# DEVELOPMENT OF DOMAIN SPECIFIC LANGUAGES AND IMPLEMENTATION OF GENETIC PROGRAMMING IN GENETICA

## *Introduction*

A general and effective approach to domain specific problem solving is to describe a problem, as well as the available knowledge on the problem solving context, in a high level language specialized on the problem's domain. High level formal languages can be expressed in lower level ones if the latter languages satisfy a minimum of expressive capabilities. A typical example is given by the fact that any computer language can be expressed in the machine language. GENETICA, which has the expressive capabilities required, supports the development of high level formal languages. The benefit is that GENETICA's evolutionary computational system becomes available in the latter languages. Genetic Programming (GP) versions of high level languages become possible if programs in the latter languages are treated as evolving data structures in GENETICA.

Section 1 presents a general method for developing in GENETICA high level domain specific languages, including GP ones. Section 2 shows how the language G-CAD (presented in the file *G-CAD_Documentation.PDF*) has been developed in GENETICA.

## 1. A general method for developing domain specific and GP languages in GENETICA

### *1.1 Specifications for the language to be developed and introduction to the development method*

This section presents a general method for developing in GENETICA a language, name it **L**, having the following properties:

- **L** includes all the GENETICA's terms, while **L**-specific terms can be constructed in GENETICA as nested lists.
- **L** atomic formulae can be constructed in GENETICA, while non atomic formulae definitions in **L** can be constructed through the **L** syntax.
- **L** includes high order formulae respecting the definitions of the GENETICA formulae **call** and **at** (see *GENETICA_Documentation.PDF*, § 2.2.2).
- A formula definition in **L** is a list having the form **($F_N$, con, $S_{inp}$, $Ref_{lst}$, $S_{out}$, ($T_{inp}$, $T_{out}$))** where:
  - $\mathbf{F_N}$ is the name of the formula
  - **con** is a connective
  - $\mathbf{S_{inp}}$ is the input section as a list of names of input terms
  - $\mathbf{Ref_{lst}}$ is the reference section as a list of formula references. A formula reference is a list having the form **($Ref_N$, $Ref_{Sin}$, $Ref_{Sout}$)** where $\mathbf{Ref_N}$ is the name of the referenced formula, while $\mathbf{Ref_{Sin}}$ is the input section and $\mathbf{Ref_{Sout}}$ is the output section of the reference as lists of the names of the input and the output terms respectively.
  - $\mathbf{S_{out}}$ is the output section as a list of names of output terms
  - **($T_{inp}$, $T_{out}$)** is the type section where $\mathbf{T_{inp}}$ and $\mathbf{T_{out}}$ are vectors of abstract data structures indicating the type of each term represented in the input and the output section of the definition respectively. Such data structures have been used in GP methods, e.g. [1] ("generic data types") or [2] ("patterns").

  Any symbol in the input section of a formula reference should be included either in $\mathbf{S_{inp}}$ or in the output section of a previous formula reference in $\mathbf{Ref_{lst}}$, while any symbol in $\mathbf{S_{out}}$ should be included in the output section of a formula reference in $\mathbf{Ref_{lst}}$.
- The connectives in **L** include at least the GENETICA's connectives, i.e. the logical operations, the quantifiers, the conditional recursion and the optimization connective, while any other connective can be constructed as a GENETICA hi order formula. A quantifier in **L** is implemented as a non atomic **L** formula whose definition includes a single reference.
- Given the output values of each formula referenced by a non atomic formula **F**, the output values of **F** are deterministically defined.
- A program in **L** is a list of formulae definitions.

Having these properties, **L** is expressive enough for general problem formulation, while it can be specialized in arbitrarily narrow domains, depending on the specific definition of the terms, the atomic formulae and the connectives in **L**.

Let the prefixes "G" and "L" denote an element of GENETICA and an element of **L** respectively.

**L** can be implemented as a G-program, name it $\mathbf{P_G}$, where the input vector for the root G-formula in $\mathbf{P_G}$ (the term "root formula" is defined in the file *GENETICA_Documentation.PDF*, § 2.3, p.8) includes an arbitrary L-program, name it $\mathbf{P_L}$, and an input vector value for the root L-formula in $\mathbf{P_L}$, while the output vector value for the latter formula, resulting from the execution of $\mathbf{P_L}$, is included in the output of the root G-formula. The execution of $\mathbf{P_L}$ reflects the execution of $\mathbf{P_G}$: any data generation scenario in $\mathbf{P_L}$ is an interpretation of a corresponding data generation scenario in $\mathbf{P_G}$, while any confirmation or optimization goal in $\mathbf{P_L}$ is an interpretation of a corresponding goal in $\mathbf{P_G}$. Due to these interpretations, the evolution of data generation scenarios in $\mathbf{P_G}$, supported by GENETICA's computational system, is reflected in $\mathbf{P_L}$.

