ESDL Multiblock Extension Proposal

Steve Dower*

November 11, 2010

Abstract

Evolutionary Systems Definition Language (ESDL) is a domain-specific language for search algorithms based on iterative improvements to a solution population. The iterating part of an algorithm is described using the notion of a generation block. This proposal extends ESDL to support multiple blocks, allowing algorithms that consist of several significantly different blocks of behaviour with a flexible block switching method.

1 Introduction

Evolutionary System Definition Language (ESDL) is a domain-specific language for describing the flow of evolutionary or population-based algorithms [2]. It defines a system as *groups* of *individuals* and the process used to create new groups from existing ones. Groups are created by selecting and modifying individuals from existing groups or from *generators*.

In ESDL as previously defined in [2], two blocks of code are required: the *initialisation* block and *generation* block. The generation block is contained within `BEGIN GENERATION` and `END GENERATION` statements. Each generation, this block is executed once. Between generations, the primary population is scanned to determine whether to terminate the experiment, and if not, the process repeats. The initialisation block is implied and contains all commands appearing before `BEGIN GENERATION`. At the start of the experiment, this block is executed once to create any groups or variables required by the generation block.

Both the initialisation and generation blocks are fixed sequences – the same source groups, destination groups, filters and selectors must be used each iteration. While parameters may be adjusted to omit certain filters, algorithms that require significantly different flows are not easily supported at present.

For example, the CHC algorithm [3] uses a "catastrophic" clone and mutation operation that is triggered by a certain level of diversity, and hybrid algorithms [1] require the ability to execute one search algorithm multiple times before switching to another, potentially based on iterations, time or some property of the process.

This proposal describes an extension to ESDL to generalise the generation block into multiple named blocks. A separate plan determines which block to execute each iteration. Group and variable scope is clarified to avoid ambiguity when sharing between blocks. Section 2 specifies the proposed extension in the context of changes to ESDL as described in [2]. Section 4 discusses two alternatives, including both supporting arguments and the rationale for their non-selection. Appendix A includes a suggested set of modifications to implement the described functionality in `esec`.[1]

*Contact via http://stevedower.id.au/

[1]The reference implementation of ESDL; available online at http://code.google.com/p/esec/.

2 Multiblock Extension

2.1 Intent

The primary intent of this extension is to support hybrid and meta-algorithms with more than one mode of operation. Algorithm classes[2] and multi-population algorithms[3] are already supported by ESDL and do not require multiple blocks.

This extension is not intended as an abstraction mechanism. Rather, it is a way to combine separate algorithms into a hybrid with a minimum amount of complexity or obscurity added to the definitions.

2.2 BEGIN Statement

The `BEGIN GENERATION` statement is replaced by a `BEGIN <identifier>` statement that defines the start of a named block. The identifier immediately following `BEGIN` is the *name* of the block. A valid block name may contain only letters, numbers and underscores and must begin with a letter. Block names may not be the same as a variable or group name.[4] All names in ESDL are case-insensitive.

2.3 END Statement

As a minor generalisation, the `END` statement closes the most recently started block, including `REPEAT` blocks. The text appearing after `END` is ignored. Including the name of the block is recommended where it improves readability, even though it does not form part of the required syntax.

2.4 Block Selector

Each iteration, a block must be selected to be executed. The actual specification of this selection process is best determined by the underlying implementation, but for consistency between implementations, it must meet the following requirements:

Selection is **immediate**: An algorithm is only required to specify a block for the current iteration. Implementations may support knowledge or hints of future selections as an optimisation, but algorithms cannot be prevented from deciding based on the timeliest results.

Selection is **final**: Once a block has been selected, it must be executed to completion. Algorithms have no way to abort a partially executed block.

Selection is **precise**: Exactly one block must be selected each iteration. The selector may use stochastic methods to decide, but must specify exactly which block will be executed. Multiple blocks cannot be executed within a single iteration; each block will form its own iteration.

Selection is **optional**: If no selector is provided, each block is executed one after the other in the order specified in the definition.

These requirements imply that a selector should be implemented as a user-defined function that is evaluated at the start of each iteration. However, implementations are free to use whatever structure is most appropriate. For example, many programming languages provide an iterator pattern that is suitable for this purpose.

There are no limits on presentation in written works – authors are best positioned to select an appropriate style – though it is intended that selector specifications will be presented separately from ESDL systems and associated through text (as shown in Section 3).

