ESDL Implicit Parameters Proposal

Steve Dower*

December 15, 2010

Abstract

Evolutionary Systems Definition Language (ESDL) is a domain-specific language for search algorithms based on iterative improvements to a solution population. Operations and selection strategies are specified outside of ESDL and accessed through function calls with named parameters. This proposal amends ESDL to simplify specification of certain parameters in order to produce less verbose system definitions.

1 Introduction

Evolutionary System Definition Language (ESDL) is a domain-specific language for describing the flow of evolutionary or population-based algorithms [1]. 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*. Filtering and modification operators are referred to as *filters*, and are provided by an underlying software implementation. Other *functions* may be specified to provide more complex arithmetic or logical operations than those available in pure ESDL.

In ESDL as previously defined in [1], function and filter parameters may be specified by providing the name of the parameter followed by an equals sign and the value (as shown in Listing 1). The use of the parameter name, unlike other programming languages, is a strict requirement to improve readability in isolation from supporting documentation.

Listing 1: An example of a function and a filter with named parameters in ESDL

```
value = adjust(value=value, rate=1.4)
FROM parents SELECT offspring USING crossover(per_pair_rate=0.2, one_child=true)
```

As can be seen from Listing 1, some redundancies occur when specifying variables. The parameter `value` is passed the variable `value`, resulting in the word appearing twice unnecessarily. The parameter `one_child` is optional, and if omitted (or passed `false`), the behaviour is equivalent to passing `true` to the `two_children` parameter; in effect, `one_child` is only ever specified with the value `true`.

For simple systems, such as those with few variation operations, it is expected that variables will be named identically to the parameters they represent. In this situation, providing both the parameter name and the variable name (which are the same) does not improve readability. In addition, providing one of a set of mutually exclusive mode-selection parameters (such as whether a recombination operator produces one or two children) can be assumed to represent selection of that option; specifying the value `true` does not clarify further.

This proposal describes an amendment to ESDL to improve and simplify parameter value specification. When a value for a parameter is omitted, the value of a variable with an identical name is used. If no such variable exists, the value `true` is assumed. Section 2 specifies the proposed amendment in the context of changes to ESDL as described in [1]. Section 4 discusses three alternatives, including

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

both supporting arguments and the rationale for their non-selection. Appendix A includes a suggested set of modifications to implement the described functionality in `esdlc`.[1]

2 Implicit Parameters

2.1 Intent

The primary intent of this extension is to reduce the amount of unnecessary text in system definitions and to promote a pattern for specifying behaviour selection parameters.

This amendment does not support positional parameters. Specifying a value or variable without a parameter name will result in an error (unless the name happens to match a valid parameter). The rationale for omitting positional parameters is given in Section 4.2.

2.2 Parameter Names

Parameter names are not affected by this amendment. In every case where a parameter is provided, the name must be provided.

2.3 Parameterised Values

Where a parameter name is followed by a comma or a right-parenthesis, the value is *provided implicitly*. Where a parameter name is followed by an equals sign and a variable or expression, the value is *provided explicitly*. A parameter name followed by an equals sign but not then followed by a value, variable or expression is a syntax error.

An implicitly provided parameter attempts to use the value of a variable with the same name. Names in ESDL are case-insensitive [1] and hence there is no requirement for casing to be identical. If such a variable exists anywhere within the system definition, the value of the variable is passed as the value of the parameter. If the variable exists but has not been initialised, it is an uninitialised variable error. If no such variable exists anywhere in the system definition, the value `true`[2] is passed as the value of the parameter.

If neither a parameter name nor value are provided, the default value of the parameter is used (this is the behaviour as of [1]). The value of a matching variable is only used if the parameter name is specified.

2.4 Behaviour Selection

The behaviour selection pattern is not enforced by ESDL, however, for readability and consistency between implementations it is strongly encouraged for function and filter developers.

Many selection and variation operations have some parts of their behaviour parameterised. For example, most two-parent crossover operations have the option of producing either one or two children and some operations that use value limits may use hard limiting, soft limiting, reflection, wrapping or another mode of restriction. In these cases, the mode is a parameter of the operation, rather than a distinct operator.

A function or filter using the behaviour selection pattern provides one Boolean parameter for each possible mode. Each of these *mode parameters* is optional and defaults to `false`. When all mode parameters are `false`, a default mode is used. When one mode parameter is `true`, the mode associated with that parameter is used. If more than one mode parameter is `true`, either a priority order is used to select from those specified or an error is produced. The only condition where a mode specified as `false` may be used is when it is the default and all mode parameters have been specified as `false`; otherwise, a mode may only be used if it is one of the mode parameters specified as `true`.

[1] The reference compiler for ESDL; available online at `http://code.google.com/p/esec/` as part of the `esec` framework.