$\mathbf{P_L}$ can be modified or even developed during the execution of $\mathbf{P_G}$, since it is treated as a data structure in $\mathbf{P_G}$. Specifically, missing L-formulae (i.e. L-formulae referenced but not defined in $\mathbf{P_L}$) can be constructed at run-time, while redundant L-formulae (i.e. L-formulae defined but not referenced in $\mathbf{P_L}$) can be referenced by L-formulae emerging at run-time. Data processing in $\mathbf{P_G}$, resulting to the development of $\mathbf{P_L}$, is controlled by the computational system of GENETICA's environment. This makes possible a general GP method, where domain specific applications could be developed, since problem specific knowledge can be encoded in incomplete L-programs (i.e. L-programs with missing or redundant L-formulae) while such programs, including the program architecture, can be fully developed by evolution. Depending on the degree of the completeness of the L-program, **L** could range from a GENETICA-like (non-GP) language, where the code is fixed whereas the data structures evolve, to a fully GP language where both the entire code and the data structures evolve. Note that, unlike standard GP, GENETICA can cope with confirmation goals. Within GENETICA-based GP such goals can be reflected not only in the problem formulation but also in the formulation of the GP method itself. For instance, type compatibility, which is typically a part of a GP method, can be implicitly converted to a part of the problem to be solved; as a consequence it can be evolved as part of the solution.

### *1.2 The outline of a GENETICA program that implements domain specific and GP languages*

A semi-formal description of the G-program $\mathbf{P_G}$, introduced in the previous paragraph, is presented in Table 1. The notation used does not respect GENETICA's syntax, for the sake of simplicity. Note that the **if-then-else** and the **do-until** structures appearing in the description, although not included in GENETICA's syntax, can be constructed in GENETICA as high order formulae. The basic formulae of the program $\mathbf{P_G}$ are presented in the next paragraphs, with respect to Table 1.

```
1   Execute (P_L, F_N, V_inp)
2       if (Atomic (F_N)) then {
3           if (First_Order (F_N)) then {
4               Call (F_N, V_inp) → V_out
5           } else {
6               Prepare_Forced_Call (V_inp) → (F_F, V_inp_F)
7               Execute (P_L, F_F, V_inp_F) → (P_L, F_F, V_out)
8           }
9       } else {
10          if (F_N = ( )) then {
11              Initialize_Definition (V_inp) → (F_N, con, S_inp, Ref_lst)
12              Add (P_L, (F_N, con, S_inp, Ref_lst, ( ), ( ))) → P_L
13          }
14          Execute_References (P_L, F_N, V_inp) → (P_L, V_out_lst)
15          Define_Output (P_L, F_N, V_out_lst) → (P_L, V_out)
16          Define_IO_Type (P_L, F_N) → P_L
17      }
18      Return (P_L, F_N, V_out)


19  Execute_References (P_L, F_N, V_inp)
20      Initialize_Table (P_L, F_N, V_inp) → T
21      V_out_lst ← ( )
22      i ← 0
23      do {
24          i +1 → i
25          Get_Reference (P_L, F_N, i) → (R, n)
26          Prepare_Execute_Call (P_L, F_N, R, T, i) → (F_N_R, S_inp_R, V_inp_R)
27          Execute (P_L, F_N_R, V_inp_R) → (P_L, F_N_R, V_out_R)
28          Add (V_out_lst, V_out_R) → V_out_lst
29          if (R = ( )) then {
30              Create_Reference_Output_Section (V_out_R) → S_out_R
31              Revise_Reference_ Section (P_L, F_N, n, F_N_R, S_inp_R, S_out_R) → P_L
32          }
33          Revise_Table (P_L, F_N, T, n, V_out_R) → T
34      } until (Termination_condition (P_L, F_N, T, i))
35      Return (P_L, V_out_lst)

36      Definitions of atomic L-formulae follow …
```

TABLE 1

Outline of the GENETICA program $\mathbf{P_G}$ which implements a domain specific—potentially GP—language **L**. The symbol "→" denotes assignment of the values calculated from the left-hand expression to the right-hand variables. The symbol "←" denotes assignment of the right-hand value to the left-hand variable.

a) The G-formula “Execute”