2.5 Rationale

For this extension, a block selector function must be provided externally from the ESDL system definition. This is similar to the way *generators*, *filters*, *selectors* and *evaluators* are currently used

[2] Algorithm classes should be implemented as a separate definition for each specific algorithm.
[3] Multi-population algorithms may be implemented using a single block with multiple groups.
[4] Names in ESDL are global to prevent confusion caused by using the same name for different purposes.

with ESDL: all of these elements are specified separately in a language or form defined by the underlying implementation (or, for publications, the author's preference) [2].

While external elements are typically referenced by name from within the definition, the block selector should not be, using the default evaluator approach as precedent. Specification of the default evaluator is handled by the underlying framework (for example, `esec` uses the value of its "landscape" property). Providing a selector in a similar manner is reasonable, and since the selector itself cannot be changed, though its internal state may, there is no need for an ESDL statement to specify it.

Using a selector defined separately from the definition can significantly simplify the language implementation, by taking advantage of whatever programmatic functionality is available. It allows sequences to be described in forms other than code when such a description is more apt (for example, by using a timeline as shown in Figure 1). The responsibility for ensuring a correctly implemented sequence belongs to the person implementing the algorithm, whereas the author can ensure that the remainder of the ESDL definition is correct prior to publication.

3 Examples

The examples presented in this section have been selected as demonstrations of the multiblock syntax and its suitability for presenting algorithms. These examples are not intended to represent improved or even necessarily useful algorithms.

3.1 Random Switching

This example demonstrates a simple evolutionary algorithm that alternates randomly between mutation and crossover each generation. Listing 1 specifies a system consisting of blocks named `mutate_block` and `crossover_block`. The `mutate_block` block uses tournament selection, Gaussian mutation and only retains individuals that are an improvement over the initial population. The `crossover` block creates a pool of parents from the best half of the population and a fitness-independent random selection.[5] Non-overlapping selection is used to replace the original population.

Listing 2 specifies which block should be run in any generation using a 0.5 probability of selecting either block, independent of any previous selection.

Listing 1: System definition with two blocks in ESDL

```
FROM random_real SELECT (size) population
YIELD population

BEGIN mutate_block
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING mutate_gaussian
    FROM population, offspring SELECT (size) population USING best

    YIELD population
END

BEGIN crossover_block
    FROM population SELECT (size/2) parents1 USING best
    FROM population SELECT (size/2) parents2 USING uniform_random
    FROM parents1, parents2 SELECT parents USING uniform_shuffle
    FROM parents SELECT population USING crossover_uniform

    YIELD population
END
```

[5] This is not actually parent-mate selection; there is no guarantee that a parent selected using `best` will mate with a parent selected at random.

Listing 2: Block selection for Listing 1 in pseudocode

```
if fair_coin_toss() = HEADS
    do 'mutate_block'
else
    do 'crossover_block'
```

3.2 Iteration Count Switching

Listing 3 provides a definition of a system with three different levels of mutation in separate blocks. While parameters such as `per_indiv_rate` and `longest` can be specified with adjustable variables, exchanging `mutate_bitflip` for `mutate_gap_inversion` is not as trivial.[6] Listing 3 separates the three mutation processes and uses the block names to label each as high, moderate or low impact.

Figure 1 provides a graphical timeline indicating when each block should be run and for how long. To run the algorithm, a reader is required to convert the timeline into whatever code is required by the implementation they are using, allowing a greater level of flexibility. For example, listings 4 and 5 show two alternative implementations of Figure 1 that may be used with `esec` (with the modifications specified in Appendix A).

Listing 3: System definition with three blocks in ESDL

```
FROM binary_zero(length=20) SELECT (size) population
YIELD population

BEGIN high_mutation
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_gap_inversion(per_indiv_rate=0.1, longest=10)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
END

BEGIN moderate_mutation
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_gap_inversion(per_indiv_rate=0.1, longest=5)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
END

BEGIN low_mutation
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_bitflip(per_indiv_rate=0.1, per_gene_rate=0.5)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
END
```

[6] For example, this could be done by always applying both mutation types and adjusting the `per_indiv_rate` to zero to bypass one or the other.

