DeathTalk

A Presentation of Concurrent Normal Computational Logic

The Author is You

http://deathtalk.centerclause.com
Normality is truth. Every use is free.

March, 2018

Contents

1 Abstract

DeathTalk is a presentation of massively concurrent normal computational logic. And as such a presentation of the fundamental technologies that a computational machine depends on. DeathTalk is what Assembler could have been. Technologies that a computational machine does not depend on are only implemented as far as necessary for this presentation.

2 Introduction

DeathTalk is available at http://deathtalk.centerclause.com.

DeathTalk reads an instruction graph from an XML file without read-ahead and without priorities and executes this graph with a traversal precisely in the order given by the document. This traversal or execution may be divided into concurrent executions at any divisible instruction of which there are many. Massive concurrency is the nature of any such graph itself and not a feature of any programming language such as with threads.

The instruction graph is normal which means that there are no further abstractions and concurrency is as broad as possible. The instruction graph may be structured into subgraphs that are named **Outsource**s here. Such an **Outsource** is called by **Routine**s and **Operator**s with **Jump**s. A graph without **Jump**s is a tree graph. A graph with **Jump**s is not a tree graph. Transformations are homeomorphic and therefore a **Routine** call does not pass data. The order of instructions in execution is not changed by a transformation.

Outsources are implemented mathematically which means that they have no state in terms of scope of their own and thus memory is untouched by execution. Routines call explicitly before inferior execution and operators call implicitly after inferior execution. All deformations of the instruction graph are continuous. Therefore routines are implemented bidirectionally which means that the called **Outsource** may call back the calling **Outsource** itself many times before the called **Outsource** finally returns.

The instruction graph can be extended from further XML files at runtime.

Runtime or temporary data is organized with a stand-alone stacked hierarchy of scopes. Such a scope is called **Contract** here. Stand-alone means independency. Independency means that runtime data and business data do not interfere and business logic is not scattered with artificial scopes. There is no scope paradigm. A **Contract** is initiated programmatically like a routine is called. Thus both calling and called routine share a **Contract** with access privileged by the calling routine.

Concurrency is applicable to the smallest of expressions. Addition is an example for a divisible instruction. It is already concurrent and does not use memory. It takes two independent operands that can be obtained in any way from inferior execution and gives a result that is independent to superior instructions. A graph of such instructions is massively concurrent. There is no interference on this lower level with business logic.

A concurrent instruction works without programmatic control and always waits by default. If the result is determined by some parameters already then a concurrent instruction may not wait for the remaining data which is a rational race. An example for that is a logical OR.

Some of these technologies are not supported by modern compilers such as contracts and bidirectional routines, though, they can be implemented with modern assemblers.

Some other technologies are not supported by modern computers themselves such as concurrent instructions. Massive concurrency is not feasible with modern computers since data is stored with synchronized particles and not available everywhere like radio waves. A machine like that itself is feasible since the brain is massively concurrent.

The presented technologies are missing in modern computing. Although the success of object-oriented programming languages and anonymous functions expresses such a need. A class is similar to contracted scopes. Getters and Setters do not require a scope. Anonymous functions asking for bidirectionallity.

Massive concurrency is simply not practical on a modern computer. Threads are too heavy for such an implementation due to implicit scopes and designated memory.

DeathTalk does not respect the limitations of modern compilers and computers. DeathTalk is a presentation of concurrent normal computation logic. DeathTalk is implemented to the point to make that point only and not further.

3 Conclusion

The conclusion can only be that once a language is implemented that supports the presented technologies then all other software is dead. Implementations of business logic will have to be rewritten since bidirectional routines that are normal are not supported by common languages and wide-spread instructions that break a single execution do not determine the state of a concurrent execution.

Technologies essential to massive concurrency such as concurrent arrays or streaming loops are simply not applicable with modern architectures. Furthermore the synchronization of execution and scopes scatters business logic in modern computing. Threads run away and require additional instructions for control.

Eventually concurrency is to be enabled with modern computing and to be disabled with a normal computational logic like DeathTalk.

DeathTalk is normal. Available software and hardware is abnormal.

4 Download

DeathTalk is available at http://deathtalk.centerclause.com as a NetBeans 8.2 Java 8 Project. The NetBeans Project Folder contains two additional folders. Folder resource contains examples. Folder debug is for stuff generated during execution.

DeathTalk is consider as truth and thus every use is free. The author has no interest in implementing DeathTalk any further without a truly massively concurrent architecture to test against. A machine that handles memory with waves and not with particles.

The main executable is class

com.centerclause.deathtalk.Realm.

Html documentation of the Dictionary is generated into folder resource by class

com.centerclause.deathtalk.documentation.Documentation.

5 Concepts

There is no simple description of DeathTalk. DeathTalk is made of a number of small classes that work together like a machine. The central method to this machine is **run** of class com.centerclause.deathtalk.machine.process.Work.

Most concepts cannot be described independently. Therefore a brief description of each concept is given here. And detailed descriptions follow later.

Algorithm An executable graph of instructions is called Algorithm.

Word, Part, Dictionary A unique piece of computational logic is called **Word**. Many words are atomic such as *value*:*int* or *value*:*add* and some words are micro-expressions such as *value*:*increment*. A **Word** is abstract logic. A set of **Word**s is called **Part**. A set of **Part**s is called **Dictionary**. A source code is read from a stream and build from looking into a **Dictionary**.

Instruction An **Instruction** specifies a **Word** with additional information from the source code. An **Instruction** is told the direction of execution and executes a **Word** accordingly.

Cycle A **Cycle** connects instructions as a graph. Direct inferior **Cycle**s are visited in order of ascending indices. A direct inferior **Cycle** is commonly called a child and the direct superior **Cycle** is commonly called the parent.

Execution An instruction graph is executed with a traversal. A node may be executed more than once due to loops, routines or vector execution. Such an instruction graph is executed with an accordingly deformed traversal.

Direction An execution starts at the top of the instruction graph and moves Down. The direction is changed at a leaf node and the execution moves Up. The direction from one child cycle to the next child cycle is called Right. A loop may require a Full Left from the last child cycle to the first child cycle.

Parameter, Perimeter, Diameter An instruction is executed at different stages of a deformed traversal. Therefore data is classified as **Diameter** if generated by inferior **Node**s. Data is classified as **Perimeter** if generated by the same **Node** from for example XML attributes or Text Nodes. And data is called **Parameter** if generated by superior **Node**s on the way down the instruction graph.

Command A **Command** takes values and gives no result.

Control A **Control** is a built-in combination of **Command**s and gives no result such as *exe*:*ifThenElse* or *exe*:*whileDo*.

Operation An **Operation** takes values and gives a value. Values are exchanged with runtime stacks and thus **Operation**s may technically give many results. However, no such standard operation is implemented. Multiple results may occur with routines and operators, though.

Value **Value**s are exchanged with runtime stacks and do not require a binding. Thus **Value**s behave like anonymous variables. L- and R-values do not occur since stacks work at runtime.

Assignment An assignment changes the value of a variable.

Bind A bind associates a unique name with a value in a scope.

Reference DeathTalk is implemented in Java and maps References accordingly.

Expression An **Expression** combines values to other values.

Impression An **Impression** evolves a single operand or **Value** of an **Expression** like a path.

Goods All **Value**s on all data stacks at some point of an execution are called **Goods** and demonstrate that more than one stream of data may be implemented. Two streams are implemented **Register** and **Hold**.

Infinite Values, Hold A Bit is actually Infinity. Logical operators such as And and Or are infinite arithmetic. **InfiniteValue**s are kept on a separate data stack called **Hold** in order to demonstrate that more than one stream of data may be implemented and since all finite values are actually combinations of infinite values.

Finite Values, Register Numeral and literal values are finite and kept on a stack called **Register**.

State A **State** is the position on the instruction graph and stores references to the current, last and next **Cycle** such that the direction of execution is determined.

Work **Work** is a serial string over **Cycle**s. Execution is divided into **Work**.

Progeny, Sibling Execution is divisible at many instruction nodes since direct inferior nodes are independent. Such a set of direct inferior nodes is called **Progeny**. A single such direct inferior instruction is called **Sibling**. Only static **Progeny**-s are implemented. Dynamic **Progeny**-s such as streams of **Sibling**s are not implemented.