The input for the root G-formula **Execute** (line 1) includes a L-program $\mathbf{P_L}$, a variable $\mathbf{F_N}$ which represents a L-formula, name it **F**, and an input vector value $\mathbf{V_{inp}}$ for **F**. **F** can be a) an atomic L-formula, b) a L-formula defined in $\mathbf{P_L}$, or c) a L-formula to be evolved. In the cases (a) and (b) the value of $\mathbf{F_N}$ is **F**’s name viewed as a symbol, $\mathbf{V_{inp}}$ is a valid input vector value for **F**, while **Execute** calls **F** with $\mathbf{V_{inp}}$. In the case (c) the initial value of $\mathbf{F_N}$ is an empty list, $\mathbf{V_{inp}}$ is an arbitrary L-list, while **Execute** creates a **F** definition compatible with $\mathbf{V_{inp}}$, registers the **F** definition in $\mathbf{P_L}$ and calls **F** with $\mathbf{V_{inp}}$. **Execute** returns $\mathbf{P_L}$ (enlarged by **F**’s definition in the case (c)), $\mathbf{F_N}$ (changed to the name of **F** in the case (c)), and the output vector value $\mathbf{V_{out}}$ of the **F** call.

If **F** is defined in $\mathbf{P_G}$ as a first order atomic L-formula (which can be identified by the value of $\mathbf{F_N}$) (lines 2 and 3) then the high order G-formula **Call** (line 4) calls **F** with input vector value $\mathbf{V_{inp}}$ while returns the output vector value $\mathbf{V_{out}}$ of the **F** call.

If **F** is defined in $\mathbf{P_G}$ as a high order atomic L-formula (line 5) then:

a) the G-formula **Prepare_Forced_Call** (line 6) returns both the L-formula name $\mathbf{F_F}$ and the input vector value $\mathbf{V_{inp_F}}$ for the forced call. Here is the definition procedure for $\mathbf{F_F}$ and $\mathbf{V_{inp_F}}$:
   a1) if the first element of $\mathbf{V_{inp}}$, name it **e**, is either the name of an atomic L-formula or the name of a L-formula defined in $\mathbf{P_L}$ then define $\mathbf{F_F = e}$, otherwise define $\mathbf{F_F = (\,)}$.
   a2) define $\mathbf{V_{inp_F}}$ to be the remaining part of $\mathbf{V_{inp}}$ after removing the first element.
b) the G-formula **Execute** is called with input terms $\mathbf{P_L}$, $\mathbf{F_F}$ and $\mathbf{V_{inp_F}}$ (line 7).

If **F** is either defined in $\mathbf{P_L}$ as a non atomic L-formula or undefined (both cases are represented in line 9) then the following steps are realized.

If $\mathbf{F_N = (\,)}$, in which case **F** is undefined (line 10), then the G-formula **Initialize_Definition** (line 11) defines $\mathbf{F_N}$ to be a new symbol (not used before in the L-code), **con** to be a random L-connective consistent with $\mathbf{V_{inp}}$, $\mathbf{S_{inp}}$ to be a list of different random symbols where the size of $\mathbf{S_{inp}}$ equals the size of $\mathbf{V_{inp}}$, and $\mathbf{Ref_{lst}}$ to be a list of empty lists where the size of $\mathbf{Ref_{lst}}$ equals a number of references consistent with **con**. Then the G-formula **Add** (line 12) redefines $\mathbf{P_L}$ by appending at the end of $\mathbf{P_L}$ the list $\mathbf{(F_N, con, S_{inp}, Ref_{lst}, (\,), (\,))}$, which represents the current state of the **F** definition, as an element.

For the sake of simplicity, within the remainder of this presentation, an element of the **F** definition (e.g. the formula’s name, the input section, the reference section etc.), where the latter definition is registered in $\mathbf{P_L}$, will be referred to as an element of **F**. Consider that any element of **F** can be accessed by any G-formula having $\mathbf{P_L}$ and $\mathbf{F_N}$ as input terms.

The G-formula **Execute_References** (line 14) performs recursive **Execute** calls. Each recursive call realizes a L-formula call directly caused by the **F** call (the term “directly caused call” is defined in the file *GENETICA_Documentation.PDF*, § 2.3, p.8). Each one of these L-formula calls, except the recursive **F** call performed in the case of L-recursion, depends on an element of the reference section of **F**. If the latter element is

not an empty list then the reference section of **F** remains intact in $\mathbf{P_L}$; otherwise the element is substituted by a valid reference constructed during the respective **Execute** call. The latter reference may refer to an undefined L-formula (i.e. a L-formula whose definition is not included in $\mathbf{P_L}$). In this case the referenced L-formula's definition will be constructed and appended to $\mathbf{P_L}$. **Execute_References** returns both the revised $\mathbf{P_L}$ and $\mathbf{V_{out_{lst}}}$ which is the list of the output vector values of the directly caused L-formula calls.