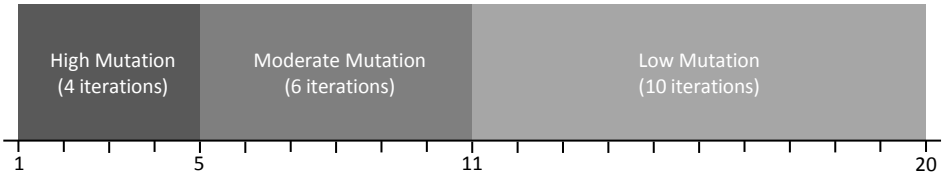

Figure 1: The block sequence for Listing 3 as a timeline

Listing 4: Python iterator implementing Figure 1 for use with `esec`

```
def SelectorIterator():
    for _ in xrange(4):
        yield 'high_mutation'
    for _ in xrange(6):
        yield 'moderate_mutation'
    for _ in xrange(10):
        yield 'low_mutation'

config['selector'] = SelectorIterator()
```

Listing 5: Python list specification of Figure 1 for use with `esec`

```
schedule = ['high_mutation'] * 4 + \
           ['moderate_mutation'] * 6 + \
           ['low_mutation'] * 10

config['selector'] = schedule
```

3.3 CHC Algorithm

The CHC algorithm adapts a number of parameters, including the population size, to maintain a high level of diversity, as well as encouraging reproduction between individuals that are different. Each generation, parents are selected by shuffling (that is, uniform random selection without replacement) and filtered using `no_incest` so that the number of differences between genes in each individual is at least `threshold`. Half-uniform crossover (`crossover_hux`) is then used to select half of the differences between the two individuals and exchange them.

Since `no_incest` performs filtering rather than selection, the `offspring` group may contain fewer individuals than `population`. If the `offspring` group is empty, `threshold` is reduced by one (line 17 of Listing 6). When `threshold` reaches zero, the population is regenerated by reproducing and aggressively mutating the current best individual. This cataclysmic mutation is defined in the `CATACLYSM` block of Listing 6 and is triggered by lines 5 and 6 of Listing 7.

Listing 6: System definition for CHC in ESDL

```
FROM random_binary(length=20) SELECT (size) population USING unique
YIELD population

threshold = 5

BEGIN GENERATION
    FROM population SELECT offspring USING \
         uniform_shuffle, \
         no_incest(threshold=threshold), \
         crossover_hux

    FROM population, offspring SELECT (size) population USING \
         unique, best

    YIELD population, offspring

    threshold = decrement_if_empty(group=offspring, original=threshold)
END GENERATION


BEGIN CATACLYSM
    FROM population SELECT 1 preserve USING best_only

    FROM preserve SELECT (size-1) offspring USING \
         repeated, \
         mutate_bitflip(per_gene_rate=0.35)

    FROM preserve, offspring SELECT population
    YIELD population

    threshold = 5
END CATACLYSM
```

Listing 7: Block selector for CHC (Listing 6) in pseudocode

```
# run 'generation' once at the start
do 'generation'

while still running:
    if the offspring group is empty and threshold is zero
        do 'cataclysm'  # block names are case-insensitive
    else
        do 'generation'
```

3.4 PSO-EO Hybrid Algorithm

Hybrid algorithms are combinations of other, generally complementary, algorithms. Particle Swarm Optimisation (PSO) [4] exhibits good convergence and exploitative characteristics, while Extremal Optimisation (EO; in this case, Continuous Extremal Optimisation as described by [5]) performs a wider exploration of the solution space. Alternating these algorithms on a single solution population provides an efficient balance between concentrating on known good solutions versus exploring other areas.

The initialisation section is given in Listing 8. A block for PSO is shown in Listing 9 and for EO in Listing 10. A simple iteration-count selector is given in Listing 11 as a Python iterator object.