Rational Race The result of a divisible **Operation** may already be determined by a single **Sibling** such as with a logical And. This determines a rational race and the node may or may not wait for all **Sibling**s for further execution.

Outsource An **Outsource** is the definition of otherwise recurring instructions with a subgraph and thus defines a routine or an operator.

Stretch A Stretch is the set of instructions between two **Jump** nodes that divide an instruction graph into **Outsources** and **Insources**. A Stretch is executed forth and later back. A Stretch is executed at many different times. The division of execution into **Work** is synchronized with the division of source into **Stretches**.

Array Arrays are implemented without type for simplicity of the presentation. Higher dimensions are nested since such arrays are traversable and as such massively concurrent.

Scope A **Scope** is a map of uniquely named values.

Article An **Article** is an absolute **Scope** that may be stored permanently. IO is not implemented.

Contract A **Contract** is a relative **Scope** for temporary or runtime data. **Contract**s are initiated programmatically only.

Type A type defines the nature of a **Value**. Some standard types and operators are implemented. Complex types may be defined and associated with according **Routine**s and **Operator**s in a hierarchy similar to packages in Java. Type safety is weak for simplicity of the presentation.

Routine A **Routine** is the execution of an **Outsource** before the execution of inferior nodes and may be called back bidirectionally from the **Outsource** many times. Data is generated during this communication.

Insource An **Insource** is a direct inferior node of a **Routine**.

Callback A **Callback** returns back and forth from an **Outsource** to an **Insource** of the calling **Routine**.

Binary Operation A System of Binary Operations is implemented for book-keeping of the many combinations of operations on standard and complex types. Many binary operations are not yet implemented since it is not required for this presentation. All implemented Binary Operations of Standard and Custom Types are executed within
com.centerclause.deathtalk.value.Calculation .

Operator An **Operator** is the execution of an **Outsource** after the execution of inferior nodes and thus is unidirectionally. Data is passed at once altogether.

Process A **Process** initiates the execution of an instruction graph.

Extension A special command *exe*:*extend* may be executed in order to read more instructions from a stream and extend the instruction graph accordingly.

Realm A **Realm** holds data shared by many **Process**es which is not tested.

Jack A **Jack** is a **FiniteValue** that wraps a **FiniteValue** in order to change the behavior for example from a variable to a constant value.

Sanctuary Rational races may produce run-away threads that may or may not be allowed to finish and are moved to a **Sanctuary** once the **Process** itself is actually finished.

Rhythm **Rhythm** is the definition of a continuous alternation of operands and operation. Rhythm gives grammar that allows for validation of the source code and generation of a language called **Epilog** which increases readability. **Rhythm** also has the effect of shallow data stacks since operands are kept closely to operations.

Arity **Arity** is the number of values required and defined with laws of **Rhythm**.

Epilog With applied **Rhythm** DeathTalk XML or Dtx translates immediately into a higher programming language. This language is called **Epilog** since that is about it. **Epilog** is not defined in every detail. **Epilog** is not tested. There is no parser and no compiler. **Epilog** code is simply produced along the way.

Comment A comment is either an XML comment or an annotation on the next instruction.

Breakpoint A Breakpoint is an annotation at which execution stops if the JVM is run in Debug Mode. Execution continues if an according thread is interrupted.

6 Dictionary of Machine Words

A DeathTalk source code is written in XML since their are plenty of XML tools and an XML document is itself a graph. Therefore XML rules apply. XML items are read from a stream without read-ahead and compared against the **MachineDictionary**, see table 1.

Validation takes place with a builtin **Rhythm** and can be switched off with *exe*:*rhythm*="*false*" on *exe*:*deathTalk* itself or any other node for all inferior nodes unless switched on again. A Schema is not provided since DeathTalk is not at a productive stage. A Schema could be generated from **Rhythm**.

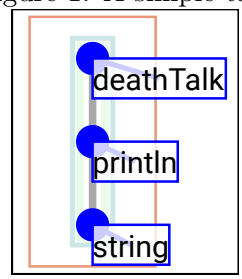
Table 1: All Parts of the Machine Dictionary

default prefix	namespace
exe	dict://deathtalk/execution
value	dict://deathtalk/value
cmd	dict://deathtalk/command
contract	dict://deathtalk/contract
art	dict://deathtalk/article
journal	dict://deathtalk/journal
rhythm	dict://deathtalk/rhythm
jack	dict://deathtalk/rhythm
debug	dict://deathtalk/debug
system	dict://deathtalk/system
massive	dict://deathtalk/massive
a	resource://deathtalk/annotation

7 The Obligatory Greeting

DeathTalk is normal. Available software and hardware is not normal. Therefore the obligatory greeting is not given. Instead DeathTalk can only say Good Bye for a first example, see Listing 1 and figure 1 for the according graph.

Figure 1: A simple talk



Listing 1: A simple talk

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <exe:deathTalk
3     xmlns:exe="dict://deathtalk/execution"
4     xmlns:value="dict://deathtalk/value"
5     xmlns:cmd="dict://deathtalk/command">
6     <cmd:println>
7         <value:string>Good Bye World!</value:string>
8     </cmd:println>
9 </exe:deathTalk>
```

8 Virtual Nodes

A virtual node of the instruction graph or algorithm of DeathTalk comprises of a **Cycle**, an **Instruction** and a **Word**, see Figure 2. Node always means Virtual Node.

A streamed XML source code is read with **DtxReader** within **Realm**. There is no read-ahead. If a read **Word** is a **FactoryWord** which is derived from **MachineWord** which is derived from **Word** then an associated **Instruction** is built by that **Word**. An **Instruction** stores **Perimeter**s read from source code. An **Instruction** builds an associated **Cycle** that is attached to the direct superior node in order of the source code.

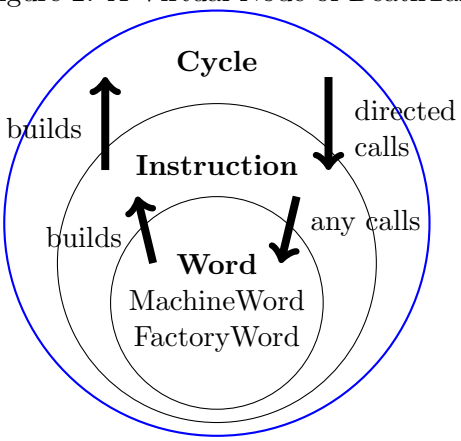
A **Cycle** stores its relations to direct superior and inferior **Cycle**s of the graph. These relations are commonly called parent and children. An **Instruction** specifies a **Word** with **Perimeter**s read from source code. A **Word** stands for a specific unique piece of logic.

An algorithm is executed with **DeathTalk** within **Realm**. A **Cycle** has a specific location in the algorithm from where it calls its associated **Instruction**. An **Instruction**s calls its

associated **Word** at specific events such as a **Shift** down or up the graph during execution. **Word**, **Instruction** and **Cycle** have no runtime state.

A derivate of a **Cycle** determines the behavior of the algorithm.

Figure 2: A Virtual Node of DeathTalk



9 Execution with Stretch and Work

Execution is divided into **Work**s. **Work** literally means physical work since a node acts like a force and a traversing shift acts like a displacement.

Execution is started by calling the method **run** of **DeathTalk** which then initiates a **Process** that starts an initial **Stretch** that starts an initial **Work**.

A **Stretch** is the range of instruction nodes between and including two **Jump** nodes. **Stretch**es are divisions of an instruction graph in space or according to source code at **Routine**s and **Operator**s. **Work**s divide an according deformed traversal or execution temporarily or at runtime. **Stretch**es are synchronized with **Work** in order to determine the direct superior **Routine** node of an **Insource** at runtime as discussed in Section 20. A new **Stretch** always begins with a new **Work**. A **Stretch** may be executed at many different times first forth and later back.

A displacement at execution from one virtual node to another directly related node is called **Shift**. A displacement at execution from one virtual node to another not directly related node is called **Jump**.

The core method of DeathTalk is the method **run** of **Work**. This method requires a **State** that tells an exact position on the instruction graph. **State** stores where execution is coming from and where it is which makes an edge with a determined direction. **Work** asks a **Cycle** for the next **Cycle** and **State** performs that shift.

Works method **run** returns on either of three conditions. Firstly, if a given end **State** is met in which case **Work** is actually finished. Secondly, if the current **Cycle** is a **ProgenyCycle**

in which case inferior **Work**s are initiated according to the definition of that **ProgenyCycle**. The same **Work** is picked up once all inferior **Work**s have finished. Thirdly, **Work** returns if its execution is overruled by either a finished rational race or an exception.

10 Standard Types

DeathTalk supports *value*:*int*, *value*:*real* and *value*:*string* only. **Real** is mapped to Javas float. DeathTalk is a presentation of a normal computational logic that is massively concurrent. More types are not required since a machine that supports massive concurrency is not available and thus according types unknown.

Int and **Real** are parsed with Javas Integer and Float parser. According notations apply. See Listing 2 as example.

Listing 2: The Standard Types of DeathTalk

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <exe:deathTalk
3      xmlns:exe="dict://deathtalk/execution"
4      xmlns:value="dict://deathtalk/value"
5      xmlns:cmd="dict://deathtalk/command">
6    <cmd:println>
7      <value:int>1</value:int>
8    </cmd:println>
9    <cmd:println>
10     <value:real>1.1</value:real>
11   </cmd:println>
12   <cmd:println>
13     <value:string>string</value:string>
14   </cmd:println>
15 </exe:deathTalk>
```