[2] Or the equivalent used by the underlying implementation.

2.5 Rationale

A common use of variables in ESDL systems is to specify parameters externally. For example, `esec`'s batch functionality allows multiple experiments to be conducted with different parameters using a single system definition. These variables can be clearly specified using the same name as that of the parameter they represent, as shown in Listing 2. Since the names match, with the proposed amendment the usage of these variables could be as shown in Listing 3.

Listing 2: Example of specifying parameters using variables in ESDL (old syntax)

```
per_gene_rate = 0.2
sigma = 0.85
FROM parents SELECT offspring USING mutate_gaussian(per_gene_rate=per_gene_rate, sigma=sigma)
```

Listing 3: Example of specifying parameters using variables in ESDL (new syntax)

```
per_gene_rate = 0.2
sigma = 0.85
FROM parents SELECT offspring USING mutate_gaussian(per_gene_rate, sigma)
```

The behaviour selection pattern described in Section 2.4 provides a consistent manner in which to adjust the behaviour of an operator. With parameters using the value `true` if no value is provided, specifying the mode name is sufficient, as shown in Listing 4.

Listing 4: Example of specifying an operator mode in ESDL (new syntax)

```
FROM population SELECT 10 parents USING uniform_random(no_replacement)
```

If a variable exists with the same name as the mode, which would result in the value of the variable being passed instead of `true`, the previous style of syntax may be used to override, as shown in Listing 5.

Listing 5: Example of specifying an operator mode in ESDL with variable name conflict

```
FROM parents SELECT 1 one_child USING uniform_random
FROM parents SELECT offspring USING crossover(one_child=true)
```

In both cases, the previous behaviour remains, ensuring backwards compatibility and the ability to override the new behaviour when required. There is the potential for users new to ESDL to confuse implicit parameter syntax with positional parameter syntax as used by other programming languages, however, this is easily clarified and is not likely to be a serious impediment to adoption of ESDL.

3 Examples

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

3.1 External Parameters

This example provides a highly generalised evolutionary algorithm in Listing 6. The parameters `size`, `length`, `k`, `points`, `per_pair_rate` and `per_gene_rate` are provided externally in Listing 7 as a configuration dictionary suitable for use with the `esec` framework. Without implicit parameters, every parameter in Listing 6 (except `size`) would need to be specified as "`per_pair_rate=per_pair_rate`," greatly increasing the amount of text and not improving the readability of the system.

Listing 6: System definition with externally provided parameters in ESDL

```
FROM random_binary(length) SELECT (size) population
YIELD population

BEGIN generation
    FROM population SELECT (size) parents USING tournament(k)
    FROM parents SELECT offspring USING crossover(points, per_pair_rate), \
                                        mutate_bitflip(per_gene_rate)
    FROM offspring SELECT population

    YIELD population
END
```

Listing 7: `esec` configuration dictionary for Listing 6 (in Python)

```
config = {
    'system': {
        'definition': DEFINITION, # see Listing 6
        'size': 100,
        # random binary
        'length': 30,
        # tournament
        'k': 2,
        # crossover
        'points': 1,
        'per_pair_rate': 0.8,
        # mutation
        'per_gene_rate': 1.0 / 30.0
    }
}
```

3.2 Particle Swarm Optimisation Mode Selection

Listing 8 provides a definition for a Particle Swarm Optimisation algorithm based on the one used in [2]. The `update_position` variation operator used on line 13 has a number of methods of limiting the resultant positions. A complete implementation of `update_position` is given in Listing 9 (a summarised implementation was provided in [2]). No limit on particle position (`unbounded`) is the default; `clamp` (hard limit and reset velocity), `wrap` (move to the opposite end of the valid range) and `bounce` (reflected position and negated velocity) are the alternative options and are prioritised in that order.

Since `update_position` correctly implements the behaviour selection pattern (as described in Section 2.4) the behaviour may be specified by simply including the name as a parameter with no value.

Listing 8: ESDL definition of Particle Swarm Optimisation (based on [2])

```
FROM random_pso(length=10, lowest=-100, highest=100) \
        SELECT (size) population

FROM population SELECT 1 global_best USING best
FROM population SELECT (size) p_bests
YIELD population

BEGIN GENERATION
    JOIN population, p_bests INTO pairs USING tuples

    FROM pairs SELECT population USING \
         update_velocity(global_best, w=1.0, c1=2.0, c2=2.0), \
         update_position(unbounded)

    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

    YIELD global_best, population
END GENERATION
```