Listing 8: Initialisation definition for PSO-EO algorithm in ESDL

```
FROM random_pso(length=30, lowest=15, highest=30, \
                position_bounds=(-100,100)) SELECT (size) population
FROM population SELECT 1 global_best USING best_only
FROM population SELECT (size) p_bests
YIELD population
```

Listing 9: PSO system definition in ESDL

```
BEGIN pso_generation
    JOIN population, p_bests INTO pairs USING tuples
    FROM pairs SELECT population USING \
                      update_velocity(global_best=global_best), \
                      update_position_clamp

    JOIN population, p_bests INTO pairs USING tuples
    FROM pairs SELECT p_bests USING best_of_tuple

    FROM population, global_best SELECT 1 global_best USING best_only

    YIELD global_best, population
END pso_generation
```

Listing 10: EO system definition in ESDL

```
BEGIN eo_generation
    FROM population SELECT 1 candidate, rest USING \
                rank_proportional(replacement=False, invert=True)
    FROM candidate SELECT replacement USING mutate_random(genes=1)
    FROM replacement, rest SELECT population
    YIELD population
END eo_generation
```

Listing 11: Block selector for listings 8–10 in Python

```
class PSOEOSelector(object):
    def __init__(self):
        self.steps = 0

    def __iter__(self):
        return self

    def next(self):
        self.steps += 1
        if self.steps <= 920:
            return 'pso_generation'
        else:
            if self.steps == 1000: self.steps = 0
            return 'eo_generation'
```

4 Alternate Designs

The two designs presented in this section were considered as alternative ways of supporting multiple blocks in ESDL. Each is presented as a summary of modifications to ESDL, along with the perceived implications.

4.1 BEGIN–IF Statement

For this alternative, the `BEGIN` statement would be replaced with a statement of the form "`BEGIN block_name IF condition`," where `condition` is an expression evaluating to a Boolean *true* or *false*.

Each iteration, the condition for each block is evaluated in the order the blocks are specified in the definition. The first block with an expression evaluating to true is the block executed for that iteration.

Compared to the proposed extension, this alternative would have a great readability advantage in specifying the condition for each block at the same location as the block is defined. However, this would place severe limits on the range of available conditions. For example, ESDL provides no access to the elapsed time, random number generators or complex mathematics. The use of external functions for each condition could mitigate this at the expense of separating the condition from the system definition.

Due to the relevance of block ordering to the behaviour of the algorithm, it would no longer be possible to break a definition into separate listings (as in listings 8–10) without clearly identifying the order they should be recombined. Random selection of a block, even using an external function, would require careful specification to avoid bias in favour of (or against) those blocks declared earlier.

Operations such as switching based on iteration count can clutter the definition, as shown in Listing 12 compared to the similar Listing 3 from Section 3.2. Tight integration between the schedule and the system definition also limits the ease of testing alternate sequences. Abstraction of the selector as in proposed extension provides significantly improved readability.

Listing 12: Switching on iteration count using the alternative `BEGIN–IF` syntax

```
FROM binary_zero(length=20) SELECT (size) population
YIELD population

iteration = 0

BEGIN high_mutation IF iteration < 4
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_gap_inversion(per_indiv_rate=0.1, longest=10)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
    iteration = (iteration + 1) % 20
END

BEGIN moderate_mutation IF iteration < 10
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_gap_inversion(per_indiv_rate=0.1, longest=5)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
    iteration = (iteration + 1) % 20
END

BEGIN low_mutation
    FROM population SELECT (size) parents USING binary_tournament
    FROM parents SELECT offspring USING \
         mutate_bitflip(per_indiv_rate=0.1, per_gene_rate=0.5)
    FROM population, offspring SELECT (size) population USING best

    YIELD population
    iteration = (iteration + 1) % 20
END
```

Finally, allowing Boolean expressions within ESDL would require significant development investment of the language definition, which currently has no support for conditional statements (as discussed

in [2]). Also, with the potential for readability to suffer due to complex conditions, the external selector proposal is considered to support better designs than the `BEGIN-IF` alternative does.

4.2 SCHEDULE Block

For this alternative, the BEGIN and END statements are replaced as in sections 2.2 and 2.3. Rather than using a selector, a new block type would be added, using the keyword `SCHEDULE`. Exactly one `SCHEDULE` block would have to be provided when more than one regular block (not including the initialisation block) was. Within the `SCHEDULE` block, only the `REPEAT` command and a "call" statement would be available.

`REPEAT` would operate identically to those in a regular block. Call statements would be specified by including a block name as the only word on a line.

A `SCHEDULE` block could be executed in one of two ways: either by treating the `SCHEDULE` block as the entire iteration, or by stepping through the `SCHEDULE` block executing one call per iteration. Each approach would result in a different interpretation of "iteration;" for the remainder of this section the latter is assumed.