11 Normal Runtime Data with Contracts

The implementation of runtime data management is normal since their is no paradigm on the creation of scopes. A **Scope** is a container that maps one value to a name that is unique within that container. There is no interference with the execution itself at any point. Therefore business data and runtime data are strictly separable.

Runtime data is temporary and therefore has no location in storage. Runtime data is managed with a stacked hierarchy of scopes in memory. A **Scope** of such a hierarchy is called **Contract**.

All **Work** runs within a **Contract**. The initial **Work** runs within the initial **Contract**. A

further **Contract** is initiated independently only and attached to the **Contract** in which it is created. This further **Contract** is then set as top **Contract** of the **Work** in which it is created. A **Contract** is initiated on **Shift**s down the instruction graph and deleted on **Shift**s up the instruction graph by the same virtual node. The direct superior **Contract** is then made the top **Contract** again for the same **Work**.

Contracts are not created implicitly through routine calls or other. Therefore successive **Routine**s run within the same **Contract** and may exchange data by this way. Access to a **Contract** may be restricted on initiation. Access to the initial **Contract** is defined with the attribute *contract*:*initial* on *exe*:*deathTalk*. Access to inferior **Contract**s is defined with the attribute *contract*:*access* on the *contract*:*further* node that initiates a **Contract**. Options are private, protected or public according to table 2.

A reference of a **Value** is mapped in the top **Contract** with *contract*:*bind* which takes both a string as key and any value from top of **Register**. A bind to superior **Contract**s is not implemented. A bind is not type specific. The **Value** is left on **Register** for further computations from which it can be removed by *cmd*:*expression*.

A reference to a **Value** of a **Contract** that is put on **Register** is made with a *contract*:*referInt*, *contract*:*referReal*, *contract*:*referString* or a *contract*:*referArticle*. These nodes are typed in order to introduce at least some type safety which is otherwise missing for simplicity of the presentation. A reference to a **Value** of a superior **Contract** requires the attribute *contract*:*super*="*N*" where N is the number of superior levels. The string parent refers to the first superior level. An exception is thrown if access is denied. Nodes for exception handling are not implemented for simplicity of the presentation.

Table 2: Access to Contracts

contract:*access*="*public*"	Data is accessible from all inferior Contracts.
contract:*access*="*private*"	Data is not accessible from inferior Contracts directly. Data is accessible to outsources through bidirectional callbacks. Data is passed per definition to operators.
contract:*access*="*protected*"	Data is accessible only from direct inferior Contracts.
contract:*access*="*N*"	Data is accessible only from inferior Contracts of the next N levels. This is provided only since the implementation works in this way anyway.

Listing 3: A simple example of Contract

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk name="Example␣of␣further␣of␣contract."
                xmlns:exe="dict://deathtalk/execution"
                xmlns:value="dict://deathtalk/value"
                xmlns:cmd="dict://deathtalk/command"
                xmlns:contract="dict://deathtalk/contract">
  <contract:further access="protected">
    <!-- delete reference that is left by bind on up -->
    <cmd:expression>
      <contract:bind>
        <value:string>name</value:string>
        <value:string>success</value:string>
      </contract:bind>
    </cmd:expression>
    <cmd:println>
      <!-- access same contract -->
      <contract:referString>
        <value:string>name</value:string>
      </contract:referString>
    </cmd:println>
    <contract:further>
      <cmd:println>
        <!-- access protected superior contract -->
        <contract:referString contract:super="parent">
          <value:string>name</value:string>
        </contract:referString>
      </cmd:println>
    </contract:further>
  </contract:further>
</exe:deathTalk>
```

12 Words

Each **Word** is either a **Command** or an **Operation**. A **Control** such as *exe*:*ifThenElse* is a builtin combination of **Command**s. A **Command** may take operands and does not give operands. An **Operation** may take operands and gives operands which is a significant difference.

Each **Work** keeps its runtime data on stacks managed with **Goods.** The number and nature of data stacks of a productive design of a computational logic depends on conditions that are not known here. The number of stacks implemented is chosen to be greater than one in order to avoid oversimplifications such as a definition of Success or Failure. A stack called **Hold** keeps **InfiniteValue**s and a stack called **Register** keeps **FiniteValue**s.

A Bit is an **InfiniteValue**. All standard types could technically be made from bits in software. An **InfiniteValue** is also the result of a comparison or a logical operation which are *exe*:*and* and *exe*:*or* and *exe*:*not*. The arithmetic of these operations is that of infinity.

An operand is either a **Diameter** or a **Perimeter** or a **Parameter**. A **Diameter** is expected once all inferior nodes have been visited. A **Perimeter** is given statically on the same node by either an XML attribute or an XML text node. A **Parameter** is expected immediately and thus produced by a superior instruction node.

An **Operation** that combines **Diameter**s exclusively is an **Expression** such as addition, multiplication, concatenation and the like. An **Operation** that combines **Diameter**s, **Perimeter**s and **Parameter**s is called **Impression** which includes scope and array access and is defined in Section 15. Specialized variants are documented accordingly.

An **Impression** makes a path and typically changes one operand of an **Expression** successively. **Impression**s are not as easy to read in XML as **Expression**s since operands appear outside of an instruction node. Readability is improved with syntactic sugar nodes defined with **Rhythm** in Section 21.

All increments are pre-increments. A post-increment is not supported since its execution is out of order.

A **Command** does not put new operands on stacks per definition. An **Expression** or a chain of **Expression**s leaves data on stack. *cmd*:*expression* is a **Command** that removes one operand from **Register** to close an expression.

See Listing 4 for examples including a multiplication of execution with *exe*:*times*.

Listing 4: Simple Commands and Expressions

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk
    xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:rhythm="dict://deathtalk/rhythm">
  <!-- a command gives nothing -->
  <cmd:println>
    <!-- an expression produces an operand -->
    <!-- a binary expression -->
    <value:add>
      <!-- two int diameters for value:add -->
      <!-- a text node is a perimeter -->
      <value:int>1</value:int>
      <value:int>2</value:int>
    </value:add>
  </cmd:println>
  <cmd:println>
    <!-- unary expression -->
    <value:increment>
      <!-- one diameter for value:increment -->
      <value:int>1</value:int>
    </value:increment>
  </cmd:println>
  <!-- remove a left over operand from stack -->
  <cmd:expression>
    <value:int>1</value:int>
  </cmd:expression>
  <!-- syntactic sugar helps with readability -->
  <rhythm:for>
    <!-- parameter to exe:times -->
    <value:int>3</value:int>
    <!-- a control is a command -->
    <exe:times exe:name="demo">
      <cmd:println>
        <!-- an attribute is a perimeter -->
        <exe:number exe:name="demo"/>
      </cmd:println>
    </exe:times>
  </rhythm:for>
</exe:deathTalk>
```

13 Concurrent Binary Operations

A truly binary operator is defined as a mathematical function and thus takes independent operands here called **Diameter**s since these operands are supposedly produced by inferior nodes. A binary operator is truly binary if the result is not assigned to one diameter.