Listing 9: Python implementation of `update_position`

```
def update_position(_source, unbounded=False, clamp=False, wrap=False, bounce=False):
    # unbounded is the default mode
    if not clamp and not wrap and not bounce:
        unbounded = True

    for indiv in _source:
        new_position = list(indiv)
        new_velocity = list(indiv.velocities)
        lower = indiv.lower_position_bounds
        upper = indiv.upper_position_bounds

        for i, (pos, vel, low, high) in \
            enumerate(zip(new_position, new_velocity, lower, upper)):

            new_pos = pos + vel
            new_vel = vel

            if unbounded:
                pass
            elif clamp:
                if new_pos < low:
                    new_pos = low
                    new_vel = 0
                if new_pos > high:
                    new_pos = high
                    new_vel = 0
            elif wrap:
                if new_pos < low:
                    new_pos = high ? (low ? new_pos)
                if new_pos > high:
                    new_pos = low ? (high ? new_pos)
            elif bounce:
                if new_pos < low:
                    new_pos = low + (low ? new_pos)
                    new_vel = -new_vel
                if new_pos > high:
                    new_pos = high + (high ? new_pos)
                    new_vel = -new_vel

            new_position[i] = new_pos
            new_velocity[i] = new_vel

        yield PSOIndividual(new_position + new_velocity, indiv)
```

3.3 Parameterised Mode Selection

Combining implicit parameters with the behaviour selection pattern allows behaviours to be neatly, if obscurely, overridden. This example demonstrates a potentially useful side effect of implicit parameters, rather than recommended practice for presenting an algorithm.

Line 7 of the algorithm in Listing 10 specifies uniform crossover apparently producing both one child and two children. However, the `esec`-style configuration dictionary specified in Listing 11 creates variables named `one_child` and `two_children`. Since the variables are now defined, their values are used instead of `true`, allowing the mode used by the crossover operation to be selected externally to the system definition.

Listing 10: System definition for a Genetic Algorithm in ESDL

```
1 FROM random_binary(length) SELECT (size) population
2 YIELD population
3
4 BEGIN GENERATION
5     FROM population SELECT (size) offspring USING \
6          fitness_proportional,
7          uniform_crossover(one_child, two_children),
8          mutate_bitflip(per_gene_rate)
9
10    FROM offspring SELECT population
11
12    YIELD population
13 END GENERATION
```

Listing 11: `esec` configuration dictionary for Listing 10 (in Python)

```
config = {
    'system': {
        'definition': DEFINITION, # see Listing 10
        'size': 100,
        # random binary
        'length': 30,
        # crossover
        'one_child': True,
        'two_children': False,
        # mutation
        'per_gene_rate': 1.0 / 30.0
    }
```

4 Alternate Designs

The three designs presented in this section were considered as alternatives to the proposed amendment. Each is presented as a summary of modifications to ESDL, along with the perceived implications.

4.1 No Modification

No functionality added by the proposed amendment is currently unavailable, so a valid option is to retain the current specification. The current behaviour is that parameters always require both a name and a value, separated by an equals sign. Specifying either a name or a value, but not both, is always an error.

The increased amount of text required for named parameters may result in ESDL authors selecting shortened or abbreviated variables, for example, as shown in Listing 12. Abbreviations of this type usually reduce readability. Encouraging the use of full variable names improves the clarity of parameter specifications.

Listing 12: Examples of abbreviated variable names in ESDL

```
l = 20
pgr = 0.1
FROM random_binary(length=l) SELECT 10 parents
FROM parents SELECT offspring USING mutate_random(per_gene_rate=pgr)
```

Secondly, there is not yet any accepted pattern for behaviour specification. While it is possible to implement the pattern described in Section 2.4, there is no reason to select it over an alternative, such as those shown in Listing 13. By treating implicit parameters as `true` when otherwise undefined, the pattern described earlier has expressive and implementation benefits over the alternatives.

Listing 13: Alternative methods of specifying operator behaviour in ESDL

```
FROM particles SELECT particles USING update_position(mode=1)
FROM particles SELECT particles USING update_position(mode='clamp')
FROM particles SELECT particles USING update_position_wrap
FROM particles SELECT particles USING update_position, wrap_positions
```

4.2 Positional Parameters

For this alternative, values specified in parameter lists are assumed to be values passed to the parameter occupying the associated position. This behaviour matches that used in or supported by most programming languages.

The primary issue with positional parameters is the difficulty in identifying the purpose of each value. For example, Listing 14 does not clearly specify whether $\sigma = 0.1$ or 0.5; clarification must be sought in the documentation or source code of `mutate_gaussian`. When parameter names are required, as in listings 2 and 3, there is no ambiguity as to which parameter receives each value.

Listing 14: Positional parameters using an alternative ESDL syntax

```
FROM parents SELECT offspring USING mutate_gaussian(0.1, 0.5)
```

In other programming languages, the use of positional parameters is supported by development environments that provide parameter names as the developer is typing. Since ESDL is intended for print publication, this feature is not available. The use of positional parameters is largely historical and most modern languages provide and encourage the use of named parameters.