The `SCHEDULE` block would lend itself to simple specification of switching on iteration count, as shown in Listing 13 (compared to Listing 12 or Listing 3 and Figure 1). Complex iteration such as that in Listing 14 is also clearly defined. Without an `IF` or `WHILE` statement, `SCHEDULE` is limited to switching on iteration counts.

The argument against conditional statements made in Section 4.1 and [2], that is, their omission improves readability, also applies to this alternative. Even though extra syntax could be added solely to the `SCHEDULE` block and remain illegal within regular blocks, the inclusion of extra complexity within ESDL could deter or hinder well-designed abstraction. Abstracting the `SCHEDULE` block into a separate specification results in the proposal made in Section 2; it can be easily implemented in whatever general-purpose programming language is preferred by the underlying framework or published using whatever style or layout is preferred by the author.

Listing 13: Switching on iteration count using the alternative `SCHEDULE` syntax

```
# The remainer of the definition is as defined in Listing 3

SCHEDULE
    REPEAT 4
        high_mutation
    END
    REPEAT 6
        moderate_mutation
    END
    REPEAT 10
        low_mutation
    END
END SCHEDULE
```

Listing 14: Complex switching using the alternative `SCHEDULE` syntax

```
# The remainer of the definition is as defined in Listing 3

SCHEDULE
    high_mutation
    REPEAT 3
        REPEAT 2
            moderate_mutation
        END
        REPEAT 3
            low_mutation
        END
    END
    high_mutation
    REPEAT 3
        low_mutation
    END
END SCHEDULE
```

5 Summary

This proposal describes an extension to ESDL allowing multiple blocks within a system definition. Multiple blocks may be used to implement algorithms with more than one mode of operation, such as hybrid or meta-algorithms.

The proposed extension defined in Section 2 abstracts the block selector and cleanly separates it from the system definition. A number of example uses are presented in Section 3. Two alternative designs that include block selection in the definition are discussed and arguments against are made in Section 4. Both are considered likely to reduce the readability of ESDL system definitions.

Multiple blocks allow ESDL to support a wider range of interesting and novel algorithms, especially those in the field of hybrid and meta-algorithms. The proposal made achieves this without compromising the readability of ESDL or restricting authors from using whatever form of description they believe is most appropriate.

References

[1] K. A. De Jong, *Evolutionary Computation: A Unified Approach.* MIT Press, 2006. 1

[2] S. Dower and C. Woodward, "Evolutionary System Definition Language," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1, 2010. 1, 3, 9

[3] L. Eshelman, "The CHC adaptive search algorithm," in *Foundations of Genetic Algorithms 1*, G. Rawlins, Ed. Morgan Kaufmann Publishers, 1990, pp. 265–283. 1

[4] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4. IEEE, 1995, pp. 1942–1948. 6

[5] T. Zhou, W.-J. Bai, L.-J. Cheng, and B.-H. Wang, "Continuous extremal optimization for Lennard-Jones clusters," *Physical Review E*, vol. 72, 2005. 6

A esec Modifications

Listing 15 provides a complete patch to revision `c5a7a90c78ac` of `esec`, which is also available as the tip of the `multiblock` branch (revision `abde71044fa2`) in the repository at http://code.google.com/p/esec.

The `compiler.py` file is modified to parse block names and emit entry points for each block. `experiment.py` is extended to accept a configuration parameter called `selector`: an iterable sequence (such as a list or generator) containing the names of each block as a string. In `system.py`, a block name parameter is added the `step` method. Finally a "block" format is added to `consolemonitor.py` and `csvmonitor.py` to allow the most recently executed block to be included with the output.