Independency means that operands may be produced concurrently which also means that operands are then produced out of order. Every binary operation is divisible into two **Work**s which in succession gives automatic massive concurrency at machine level. Such machine instructions are not available as of today since technology for memory is still very limited. Data is distributed with particles and not with waves.

A generic divisible node *exe*:*divide* is implemented in DeathTalk that runs each direct inferior node in a separate **Work** anyway. A further concurrent implementation of binary operations in software is therefore straightforward. Operands are spilled in order of nodes automatically onto the stack of the **Work** of the **Operation**. A few of these operators are available with **Part massive** of the **MachineDictionary**. A default application of concurrent binary operations is not practical since threads are too heavy and other concurrent software technology is not available. See Listing 5 for an example.

Logical binary operators such as And and Or are incompletely determined and all other operators are completely determined. The operator And is already determined if one operand equals False. The operator Or is already determined if one operand equals True. In concurrent form these operations may therefore return before all inferior nodes have returned which makes a rational race. The handling of operands that are left behind depends on the application and further discussion is not necessary here.

Programmatic threading at business level is not required with concurrency available at machine level which should load balanced well automatically if necessary at all since it is as fine grained as possible and therefore normal. Business logic is then free of extra logic for threading.

The brain is massively concurrent and therefore massively concurrent technology will be available one day.

Massive concurrency is the nature of a normal instruction tree. Massive concurrency is intrinsic and therefore automatic. Instructions for quitting are abnormal. break, continue and return only determine the state of a single execution. The state of concurrent execution is not determined with break, continue and return. Software will have to be implemented without quitting instructions. All nodes have to finish normally. This is always possible. And stating a return value only improves readability. Software design patterns will have to change with future higher programming languages.

An instruction that disables further concurrent execution of inferior nodes is implemented with *exe*:*serial*. This would not prevent execution of other **Work** in case of a quitting instruction. It only changes the behavior of execution of inferior **Work**s from concurrent to serial.

Listing 5: Divisible Operations

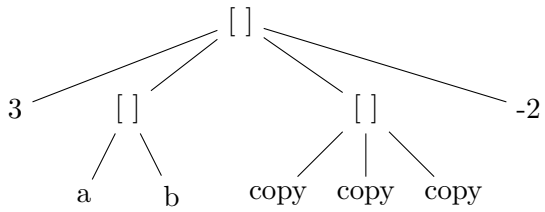
```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk
    xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:massive="dict://deathtalk/massive">
  <cmd:println>
    <!-- serial add -->
    <value:add>
      <value:int>0</value:int>
      <value:int>1</value:int>
    </value:add>
  </cmd:println>
  <!-- run inferiors separately -->
  <exe:divide>
    <cmd:println>
      <value:string>probably</value:string>
    </cmd:println>
    <cmd:println>
      <value:string>too slow</value:string>
    </cmd:println>
    <cmd:println>
      <value:string>to tell</value:string>
    </cmd:println>
  </exe:divide>
  <cmd:println>
    <!-- concurrently add execution numbers -->
    <massive:add exe:name="operand">
      <exe:number exe:name="operand"/>
      <exe:number exe:name="operand"/>
    </massive:add>
  </cmd:println>
</exe:deathTalk>
```

14 Arrays

Arrays are not typed for simplicity of the presentation of a normal computational logic. **Array**s are nested since this results a tree graph. A nested **Array** can be executed or traversed like any other tree graph. If all elements are independent then the **Array** can be operated with massive concurrency which is not supported with current hardware and not practical with current multitasking software.

The same exemplary nested **Array** is defined by all three notations, firstly, the following line of **Epilog**, secondly, the below tree graph and thirdly, Listing 6 of DeathTalk.

```
define[3,define["a","b"],(declare[3] = "copy"),-2]
```



In DeathTalk a nested **array** is defined by an initializer list or element by element with *value*:*arrayDefinition*.

A nested **Array** of any dimensions is declared with *value*:*arrayDeclaration*. Elements are initialized with a **NullFiniteValue**. The extension of an **Array** with another dimension is implemented with a combination in **ArrayDeclarationFiniteValue**. Each leaf element is assigned a new **ArrayFiniteValue** of the specified *value*:*dimension*. Further details such as the presence of **Value**s that are not null are not implemented.

All leaf elements of a nested **Array** are set each with a copy of a given **Value** by *value*:*arrayDesign* through a traversal. This binds a new **Value** to each leaf element. An assignment is not available through traversal since **Array**s are implemented without type for simplicity.

This traversal is named **ArrayZip** and is also applied to print all elements.

More details are discussed in Section 15 and shown in Listing 7.

Listing 6: Arrays

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
     xmlns:cmd="dict://deathtalk/command">
  <cmd:println>
    <value:arrayDefinition>
      <value:arrayElement>
        <value:int>3</value:int> <!-- 1st element is int-->
      </value:arrayElement>
      <!-- second element makes a nested array -->
      <value:arrayElement>
        <value:arrayDefinition>
          <value:arrayElement>
            <value:string>a</value:string> <!-- 1st of nest -->
          </value:arrayElement>
          <value:arrayElement>
            <value:string>b</value:string> <!-- 2nd of nest -->
          </value:arrayElement>
        </value:arrayDefinition>
      </value:arrayElement>
      <!-- third element makes a nested array -->
      <value:arrayElement>
        <!-- combine array and value -->
        <value:arrayDesign>
          <value:arrayDeclaration>
            <value:dimension>
              <value:int>3</value:int>
            </value:dimension>
          </value:arrayDeclaration>
          <!-- set a copy to all elements -->
          <value:string>copy</value:string>
        </value:arrayDesign>
      </value:arrayElement>
      <value:arrayElement>
        <value:int>-2</value:int> <!-- 4th element is int -->
      </value:arrayElement>
    </value:arrayDefinition>
  </cmd:println>
</exe:deathTalk>
```

15 Articles

DeathTalk supports objects with **Article** which is a **Scope** which is a map. **Article** is an absolute **Scope** and therefore suitable for business data and for IO which is not implemented.

Part **article** provides basic in-scope commands and operations. Part **value** provides basic out-scope commands and operations for binding and referring values of **Article**. A complete set of commands is not implemented for simplicity of the presentation of a normal computational logic.

An **Article** can be made an instance of a **Type** as discussed in Section 20.5 and thereby associated with **Routine**s and **Operator**s. Bidirectional **Routine**s can be used as constructors and so on. Therefore **Article** is a type that is defined weakly at runtime through bidirectional **Routine**s and unidirectional **Operation**s.

Type safety, classes and inheritance or object orientated programming is a matter of a higher programming language and is not defined here.

The following three lines of **Epilog** are generated with Listing 7 and print `def`. The at sign symbolizes **Article** and the paragraph sign symbolizes **Contract**. The **Diameter** to `println` is an **Impression** or path as discussed in Section 16.

```
1 expression((§ bind "object" = (@new)));
2 expression((bind §"object"@"array" = define["abc","def"]));
3 println((§"object"@"array"[1]));
```

16 Impressions

An **Impression** is an operand that is build from a sequence of dereferences similar to a path. It is implemented as a stack in a **Value** on the stack of **Work**.

A *value*:*impression* node takes a **Value** from stack before execution of its inferior nodes. The **Value** is wrapped inside an **ImpressionFiniteValue** which is then put on stack instead. The contained **Value** can be dereferenced with two operations. *value*:*articleAt* dereferences a contained **Article** with a given string as name and *value*:*arrayAt* dereferences a contained **Array** with a given int as index. After execution of all inferior nodes a *value*:*impression* node takes an **ImpressionFiniteValue** from stack and puts its contained **Value** on stack instead.

Listing 7 gives an example.

Listing 7: Impression with Article and Array

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:contract="dict://deathtalk/contract"
    xmlns:art="dict://deathtalk/article">
  <cmd:expression>
    <contract:bind>
      <value:string>object</value:string>
      <art:new/>
    </contract:bind>
  </cmd:expression>
  <cmd:expression>
    <value:articleBindAs>
      <contract:referArticle>
        <value:string>object</value:string>
      </contract:referArticle>
      <value:string>array</value:string>
      <value:arrayDefinition>
        <value:arrayElement>
          <value:string>abc</value:string>
        </value:arrayElement>
        <value:arrayElement>
          <value:string>def</value:string>
        </value:arrayElement>
      </value:arrayDefinition>
    </value:articleBindAs>
  </cmd:expression>
  <cmd:println>
    <rhythm:rereferString>
      <contract:referArticle>
        <value:string>object</value:string>
      </contract:referArticle>
      <value:impression>
        <value:articleAt>
          <value:string>array</value:string>
        </value:articleAt>
        <value:arrayAt>
          <value:int>1</value:int>
        </value:arrayAt>
      </value:impression>
    </rhythm:rereferString>
  </cmd:println>
</exe:deathTalk>
```

17 Conditional Execution

DeathTalk supports conditional execution or branches with *exe*:*ifThen* and *exe*:*ifThenElse*, see Listing 8. *exe*:*ifThen* requires exactly two direct inferior nodes and *exe*:*ifThenElse* requires exactly three direct inferior nodes. The first node is the condition on which to execute the next node. Further conditions are to be nested. *exe*:*parent* groups nodes.

