 Output Parameters

Refinements in ACP may have value parameters. This leads to specifications with mathematical Sigma symbols, standing for parameterized addition. For instance, suppose a number i between 0 and 9 is read from a channel, depicted by r(i); then some action a(i) is performed. In ACP this would typically be written down like

$$ra = \sum_{i=0}^{9} r(i) \cdot a(i)$$

Programmers would be much more familiar with a solution that would not require a sigma. SubScript therefore offers output parameters. For instance, a script definition could start with

```
  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)
```

3.12 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. Only when script call succeeds are formal parameter values copied onto the actual output parameters.

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 ?}
```

3.13 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)
```

3.14 Adapting Parameters

Formal constrained parameters may be transparently passed through script calls, by having the parameter list enclosed in parentheses:

```
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

3.15 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 $@code : term$.

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 implicit shorthand value: $there$, which is also implicit, instead of $here$.

Annotation code may install handlers with code that be called on occasions of activation, deactivation, suspension and resumption. For instance, an annotation may install such handlers as in:

```
  @code1
   there.onActivateOrResume   {code2}
   there.onDeactivate         {code3}
   there.onDeactivateOrSuspend{code4}:
```

3.16 Communication

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. For instance,

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

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

In case a may also communicate with a partner c, SubScript prescribes that these alternatives are marked, by writing $+ =$ instead of $=$ in the definition.

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

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

3.16.1 Multiple Communication Partners

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

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

A normal script may be regarded as an efficient kind of communication, involving only 1 partner. It is possible to express that a can act on its own, but also as a partner in a communication:

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

In case both an eligible a and b have become active, it is up to the script executer to decide what communication (a vs a, b) gets precedence.

Even any number of partners with a given name and signature may be allowed to communicate:

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

So 1 or more calls to a could together do ”hello”, and 1 call to b and 1 or more calls to c could together do ”world”. Unless specified otherwise, a script executer execution must bind a maximum number of partners. That is, a set of partners is allowed if it cannot be extended any more with other active calls.

3.16.2 Communication Body

The body of a communication in ACP may be a normal atomic action, but also an atom that wants to communicate in turn. In SubScript, any kind of script expression is allowed as the body of the communication.

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

Normally a communication body should be built up from atomic actions. Syntacticly it is possible to abuse the freedom, such as in:

```
script a,b = ..
script c,d = (+-)
```

Such definitions are not recommended, but their behaviour is defined. For instance, (+-) behaves like (-) if each of the communication partners belongs to an or-like operator; else the it behaves like (+).

3.16.3 Communication Parameters

Communicating scripts may have parameters. These parameters may be shared; only the first occurrence of shared parameters in the formal parameter lists specifies its type. For instance, a communication with a send action and a receive action would be like:

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

script test1 = s(1) & j:Int receive(j?) {print(j)}
script test2 = s(1) &       receive(1)  {print(1)}
script test3 = s(1) &       receive(2)  {print(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.

3.16.4 Communication over Channels

There is a more convenient notation for common send and receive pairs, so that parameter lists need not be copied. First, script names may end in arrow symbols, denoting send and receive actions over a channel. There need not be a name part before the arrows. 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. E.g.,

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

Calls to the send and receive actions could be like

```
script test1 = a<-(1) & var j:Int a->(j?)
script test2 = b<-(1) & b->(1)
script test3 = c<-(1) & c->(2)
script test4 =  <-(1) &  ->(2)
```

3.16.5 Communication over networks

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

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

For such communication, an additional restriction applies: it should conform a network topology, built at a higher level using variations of the network operator $<<==>>$; this network should also be defined in scripts belonging to the same object or class instance as the prospective send and receive actions.

Send actions and receive actions are normally bound to the nearest by networking operator above them. However, some of such actions may ”fall through”, upwards. For instance:

```
<=(1) ==>
(var i: Int =>(i?) <=i   ==>  var j:Int =>(j?)
```

The receive action $=> (i?)$ cannot possibly communicate in the inner pipe operator $==>$ (since that allows only for sending from left to right); therefore the receive action reaches towards the outer pipe operator, where it may communicate with $<= (1)$. The subsequent send action $<= (i)$ may communicate over the inner pipe, so it will likely communicate with ”$=> (j?)$”.

3.16.6 Asynchronous Communication

To do an asynchronous send 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, only the latter way will yield good 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 the aimed network operator, it cannot fulfill the networking constraint. By using the form $a <= *(1)$, the send action would be launched as a direct subordinate of the nearest network operator ancestor.

3.16.7 Linda Style Communication

The next two variations for receiving over channels had been inspired by the Linda model for tuple spaces. Sometimes it may be useful to do a non-blocking receive:

```
a->?(2)
var i:Int ->?(i?)
```

Such phrases would behave like δ in case no applicable send partner is available. Since δ is a blocking operand, either the description "non-blocking" receive may need to be changed here, or δ should be replaced by ϵ.

Also it may be at times be useful to do a non-consuming receive:

```
a->*(2)
var i:Int ->*(i?)
```

Such receive actions would leave the corresponding send action available for yet another communication. A combination of the two would be a non-blocking non-consuming receive:

```
a->?*(2)
var i:Int ->?*(i?)
```

3.17 Exception handling

A try-catch-finally construct is available, much like the one in Scala. The main differences are that the try and catch parts contain script expressions, rather than Scala code. Also, the catch handlers normally disrupt the try part. This is important, since a thrown exception does not automatically kill the try part, as it would in Scala code.

Suppose an exception would be thrown somewhere inside the script a in

```
try ( a & b ) catch (e: Exception => {println(e)})
```

Then b would be disrupted, as well as a possibly still active part of a. In case the catch handler should not disrupt the try part, specify using $* =>$ that it will act as if it launches a process that will be directly subordinate to the try-catch construct:

```
try ( a & b ) catch (e: Exception *=> {println(e)})
```

$throw\ anException$ may also be used as an operand of a script operator, just as in Scala code

3.18 Syntactic sugar

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

3.18.1 script.. Sections

Often classes will contain sequences of multiple scripts. The repeated word $script$ can be factored out: start a "scripts" section as in

```
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.
- All script names in such a section should start at the same indent level.

Similar sections could be allowed for regular Scala constructs: classes, variables, values and methods. From here on in this paper, we will also leave the phrase $script..$ when it suits.

3.18.2 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 both the following scripts are valid:

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

3.18.3 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. Only a comma needs then to be added as a separator after the script name (or method name):

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

3.18.4 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(s1: String, s2: String)
  = {println(s1+" "+s2)}
a = "hello", "world"
```

3.18.5 Local Value Declarations in Calls

A sequence of a value declaration and a call that initializes that value in the position of an output parameter, may be replaced by a call that also does the declaration. So consider these subsequences with their shorthand notations:

```
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?
```

3.18.6 Prefix Notation

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

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

This latter line is part of the SubScript syntax definition, that is written in SubScript itself; see the appendix. Two other rules in that definition have a choice between sequences. It is possible to have the these alternatives on separate lines, while each line denotes a sequence. For that purpose, the equals symbol must immediately be followed by the semicolon and a plus symbol. The first symbol refers to the white space within a line, and the second refers to the white space between lines (which now binds a bit softer):

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

4. 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. Finally the results from the database are shown in the text area.

In SubScript you can program that in an intuitively clear way:

```
live           = searchSequence...
searchSequence = searchCommand  showSearchingText
               searchInDatabase showSearchResults

searchCommand     = searchButton
showSearchingText = @gui: {outputTA.text = ...}
showSearchResults = @gui: {outputTA.text = ...}
searchInDatabase  = {* Thread.sleep(3000) *}
```

Here $searchCommand$ represents the event of the user pressing the button. It 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. It will automatically be disabled as long as $searchInDatabase$ is going on.

$showSearchingText$ and $showSearchResults$ each write something in the text area., which is represented by the variable named $outputTA$. An annotation makes sure this happen in the GUI thread, as needed.

$searchInDatabase$ represents a lasting database search. This is simulated by a short sleep, but still in a background thread, so that the GUI will not be harmed during the sleep.

If you would to program this functionality in plain Scala, the resulting code will be much more complex, like:

```
  val searchButton = new Button("Go")    {
    reactions.+= {
      case ButtonClicked(b) =>
        enabled = false
        outputTA.text = "Starting search..."
        new Thread(new Runnable {
         def run() {
          Thread.sleep(3000)
          SwingUtilities.invokeLater(new Runnable{
            def run() {outputTA.text="Search ready"
                       enabled = true
          }})
        }}).start
    }
  }
```

4.1 Extending the program

It is easy to extend the functionality of this program. For instance, the search action may also be triggered by the user pressing the Enter key in the search text field ($searchTF$). Another user command could be to cancel an ongoing search in the database. For this the user could press a Cancel button, or press the Escape key. Finally the user may want to exit the application by pressing an Exit button, or by clicking in the close box at the window's upper right corner. But exiting should be confirmed in a dialog box. An extra prerequisite for enabling the $searchCommand$ button would be that the input text field is not empty. For this purpose we would insert an active guard just before the call to $searchCommand$.

```
searchCommand = searchButton + Key.Enter
cancelCommand = cancelButton + Key.Escape
exitCommand   = exitButton   + windowClosing
exit          =   exitCommand@gui:while(!confirmExit)
cancelSearch  = cancelCommand@gui:showCanceledText

searchSequence = guard(searchTF,
                       ()=>!searchTF.text.isEmpty);
                 searchCommand;
                 searchAction / cancelSearch
searchAction   = showSearchingText
                 searchInDatabase
                 showSearchResults

live = searchSequence... || exit
```

$windowClosing$ is a predefined event handling script. $Key.Enter$ and $Key.Escape$ cause calls to the implicit script $vkey$.

The or-parallelism in the $live$ script makes both its operands happen; the left hand side is an eternal loop, but the right hand side ($exit$) may terminates successfully, and then the parallel composition also terminates successfully.

$guard$ is a predefined and rather sophisticated script; it repeatedly evaluates a given test expression and waits for an event at a given component. When the test succeeds, the loop may end:

```
 guard(comp: Component, test: => Boolean) =
   if (test). anyEvent(comp) ...
```

4.2 Progress Monitoring

It is easy to add a process monitor, that adds a sequential number to the output text area, 4 times per second as long as the database search is ongoing. Redefine the script $searchInDatabase$:

```
searchInDatabase   =  {* Thread.sleep(3000)*}
                   || progressMonitor
progressMonitor    = {*Thread.sleep(250)*}
                     @gui:{searchTF.text+=here.pass}
                     ...
```

$here.pass$ returns a loop counter of the sequence in $progressMonitor$.

5. 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 . Informally, the algorithm starts with the first prime number, 2. From the natural numbers, up to a maximum value for practical reasons, it wipes out all multiples of this prime. The next remaining number in the list, 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 in parallel, at least conceptually. 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:

```
object Eratosthenes {
  val toPrint = new NetworkConnection

  public script..

   main(args:Array[String]) =

      generator(2,1000000) ==> (..==>sieve)
                 =={toPrint}==> printer

   generator(s:Int,e:Int) = for(i<-s to e) <=i

   sieve        =   =>p:Int?    @toPrint:<=p;
                  ..=>i:Int? if (i%p!=0) <=i
   printer      = ..=>i:Int? println,i

   <==>(i:Int)  = {}
}
```

The $generator$ script generates numbers and sends these over the network. $printer$ receives numbers and prints those. Both scripts are quite generic and reusable; in principle they may be moved to a trait.

Note that the first number that each sieve receives is its own prime, and it must be forwarded to the printer. Immediately after this reception, the next sieve is created; this is the effect of the optional exit in combination with the parallel loop of sieves ($.. ==> sieve$). Subsequent numbers are forwarded to the next sieve in the chain.

The pipe in $main$ towards the printer contains annotation $toPrint$, which corresponds to the annotation inside the $sieve$ script. $toPrint$ is an instance of class $NetworkConnection$; the implementation of this class will ensure that the sending of a prime by $<= p$ will be redirected towards the printer. Without this provision, the prime would accidentally be forwarded to the brand new next sieve.

6. Parsing

Grammar descriptions may be very concise in SubScript. This eases text parsing. For instance, you would specify the structure of a sequence of comma separated values, and lines thereof, as:

```
csv      = value..","
csvLines = csv "\n"..
```

This would work together with implicit scripts for value and string constants, that parse those. With parameters the result of the parse would be returned:

```
csv      (r :      List[T] ?)      = value,v:T? {!r+=v!} ..","
csvLines(rr: List[List[T]]?) = csv,r:List[T]? {!rr+=r!} "\n"..
```

Comma separated values CSV List Disambiguation

Low level: expectations

GUI similarity: - process expressions record state; less state variables needed - expectations

7. Call Graph Semantics

Template trees Call graph

Semantics by - user defined operators - executers

Messages... - AA events - exclude, suspend - tree buildup/management - activate, success, resume, deactivate - continuations - communication establishment - asynchronous AA result handling - AA execution - AA relocation

8. Implementation

- data structures for graphs - DSL to build these up advantages: easy to develop & install; free language features; relative simple changes to compiler needed - messages - tbd: suspend/resume, communication 2000 lines of code; table

9. Conclusion

ACP addition plus some syntactic sugar enables: - Easy event-driven - Concurrency and Parallelism - Grammar - Dataflow - Logic - Actors - Timing

We see that searchCommand has been defined as an addition of a button and a character. This is something new in programming; I call it ”Item Algebra”. Scala as base language - concise (class definitions); handy case classes and function types - similar definitions with equals: good fit - inspiring syntactic sugar TBD: complete the implementation develop formalism for language definition script blocks

A. Syntax definition

A.1 Syntax ambiguities

The SubScript has been optimized for conciseness and minimal use of parentheses and braces. There result is a set of ambiguities, dealing with . if script.. abstract scripts vs = next line

B. Event handling scripts

C. Execution Manipulation

Using script annotations of the form @code:, the execution of parts of a SubScript program may be manipulated, in cooperation with the script executer. Specific objects with names such as sim, gui, processor may be defined so that they would have the following meanings when used in annotations:

Annotation	Meaning
gui	The Scala code $there$ must be executed in the GUI thread
$dbWriteThread$	The Scala code $there$ must be executed in the given database write thread
$threadPool$	The Scala code $there$ must be executed in a thread in the given thread pool
$processor$	All Scala code $there$ and below must be executed at the given processor
$lowPriority$	The threaded Scala code $there$ should run at a low priority
$lowActionPriority$	The atomic actions $there$ and below have a low priority
$key.typed$	The event handling code $there$ is be executed in response to key typed events
$topology$	The topology for the network $there$
$parentNetwork$	The send or receive call $there$ is directed to the network one level up
$parentPipe$	The send or receive call $there$ is directed to the pipe one level up
$disambiguate$	Operators $there$ and below are disambiguated
$markov$	The program part $there$ and below is managed by a specific Markov system
$markovchance = .5$	The atomic action $there$ has a relative chance to succeed in the given Markov system
$realtimer$	The program part $there$ and below is managed by a specific realtime engine
$startTime = 1\ pm$	The atomic action $there$ starts at 1 PM real time
$duration = 2\ seconds$	The atomic action $there$ succeeds after 2 real time seconds from its start
sim	The program part $there$ and below is managed by a specific timed simulation engine
$startTime = 1\ pm$	The atomic action $there$ starts at 1 PM simulation time
$duration = 2\ seconds$	The atomic action $there$ succeeds after 2 simulation time seconds from its start

D. Formal and Actual Script Parameters

```
Overview of formal and actual parameter use
Formal declaration  Formal type  Actual call  Value of p
p: P  FormalInputParameter[P]  expr  ActualValueParameter ( expr)
p: P?  FormalOutputParameter[P]  varExpr?  ActualOutputParameter (varExpr, {=>varExpr=_)
p: P??  FormalConstrainedParameter[P]  expr  ActualValueParameter ( expr, {=> expr=_)
varExpr?  ActualOutputParameter (varExpr, {=>varExpr=_)
varExpr if(c)?  ActualConstrainedParameter( expr, {=> expr=_}, {_=>c})
formalParam??  ActualAdaptingParameter(_formalParam)
formalParam if(c)??  ActualAdaptingParameter(_formalParam, {=>c})
!! To be updated from source code !!

Suppose a script key(c:Char??) reads a character from the keyboard. How to use this script in numKey(i: Int??), tha
```

```
numKey(i: Int??) =
  _i match (
    case ActualValueParameter(value) => if (i>=0 && i<=9)
    case ActualOutputParameter(_)     => key,c: Char if(c
    case ActualConstrainedParameter(_,constraint) => key,
    case ActualAdaptingParameter(_,formalParameter,constra
```

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...