Listing 15: Modifications to `esec` implementing the multiblock extension

```
diff -r c5a7a90c78ac -r abde71044fa2 esec/esec/compiler.py
--- a/esec/esec/compiler.py Mon Jun 28 08:07:01 2010 +1000
+++ b/esec/esec/compiler.py Tue Nov 09 14:34:19 2010 +1100
@@ -84,8 +84,7 @@

         self._groups = None
         self.src_lines = None
-        self.reset = None
-        self.breed = None
+        self.code = None

     def compile(self):
         '''Compiles the source associated with this compiler object. The result
@@ -99,25 +98,16 @@
         exec _BORN_ITER_DEF in self.context #pylint: disable=W0122

         self._groups = set()
+        self.blocks = [ ]
         self.src_lines = list(self._filter_source(self.source_code))
-        code_lines = list(self._transform(self.src_lines))
-        code_blocks = [ [] ]
-        for line in code_lines:
-            if line == None:
-                code_blocks.append([])
-            else:
-                code_blocks[-1].append(line)
+        transformed_lines = list(self._transform(self.src_lines))

-        if len(code_blocks) > 2:
-            raise ESDLSyntaxError('Code after generation definition.', ("ESDL", None, None, None))
-        elif len(code_blocks) == 2:
-            init_code = '\n'.join(('%s = _group()' % g for g in self._groups))
-            self.reset = init_code + '\n' + '\n'.join(code_blocks[0])
-            self.breed = '\n'.join(code_blocks[1])
-        else:
-            self.reset = None
-            self.breed = None
-            raise ESDLSyntaxError('No generation definition included.', ("ESDL", None, None, None))
+        code_lines = [ ]
+        code_lines.extend('%s = _group()' % g for g in self._groups)
+        code_lines.append('')
+        code_lines.extend(transformed_lines)
+
+        self.code = '\n'.join(code_lines)

     @classmethod
     def _hide_nested(cls, src):
@@ -221,12 +211,12 @@
             first_word = parts[0].upper()
             if first_word == 'BEGIN':
                 second_word = parts[2].partition(' ')[0].upper()
-                if second_word == 'GENERATION':
-                    yield None
-                    indent = ''
+                if second_word:
+                    yield indent + "def _block_" + second_word + "():"
+                    self.blocks.append(second_word)
+                    indent += ' ' * 4
                 else:
-                    raise ESDLSyntaxError('Unrecognised parameter to BEGIN: ' + second_word,
-                                          ("ESDL", line_no+1, None, source_line))
+                    raise ESDLSyntaxError('BEGIN requires a block name.', ("ESDL", line_no+1, None, source_line))


             elif first_word == 'END':

diff -r c5a7a90c78ac -r abde71044fa2 esec/esec/experiment.py
--- a/esec/esec/experiment.py Mon Jun 28 08:07:01 2010 +1000
+++ b/esec/esec/experiment.py Tue Nov 09 14:34:19 2010 +1100
@@ -21,6 +21,7 @@
         'monitor': '*', # pre-initialised MonitorBase instance, class or dict
         'landscape': '*',
         'system': '*', # allow System to validate
+        'selector?': '*',
         'verbose': int,
     }
```

```
     '''The expected format of the configuration dictionary passed to `__init__`.
@@ -154,6 +155,10 @@
         self.system = System(cfg, self.lscape)
         self.system.monitor = self.monitor

+        # -- Selector --
+        self.selector = cfg.selector or self.system.blocks
+        self.selector_iter = None
+
         # -- Pass full configuration to monitor --
         self.monitor.notify('Experiment', 'System', self.system)
         self.monitor.notify('Experiment', 'Landscape', self.lscape)
@@ -173,6 +178,7 @@
         '''Start the experiment.
         '''
         self.system.begin()
+        self.selector_iter = iter(self.selector)

     def step(self, ignore_monitor=False):
         '''Executes the next step in the experiment. If the monitor's
@@ -190,11 +196,17 @@
             This value is unaffected by `ignore_monitor`.
         '''

+        try:
+            block = next(self.selector_iter)
+        except StopIteration:
+            self.selector_iter = iter(self.selector)
+            block = next(self.selector_iter)
+
         if self.monitor.should_terminate(self.system):  #pylint: disable=E1103
-            if ignore_monitor: self.system.step()
+            if ignore_monitor: self.system.step(block)
             return False
         else:
-            self.system.step()
+            self.system.step(block)
             return True

     def close(self):

diff -r c5a7a90c78ac -r abde71044fa2 esec/esec/monitors/consolemonitor.py
--- a/esec/esec/monitors/consolemonitor.py Mon Jun 28 08:07:01 2010 +1000