A basic *exe*:*switch* is implemented but not documented here.

Another branch is possible say Take that would take an integer and execute the direct inferior node of that index.

Listing 8: Conditional Execution

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
xmlns:value="dict://deathtalk/value"
xmlns:cmd="dict://deathtalk/command"
xmlns:contract="dict://deathtalk/contract">
  <exe:ifThen>
    <value:smallerThan>
      <value:int>1</value:int>
      <value:int>2</value:int>
    </value:smallerThan>
    <cmd:println>
      <value:string>then of ifThen</value:string>
    </cmd:println>
  </exe:ifThen>
  <exe:ifThenElse>
    <value:smallerThan>
      <value:int>2</value:int>
      <value:int>1</value:int>
    </value:smallerThan>
    <cmd:println>
      <value:string>then of ifThenElse</value:string>
    </cmd:println>
    <exe:superior>
      <cmd:println>
        <value:string>else of ifThenElse</value:string>
      </cmd:println>
    </exe:superior>
  </exe:ifThenElse>
</exe:deathTalk>
```

18 Temporary Recurring Nodes

Sections of recurring nodes are either repetitive in time and implemented with loops that are executed in place. Or recurring nodes are not repetitive in time and implemented with designated subgraphs, see Section 20.

18.1 Loops

Of all serial loops only *exe*:*whileDo* is implemented in DeathTalk for simplicity of the presentation. See Listing 9 which is basically a for-loop and generates the below **Epilog**.

```
1 expression((§ bind "count" = 0));
2 while (§"count" < 3) do
3 {
4   println("loop");
5   expression((++§"count"));
6 }
7 expression((§ unbind "count"));
```

Listing 9: Conditional Execution

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:contract="dict://deathtalk/contract">
  <cmd:expression>
    <contract:bind>
      <value:string>count</value:string>
      <value:int>0</value:int>
    </contract:bind>
  </cmd:expression>
  <exe:whileDo>
    <value:smallerThan>
      <contract:referInt>
        <value:string>count</value:string>
      </contract:referInt>
      <value:int>3</value:int>
    </value:smallerThan>
    <exe:superior>
      <cmd:println>
        <value:string>loop</value:string>
      </cmd:println>
      <cmd:expression>
        <value:increment>
          <contract:referInt>
            <value:string>count</value:string>
          </contract:referInt>
        </value:increment>
      </cmd:expression>
    </exe:superior>
  </exe:whileDo>
  <cmd:expression>
    <contract:unbind>
      <value:string>count</value:string>
    </contract:unbind>
  </cmd:expression>
</exe:deathTalk>
```

19 Types

DeathTalk provides with just a few standard types for simplicity of the presentation of a normal computational logic. More complex **Type**s can be declared and organized together with then associated **Routine**s and **Operator**s in a dynamic hierarchy similar to Java packages. The definition of a **Type** with a class and according **Type** safety is not implemented since that is a machine independent matter of a higher programming language. A machine is defined in terms of **Operation**s and not in terms of operands. An **Article** can join a **Type**, though, in order to call **Operator**s and specific **Routine**s implicitly, as discussed in Section 20.5.

The **Type** hierarchy includes all complex types. Each **Work** is executed within a current **Type**. The initial **Work** is executed within the root **Type**. Basic **Command**s and **Operation**s to the **Type** hierarchy are listed below.

type:*root*	puts a reference to the root **Type** of the hierarchy on stack.
type:*new*	declares a new **Type** relative to the current **Type** of **Work** and puts a reference on stack.
type:*refer*	takes a **String** from stack, looks up a **Type** by that name in the current working **Type** and puts an according reference on stack.
type:*change*	takes a reference from stack and changes the current working **Type**.
type:*open*	combines *type*:*refer* and *type*:*change*.
type:*close*	changes the current working **Type** to its direct superior **Type**.

20 Spatially Recurring Nodes

Sections of nodes that repeat in source code or space are commonly deformed into subgraphs that are callable from special nodes out of place. Such a callable subgraph is named **Outsource** in DeathTalk. In DeathTalk a **Jump** call of an **Outsource** takes place either explicitly with a bidirectional **Routine** or it takes place implicitly with a unidirectional **Operator**.

In DeathTalk a deformation into **Outsource**s is normal since the traversal or execution takes place in precisely the same order from which it were derived analytically. Data is fetched when required in case of a **Routine** which means that an **Outsource** may callback **Insource**s of the calling **Routine** in any order and as many times as necessary until return. In case of an **Operator** all data is available already and passed at once.

Through **Jump**s execution deviates from the order given in source code and a lot can change between a call and a callback. DeathTalk maintains continuity over **Call**s and **Callback**s according to source code automatically.

An **Outsource** is mathematical in that it has no runtime state or **Scope**. Some nested **Routine** calls may run within the same **Contract** and exchange data this way. Others may initiate further even privileged **Contract**s and restrict data exchange to callbacks. DeathTalk

maintains **Contract** continuity automatically in order for nodes that are called back from an **Outsource** at runtime to run in the same **Contract** as their direct superior **Routine** node according to source code.

A tree traversal or execution of an instruction graph is directed with continuous events at **Operation** and **Command** nodes. The division of an instruction graph into subgraphs introduces discontinuity events of **Void** in a then deformed traversal since the direct inferior nodes in source are not the direct inferior nodes at runtime. Instead execution jumps to another node that is defined somewhere else in source code.

Rhythm introduces concretion with grammar in order to approximate continuity at abstract **Void** events and generate **Epilog**. Execution does not require continuity of operands and operations. To the machine **Rhythm** is just syntactic sugar. **Rhythm** improves readability and is discussed later for simplicity.

20.1 Outsources

An **Outsource** makes a proper **Value** that can be passed around and bound. The value contains the root node of the **Outsource**.

An instruction graph with **Outsource**s is read and build without change. At runtime, though, execution bounces at an **Outsource** node as if it is a leaf node. An **Outsource** can only be entered with special explicit or implicit **Jump** calls.