4.3 Partial Implementation

The proposed amendment contains two parts, either of which can be implemented independently of the other. For example, an error could be produced when implicitly specifying a variable that does not exist, rather than assuming the value `true`. This may reduce unexpected behaviour as the result of typographical errors, though such an error is likely to be detected since the name is required to match a valid parameter for the function or filter. Implementation of implicit parameters is simpler without requiring the compiler to detect whether or not a variable exists, however, a production quality compiler should detect this situation and be capable of substituting the value without difficulty.

Alternatively, all parameters specified without a value could assume the intended value is `true`. Implementation of this alternative is trivial, however it does nothing to encourage good naming habits and discourage abbreviations such as those shown in Listing 12.

5 Summary

This proposal describes an amendment to ESDL allowing implicit parameters in function and filter specifications within a system definition. Parameter names provided without values use the value of a variable with a matching name or, if no such variable exists, `true`.

The proposed amendment specified in Section 2 identifies the behavioural changes to parameter specifications, including error conditions. A number of example uses are presented in Section 3. Three alternatives to the amendment proposed are discussed and arguments against are made in Section 4. None are considered likely to improve the readability of ESDL system definitions when compared to the proposal.

The behaviour specification pattern allows developers of external functions and operators to provide consistent interfaces that are easy to use when coupled with implicit parameters. The proposal made achieves this without restricting authors from using whatever form of description they believe is most appropriate.

References

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

[2] ——, "Specifying Particle Swarm Optimisation with ESDL," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/5, 2010. 4, 5

A esdlc Modifications

Listing 15 provides a complete patch to revision `ecd8e91fc6c0` of `esdlc`, which is also available as the tip of the `implicit_parameters` branch (revision `fb09d2539645`) in the repository at http://code.google.com/p/esec.

The `nodes.py` file is modified to add a new property to variable nodes to indicate whether it is a potentially unresolved implicit parameter. `esec.py` is modified to emit dynamic code to use the variable if available or to substitute `True` if it is not. Since `esec` provides some variable values after compilation, it is not possible to determine at compile-time which parameters should be replaced by `True`.

Listing 15: Modifications to `esdlc` implementing implicit parameters

```
diff -r ecd8e91fc6c0 -r fb09d2539645 esec/esdlc/esec.py
--- a/esec/esdlc/esec.py Fri Dec 10 09:21:01 2010 +1100
+++ b/esec/esdlc/esec.py Mon Dec 13 12:02:28 2010 +1100
@@ -155,7 +155,10 @@

     def write_variable(self, node):
         '''Emits text for variable nodes.'''
-        yield self.safe_variable(node.name)
+        if node.implicit:
+            yield 'globals().get("%s", True)' % self.safe_variable(node.name)
+        else:
+            yield self.safe_variable(node.name)

     def write_unknown(self, node):
         '''Emits text for unknown nodes.'''


diff -r ecd8e91fc6c0 -r fb09d2539645 esec/esdlc/nodes.py
--- a/esec/esdlc/nodes.py Fri Dec 10 09:21:01 2010 +1100
+++ b/esec/esdlc/nodes.py Mon Dec 13 12:02:28 2010 +1100
@@ -447,11 +447,14 @@

             if not token: raise error.ExpectedParameterValueError(tokens[-1])
             if token.tag == 'eos': raise error.ExpectedParameterValueError(token)
-            if token.tag != '=': raise error.ExpectedParameterValueError(token)

-            token_i += 1
-            token_i, arg_node = UnknownNode.parse(tokens, token_i)
-            func_args[arg_name] = arg_node
+            if token.tag == '=':
+                token_i += 1
+                token_i, arg_node = UnknownNode.parse(tokens, token_i)
+                func_args[arg_name] = arg_node
+            else:
+                func_args[arg_name] = VariableNode(arg_name, [tokens[token_i-1]])
+                func_args[arg_name].implicit = True

             token = _get_token(tokens, token_i)
             if token and token.tag == ',': token_i += 1
@@ -495,7 +498,7 @@

         token_i += 1
         return token_i, FunctionNode(func_name, tokens[first_token:token_i], *func_args)
-
+
 class NameNode(NodeBase):
     '''Represents a node with a name.'''
     tag = 'name'
@@ -566,6 +569,10 @@
     tag = 'variable'
     def __init__(self, name, tokens):
         super(VariableNode, self).__init__(name, tokens)
+        self.implicit = False
+        '''If ``True``, replace with `ValueNode` rather than raising a
+        warning if the variable does not exist.
+        '''

     @classmethod
     def parse(cls, tokens, first_token):
```