+++ b/esec/esec/monitors/consolemonitor.py Tue Nov 09 14:34:19 2010 +1100
@@ -109,6 +109,8 @@
         # elapsed CPU time
         'time': [' elapsed time  ', "%4d:%02d'%02d.%03d ", '_time'],
         'time_delta': [ ' delta time    ', "%4d:%02d'%02d.%03d ", '_time_delta'],
+        # most recently executed block
+        'block': [ '  block           ', ' %-16s ', '_last_block'],
     }
     '''The set of known column descriptors.

@@ -322,6 +324,7 @@
         self.stop_now = False
         self.end_code = None
         self._stats = None
+        self._last_block_name = 'initialisation'

     class _read_stats(object):  #pylint: disable=C0103,R0903
         '''Read any specified statistic from the primary population's Statistics object
@@ -517,6 +520,17 @@
                     print >> self.config_out, '\n'.join(value.list())
                     print >> self.config_out

+        elif sender == 'System':
+            if name == 'Block':
+                # `value` contains a block name
+                key = value
+                self._last_block_name = key
+                blocks = self._stats['blocks']
+                if key in blocks:
+                    blocks[key] += 1
+                else:
+                    blocks[key] = 1
+
         elif sender == 'Monitor':
             if name == 'Statistics':
                 # `value` contains the _stats dictionary
@@ -563,6 +577,7 @@
             'global_evals': 0,
             'local_evals': 0,
             'groups': set(),
+            'blocks': { },
             self.primary : { 'global_max': EmptyIndividual() }
         }
         self.stop_now = False
@@ -749,3 +764,7 @@
             seconds -= minutes * 60
             minutes -= hours * 60
             return (hours, minutes, seconds, milliseconds)
+
+    def _last_block(self, owner):
+        '''Returns ``(last_block_name,)``.'''
+        return (self._last_block_name,)

diff -r c5a7a90c78ac -r abde71044fa2 esec/esec/monitors/csvmonitor.py
--- a/esec/esec/monitors/csvmonitor.py Mon Jun 28 08:07:01 2010 +1000
+++ b/esec/esec/monitors/csvmonitor.py Tue Nov 09 14:34:19 2010 +1100
```

```
@@ -98,6 +98,8 @@
         # elapsed CPU time
         'time': ['Elapsed time (ms)', '%d', '_time'],
         'time_delta': [ 'Delta time (ms)', '%d', '_time_delta'],
+        # most recently executed block
+        'block': [ 'Block', '%s', '_last_block'],
     }
     '''The set of known column descriptors.

diff -r c5a7a90c78ac -r abde71044fa2 esec/esec/system.py
--- a/esec/esec/system.py Mon Jun 28 08:07:01 2010 +1000
+++ b/esec/esec/system.py Tue Nov 09 14:34:19 2010 +1100
@@ -115,14 +115,13 @@
         # Also expose context
         builtin['context'] = self._context

-        self._reset_code = compiler.reset
-        self._breed_code = compiler.breed
+        self._code_string = compiler.code
+        self.blocks = compiler.blocks

         self.monitor = self._context.get('_monitor', MonitorBase())
         self._context['_on_yield'] = lambda name, group: self.monitor.on_yield(self, name, group)

-        self._reset = compile(self._reset_code, 'ESDL Preamble', 'exec')
-        self._breed = compile(self._breed_code, 'ESDL Generation', 'exec')
+        self._code = compile(self._code_string, 'ESDL Definition', 'exec')

         self._in_step = False
         self._continue_step = False
@@ -157,11 +156,8 @@
             result.append(self.definition.strip(' \t').strip('\n'))
             result.append('')
         if level > 3:
-            result.append('>> Compiled Reset Code:')
-            result.append(self._reset_code)
-            result.append('')
-            result.append('>> Compiled Breed Code:')
-            result.append(self._breed_code)
+            result.append('>> Compiled Code:')
+            result.append(self._code_string)
             result.append('')
         if level > 2:
             result.append('>> ESDL cfg instance:')
@@ -187,7 +183,7 @@
             self.monitor.on_pre_reset(self)

             Individual.reset_birthday()
-            exec self._reset in self._context
+            exec self._code in self._context

             self.monitor.on_post_reset(self)

@@ -206,7 +202,7 @@
             self.monitor.on_run_end(self)
             return

-    def step(self):
+    def step(self, block="generation"):
         '''Executes one generation.
         '''
         # Allowed to use exec
@@ -226,7 +222,8 @@
             try:
                 self.monitor.on_pre_breed(self)

-                exec self._breed in self._context
+                self.monitor.notify('System', 'Block', block)
+                exec ('_block_%s()' % block.upper()) in self._context

             except KeyboardInterrupt:
                 self.monitor.on_run_end(self)
```