Listing 10: An Outsource is a Value

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command">
  <cmd:expression>
    <value:outsource>
      <cmd:println>
        <value:string>never executed</value:string>
      </cmd:println>
    </value:outsource>
  </cmd:expression>
</exe:deathTalk>
```

20.2 Routines

DeathTalk is a normal computational logic. In DeathTalk deformations of an instruction graph are continuous. Source and destination graphs are homeomorphic. Thus the deformed traversal is topologically equivalent to the tree traversal. Therefore data is fetched from the calling node with callbacks by special nodes *exe*:*externName* or *exe*:*externIndex* within the **Outsource**. Which means that data is not passed to an **Outsource** as arguments in case of a **Routine** call.

Figure 3 of Listing 11 shows a bidirectional **Routine**. **Epilog** is not given since **Rhythm** is disabled, see line 7 of the Listing.

In Figure 3 the green section of nodes {{1,2,4,5}} prints the concatenation of three strings {3}, {5} and {6} that is `"first string second string"`. Node {5} is a whitespace according to line 12 of Listing 11.

In Figure 3 the red section of nodes {{10,11,15,16}} is identical to the green section and stored inside a non-bound **Outsource**. During the deformed traversal Node {8} behaves like a leaf node and execution bounces. The **Outsource** is entered with **Routine** node {20}.

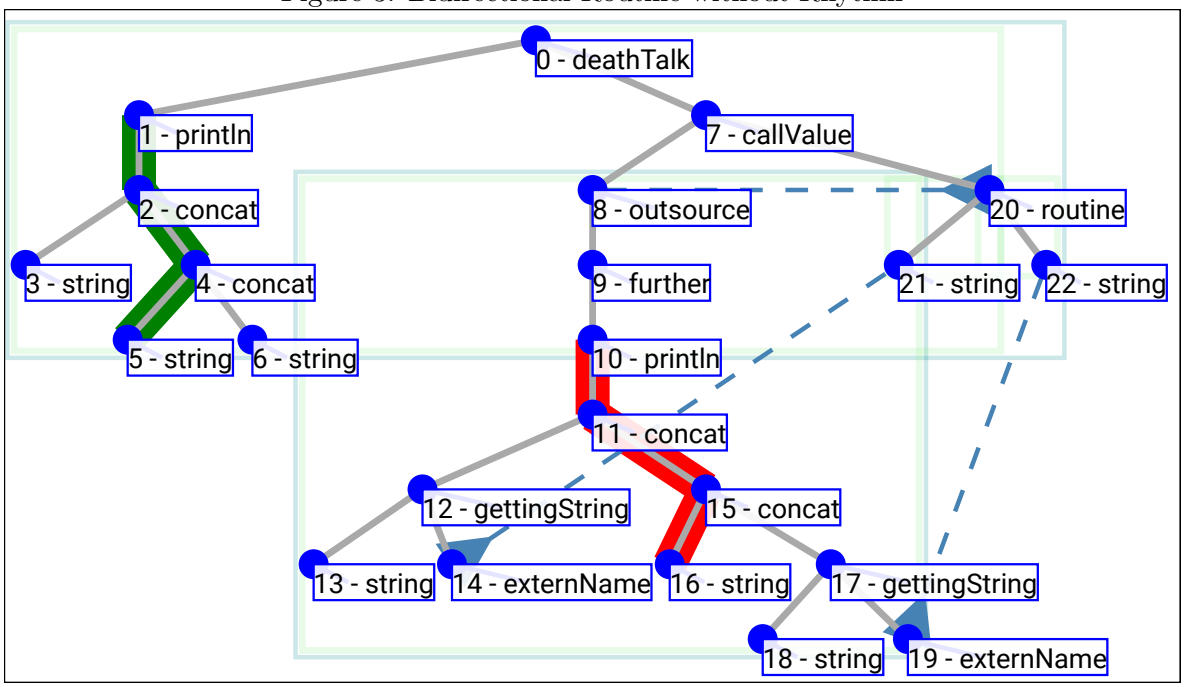
Node {14} *exe*:*externName* takes the parameter string `"first callback"` that was produced by Node {13} and execution jumps to **Insource** {21} which carries the *exe*:*name* of `"first callback"`. Execution returns from the callback immediately since there are no inferior nodes.

Node {19} takes the parameter string `"second callback"` of Node {18} and performs a call back and forth to **Insource** {22}. Execution proceeds and the non-bound **Outsource** returns from Node {8} to **Routine** call {20}. Three jumps shown with dashed lines are required to print the concatenation `"A C"` with this particular **Outsource** through a bidirectional **Routine** call.

Execution stacks are dynamic in DeathTalk and if an inferior **Work** is finished then all data spills over to its direct superior **Work** in runtime. This way string {21} is spilled on the stack of {14} and string {22} is spilled on the stack of {19}. The result of the **Outsource** is empty and would otherwise spill over from the stack of {8} onto the stack of **Routine** Node {20}.

A bidirectional **Routine** call is in any way dynamic and the order as well as the number of callbacks is not required at build time. An instruction graph could even be extended by one callback between call and another callback. A paradigm does not exist for a bidirectional **Routine**.

Figure 3: Bidirectional Routine without Rhythm



Listing 11: Bidirectional Routine without Rhythm

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:cmd="dict://deathtalk/command"
    xmlns:exe="dict://deathtalk/execution"
    xmlns:rhythm="dict://deathtalk/rhythm"
    xmlns:value="dict://deathtalk/value"
    xmlns:contract="dict://deathtalk/contract"
    exe:rhythm="false">
  <cmd:println>
    <value:concat>
      <value:string>first string</value:string>
      <value:concat>
        <value:string> </value:string>
        <value:string>second string</value:string>
      </value:concat>
    </value:concat>
  </cmd:println>
  <rhythm:callValue>
    <value:outsource>
      <contract:further>
        <cmd:println>
          <value:concat>
            <rhythm:gettingString>
              <value:string>first callback</value:string>
              <exe:externName/>
            </rhythm:gettingString>
            <value:concat>
              <value:string> </value:string>
              <rhythm:gettingString>
                <value:string>second callback</value:string>
                <exe:externName/>
              </rhythm:gettingString>
            </value:concat>
          </value:concat>
        </cmd:println>
      </contract:further>
    </value:outsource>
    <value:routine>
      <value:string exe:name="first␣callback">A</value:string>
      <value:string exe:name="second␣callback">C</value:string>
    </value:routine>
  </rhythm:callValue>
</exe:deathTalk>
```

20.3 Bound Outsources

An **Outsource** can be bound like any other **Value** to a **Type**, an **Article** or a **Contract**. A bound **Outsource** is executed with a reference.

Listing 12 is an example of an **Outsource** bound to a newly declared **Type**. Listing 13 is an example of an **Outsource** bound to an **Article**. The applied **Impression** is discussed in Section 15.

Listing 12: Outsource Bound to Type

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:type="dict://deathtalk/type"
    xmlns:rhythm="dict://deathtalk/rhythm"
    xmlns:cmd="dict://deathtalk/command">
  <!-- change to a new working type -->
  <type:change>
    <type:new>
      <value:string>some type name</value:string>
    </type:new>
  </type:change>
  <!-- bind a new outsource to working type -->
  <type:bindOutsource>
    <value:string>name</value:string>
    <value:outsource>
      <cmd:println>
        <value:string>content</value:string>
      </cmd:println>
    </value:outsource>
  </type:bindOutsource>
  <!-- call outsource of working type -->
  <rhythm:unvoid>
    <rhythm:callName>
      <value:string>name</value:string>
      <type:routine/>
    </rhythm:callName>
  </rhythm:unvoid>
  <!-- prints "content" and exits -->
</exe:deathTalk>
```

Listing 13: Outsource Bound to Article

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:art="dict://deathtalk/article"
    xmlns:contract="dict://deathtalk/contract"
    xmlns:rhythm="dict://deathtalk/rhythm">
  <cmd:expression>
    <contract:bind>
      <value:string>object</value:string>
      <art:new/>
    </contract:bind>
  </cmd:expression>
  <cmd:expression>
    <value:articleBindAs>
      <contract:referArticle>
        <value:string>object</value:string>
      </contract:referArticle>
      <value:string>member</value:string>
      <value:outsource>
        <cmd:println>
          <value:string>outsource of article</value:string>
        </cmd:println>
      </value:outsource>
    </value:articleBindAs>
  </cmd:expression>
  <rhythm:unvoid>
    <rhythm:callValue>
      <rhythm:rereferOutsource>
        <contract:referArticle>
          <value:string>object</value:string>
        </contract:referArticle>
        <value:impression>
          <value:articleAt>
            <value:string>member</value:string>
          </value:articleAt>
        </value:impression>
      </rhythm:rereferOutsource>
      <value:routine/>
    </rhythm:callValue>
  </rhythm:unvoid>
  <!-- prints "outsource␣of␣article" and exits -->
</exe:deathTalk>
```

20.4 Continuity

Program sources are usually divided into **Outsource**s that each define the instruction subgraph for a common task and that are called at runtime. In DeathTalk a **Routine** is a bidirectional **Call** of an **Outsource**. A **Routine** can have several **Insource**s that may be called back from an **Outsource** with *exe*:*externName* or *exe*:*externIndex* in any order and as many times as necessary.

Execution is explicitly divided at **Stretch**es or **Routine Calls** and **Extern Callback**s. Additionally execution is implicitly divided at **Stretch**es or **Call**s to **Operator**s. A **Stretch** contains all nodes between two **Jump**s nodes and is executed in both directions.

In source code **Routine** and **Insource**s are always directly related. Execution deviates from this order which requires the implementation of two continuous mappings. Firstly, a mapping of **Insource**s to their superior **Routine**. Secondly, a mapping of the **Contract** to an **Insource** from which **Routine** was executed.

Both continuity mappings are discussed below with references to Figure 4 which shows the division into stretches of a program that makes a detour just to print the word `"here"`. In that figure a thick edge represents a **Call Stretch** of which the label has three implications. Firstly, the label is the running number of a **Stretch**. Secondly, the label is the string that an **Outsource** is looked up with. Thirdly, the label is the string that the **Outsource** is bound with. In Figure 4 a double edge represents a **Callback Stretch** of which the label has three implications. Firstly, the label is again the running number of a **Stretch**. Secondly, the label is the string that an **Insource** is looked up with. Thirdly, the label is the string that an **Insource** is named with. Each edge points away from the initiation node with the arrowhead on the One end of the One-to-Many-relationship.

Figure 4: Stretches are defined between Jump nodes and executed in both directions

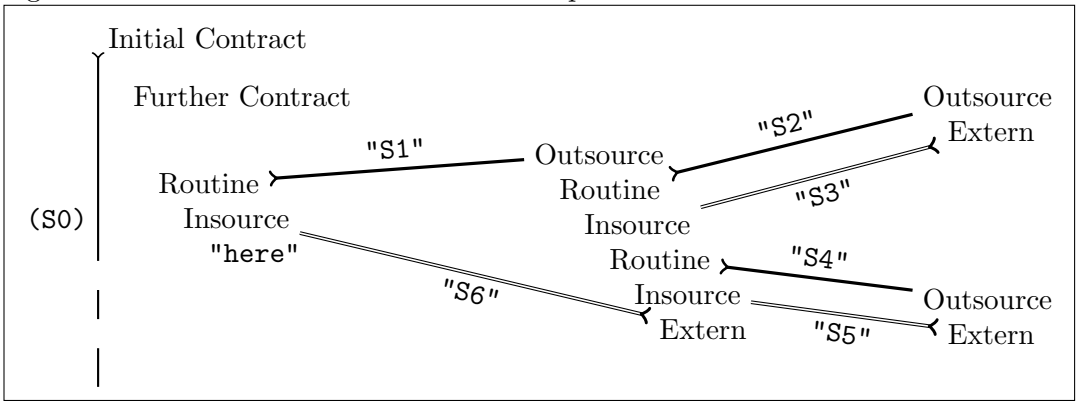


Table 3: Key Relations of Figure 4 according to Section 20.4.1

Stretch	0	1	2	3	4	5	6
Executive Parent	-	0	1	2	3	4	5
Source Parent	-	0	1	1	1	1	0
Self	0	1	2	1	4	1	0

20.4.1 Source Continuity

Three key relations are stored with each **Stretch** in order to maintain source continuity over execution. A central fourth relation is derived from the first three relations. A parent is either the direct superior **Stretch** at runtime or in source code.

1. An own unique number is generated from a sequence.
2. The executive parent **Stretch** by which the **Stretch** was initiated at runtime is stored.
3. The parent **Stretch** according to source code is determined by either of two by two cases.
   (a) If the initiating **Stretch** and therefore the executive parent is a **Routine Stretch**
      i. and if the new **Stretch** is also a **Routine Stretch** then its executive parent is also its source parent.
      ii. and if the new **Stretch** is a **Callback Stretch** then the source parent of the new **Stretch** is the source parent of the executive parent **Stretch**.
   (b) If the initiating **Stretch** is a **Callback Stretch**
      i. and if the new **Stretch** is a **Routine Stretch** then the source parent of the new **Stretch** is the source parent of the executive parent **Stretch**.
      ii. and if the new **Stretch** is also a **Callback Stretch** then the source parent of the new **Stretch** is the source parent of the source parent of the executive **Stretch**. Which is the only case that cannot be derived from the executive parent and that therefore makes this bookkeeping necessary.
4. **Self** is a frequent relation at runtime and determined by either of two cases.
   (a) A **Routine Stretch** is its own **Self**.
   (b) The **Self** of a **Callback Stretch** is the **Routine Stretch** that initiated the **Call** thus the source parent.

DeathTalk starts execution with an initial **Stretch** that has no direct superior **Stretch** or parent. The determination of the three key relations for further stretches is given again as a short table.

Determine below for new	from **Routine**	from **Callback**
executive parent	sequential	sequential
source parent for **Routine**	`this`	source parent
source parent for **Callback**	source parent	source parent of source parent

20.4.2 Contract Continuity

The **Contract** hierarchy is discontinuous in execution at a **Callback Jump**. It is this discontinuity that requires the construct of **Stretch**es and in return allows for even more concurrency through **Stretch**es. A **Callback Stretch** begins at an **Insource** which is a direct inferior node of a **Routine** node in source code and therefore inherits the **Contract** according to source code. The then required **Contract** is only directly accessible from **Self** in the exceptional case that the **Contract** hierarchy was not furthered between **Self** and **Call**.

In all other cases the required **Contract** can only be found back in execution for which a simple algorithm is available. A first loop goes back over all consecutive **Callback Stretch**es in execution. The required **Contract** is then determined by a second loop that starts at the first **Routine Stretch** back in execution and that goes back in source for as many iterations less one which only covers **Routine Stretch**es. Which touches as many **Routine Stretch**es as **Callback Stretch**es.

The **Contract** required for **Callback Stretch** 6 in the example according to Figure 4 is not directly accessible from **Self** of **Callback Stretch** 6 since the **Contract** hierarchy was furthered between **Self** and **Call**. The required **Contract** is found by going back in execution over (`"S6" - "S5" - "S4"`) and by then going back in source over (`"S4" - "S1"`).

20.5 Operators

An **Operator** is an **Outsource** that is executed implicitly for a unary or binary combination of **Article**s that are each joined with a **Type** and as such is just a combination of some of the already discussed technologies.

Listing 14 is an example of the definition and application of the *value*:*utter* operator for the root **Type** in five steps.

Line 9: A new **Outsource** is associated with the **utter Operator** of the root **Type**.
Line 18: A new **Article** is bound by the name of `"object"` to the initial **Contract**.
Line 23: A **String** is bound by the name of `"value"` to the **Article**.
Line 31: The **Article** is made an instance of the root **Type** with *art*:*join*.
Line 37: The result of *value*:*utter* on the **Article** is printed and gives `"Good Bye World"`.

Only very few binary operators are actually implemented for simplicity of the presentation such as *type*:*joinAddArticleArticle* or *type*:*joinMultiplyRealArticle*.

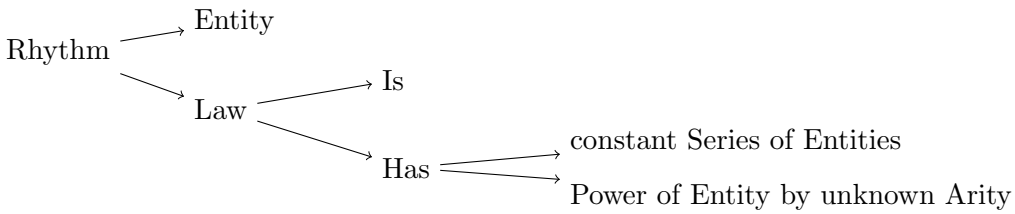
Listing 14: Utter Operator

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    xmlns:type="dict://deathtalk/type"
    xmlns:contract="dict://deathtalk/contract"
    xmlns:art="dict://deathtalk/article"
    exe:rhythm="false">
  <type:joinUtter>
    <value:outsource>
      <value:impression>
        <value:articleAt>
          <value:string>value</value:string>
        </value:articleAt>
      </value:impression>
    </value:outsource>
  </type:joinUtter>
  <cmd:expression>   <contract:bind>
      <value:string>object</value:string>
      <art:new/>
    </contract:bind>
  </cmd:expression>
  <cmd:expression>  <value:articleBindAs>
      <contract:referArticle>
        <value:string>object</value:string>
      </contract:referArticle>
      <value:string>value</value:string>
      <value:string>Good Bye World!</value:string>
    </value:articleBindAs>
  </cmd:expression>
  <art:join>
    <contract:referArticle>
      <value:string>object</value:string>
    </contract:referArticle>
    <type:current/>
  </art:join>
  <cmd:println>
    <value:utter>
      <contract:referArticle>
        <value:string>object</value:string>
      </contract:referArticle>
    </value:utter>
  </cmd:println>
</exe:deathTalk>
```

21 Rhythm

Rhythm is a builtin schema for the validation of a DeathTalk source code. An XML schema is not derived since DeathTalk is not at a productive stage.

Rhythm is a normal grammar. Their are no further abstractions. **Rhythm** is aimed at an alternate succession of operands and operations. **Rhythm** improves readability, allows for generation of **Epilog** and keeps data close to their target instruction. A machine has no **Rhythm** and does not understand sense.



The two fundamental entities of **Rhythm** are *Part* and *Value*. Everything instructive is a *Part* such as a **Command**, a **Control** or an **Operation**. Everything combinable that eventually turns into a **Value** is *Value*.

Rhythm has three laws that define any programming language. All it takes is proper labels for *Entities*.

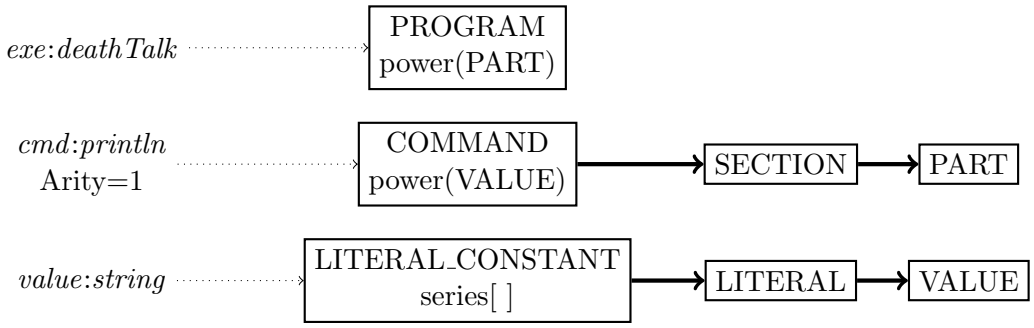
1. The Is-Law concretes.
2. The two Has-Laws define direct inferior *Entities*.
   2a. The Series-Law states the exactly required inferior Entities.
   2b. The Power-Law states that an Entity has an Arity number of a specific Entity.

21.1 Entities of the Obligatory Greeting

The obligatory greeting has three entities as shown in Figure 5.

1. *exe*:*deathTalk* is a *PROGRAM*. An *exe*:*deathTalk* has an infinite number of *PART* entities. Thus arity is infinite.
2. *cmd*:*println* is a *COMMAND*. *cmd*:*println* has one *VALUE*. *cmd*:*println* calls toString internally for simplicity and therefore it is not necessary to pass a string. A *COMMAND* is a *SECTION*. A *SECTION* is a *PART* of the same file.
3. *value*:*string* is a *LITERAL_CONSTANT* that has no inferior *Entities*.
   A *LITERAL_CONSTANT* is a *LITERAL* and *LITERAL* is a *VALUE*.

Figure 5: Entities of the obligatory greeting.



Listing 15: A machine has no rhythm.

```
<?xml version="1.0" encoding="UTF-8"?>
<exe:deathTalk xmlns:exe="dict://deathtalk/execution"
    xmlns:value="dict://deathtalk/value"
    xmlns:cmd="dict://deathtalk/command"
    exe:rhythm="false">
  <value:string>this</value:string>
  <value:string>makes</value:string>
  <value:string>no</value:string>
  <value:string>sense</value:string>
  <value:concat/><value:concat/><value:concat/>
  <cmd:println/>
</exe:deathTalk>
```

22 Internal Iterator with Rhythm

A bidirectional **Routine** allows for a normal internal iterator. A complete example with **Rhythm** is given in this Section. The iterator is normal since it only iterates. The calling **Routine** is called back for each iteration and executes logic that is not exposed to the iterator with some other **Routine** as necessary with abnormal logic.

The root element loads all necessary **Part**s of the **MachineDictionary** and flags that the name of the document is put out to console at the beginning of execution which is quite useful with several documents open.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <exe:deathTalk name="Internal␣Iteration" debug:greet="true"
3      xmlns:debug="dict://deathtalk/debug"
4      xmlns:exe="dict://deathtalk/execution"
5      xmlns:value="dict://deathtalk/value"
6      xmlns:type="dict://deathtalk/type"
7      xmlns:cmd="dict://deathtalk/command"
8      xmlns:contract="dict://deathtalk/contract"
9      xmlns:rhythm="dict://deathtalk/rhythm"
10     xmlns:a="resource://deathtalk/annotation">
```

The iterator is supposed to be some reusable outsource that may be bound to some library. It is bound to the root type here.

```
11   <a:comment>internal iterator</a:comment>
12   <type:bindOutsource>
13     <value:string>iterator</value:string>
14     <value:outsource>
```

The outsource runs in a **Contract** that must not be privatized since the counter is bound to the shared contract. The contract may be protected, though, such that data is only shared between calling **Routine** and this iterator.

```
15       <cmd:expression>
16         <contract:bind>
17           <value:string>counter</value:string>
18           <value:int>0</value:int>
19         </contract:bind>
20       </cmd:expression>
```

The iterator executes a simple loop from 1 to 3 here and may be as complex as necessary for a proper application.

```
21       <!-- a simple loop -->
22       <exe:whileDo>
23         <value:smallerThan>
```

```
24            <contract:referInt>
25              <value:string>counter</value:string>
26            </contract:referInt>
27            <value:int>3</value:int>
28          </value:smallerThan>
```

The loop body only calls back the insource by the name of handler of the calling **Routine** and increments the counter. The name of the insource is passed as **Parameter**. Readability is improved with **Rhythm**. *rhythm*:*unvoid* masks a **Void** as section and *rhythm*:*externName* pulls name and callback together such that **Epilog** gives sense.

```
29          <exe:superior>
30            <!-- callback caller -->
31            <rhythm:unvoid>
32              <rhythm:externName>
33                <value:string>handler</value:string>
34                <exe:externName/>
35              </rhythm:externName>
36            </rhythm:unvoid>
37            <!-- increment counter -->
38            <cmd:expression>
39              <value:increment>
40                <contract:referInt>
41                  <value:string>counter</value:string>
42                </contract:referInt>
43              </value:increment>
44            </cmd:expression>
45          </exe:superior>
46        </exe:whileDo>
47      </value:outsource>
48    </type:bindOutsource>
```

The caller initiates a **Contract** with protected access such that data is only visible to the iterator. The insource only prints the value of the counter here. *exe*:*superior* is actually not required. Readability is improve with **Rhythm**. *rhythm*:*unvoid* masks a **Void** as section and *rhythm*:*callName* pulls name and call together such that **Epilog** gives sense.

```
49    <a:comment>contract allows for exchanging data</a:comment>
50    <contract:further contract:access="protected">
51      <rhythm:unvoid>
52        <rhythm:callName>
53          <value:string>iterator</value:string>
54          <type:routine>
55            <exe:superior exe:name="handler">
56              <cmd:println>
57                <contract:referInt>
58                  <value:string>counter</value:string>
```

```
              </contract:referInt>
            </cmd:println>
          </exe:superior>
        </type:routine>
      </rhythm:callName>
    </rhythm:unvoid>
  </contract:further>
</exe:deathTalk>
```

The below **Epilog** is generated along execution into directory debug. **Epilog** may not validate and indentation is not complete.

```
/* internal iterator */
bindOutsource("iterator", outsource
{(
  expression((§ bind "counter" = 0));
  while (§"counter" < 3) do
  {
    unvoid(externName("handler"));
    expression((++§"counter"));
  }
)}
);
/* contract allows for exchanging data */
protected {§
  unvoid(routine "iterator"(( {
    println(  "counter");
  }
) as "handler"));
§}
```