The G-formula **Define_Output** (line 15) revises the output section of the **F** definition in $\mathbf{P_L}$ and creates $\mathbf{V_{out}}$ which is the output vector value of the **F** call. If the current output section is not an empty list then it remains intact. Otherwise it is redefined as a list of symbols randomly selected either from the input section or from the reference output sections of **F**. The random selection procedure respects L-syntax constraints relating the connective, the input section, the reference section and the output section of a L-formula (for instance if the L-connective does not denote recursion then no symbol from the input section could be selected for the output section). Due to the construction method of $\mathbf{V_{out_{lst}}}$ there is a map, name it **M**, from the set of symbols included in either the input section or the reference output sections of **F** to a set of structures of values included in the elements of $\mathbf{V_{out_{lst}}}$. Specifically **M** maps each symbol either to a single value (which is assigned to the symbol by a directly caused call) or to a list of values, each value included in a different element of $\mathbf{V_{out_{lst}}}$, (where each value is assigned to the symbol by a different directly caused call in the case of a L-quantifier) depending on the L-connective of **F**. The list $\mathbf{V_{out}}$ is derived from **M**, given the L-connective and the output section of **F**.

The G-formula **Define_IO_Type** (line 16) revises the type section of the **F** definition in $\mathbf{P_L}$. If the current type section is not an empty list then it remains intact. Otherwise it is substituted by a type section whose construction is based on the type sections of the L-formulae referenced in the **F** definition.

b) The G-formula "Execute_References"

As mentioned in the previous paragraph, the G-formula **Execute_References** calls (potentially also creates) the L-formulae referenced in the **F** definition.

The G-formula **Initialize_Table** (line 20) creates a table **T** where the one-to-one mapping between the symbols in the input section of **F** and the homologous values in $\mathbf{V_{inp}}$ is registered.

The empty list and the value **0** are assigned to the variables $\mathbf{V_{out_{lst}}}$ and **i** respectively (lines 21 and 22).

The **do-until** loop (lines 23 to 34) is executed once for each L-formula call directly caused by the **F** call. The confirmation of the termination condition (line 34) depends on the content of the **F** definition, on the value assignment registered in **T** and on the order **i** of the directly caused L-formula call. For instance, if **F** is constructed with a L-**and** then the termination condition is **i = k**, where **k** is the size of the reference section of **F**, whereas if **F** is constructed with a L-quantifier then the termination condition is **i = h**, where **h** is the size of the list—registered in **T**—referred to by the L-quantifier.

The G-formula **Get_Reference** (line 25) returns both **R**, which is an element of the reference section of **F,** and **n**, which is the order of **R** in the reference section. **R** defines the directly caused L-formula call. **n** (which specifies

**R**) depends both on the L-connective of **F** and on the order **i** of the directly caused L-formula call: for instance **n = i** in the case of a L-**and** where each directly caused call is defined by a different element of the reference section of **F**, whereas **n = 1** in the case of a L-quantifier where there is only one element, which defines all the directly caused calls, in the reference section of **F**. If the directly caused L-formula call is the recursive **F** call in the case of L-recursion (in which case the directly caused call is not represented in the reference section of **F**) then **R** is defined to be a two element list having fist element $\mathbf{F_N}$ and second element the input section of **F**, while **n** is ignored.

The G-formula **Prepare_Execute_Call** (line 26) returns $\mathbf{F_{N_R}}$, $\mathbf{S_{inp_R}}$ and $\mathbf{V_{inp_R}}$ where $\mathbf{F_{N_R}}$ and $\mathbf{S_{inp_R}}$ are respectively the referenced L-formula name and the input section of the reference represented by **R**, while $\mathbf{V_{inp_R}}$ is the input vector value for the directly caused call. Here is the definition procedure for the output values of **Prepare_Execute_Call**:

a) if $\mathbf{R \neq ( )}$ then define $\mathbf{F_{N_R}}$ and $\mathbf{S_{inp_R}}$ equal to the first and the second element of **R** respectively.

b) if $\mathbf{R = ( )}$ then define $\mathbf{F_{N_R}}$ and $\mathbf{V_{inp_R}}$ by randomly choosing one of the following definitions:

   b1) Define $\mathbf{F_{N_R} = ( )}$. Define $\mathbf{S_{inp_R}}$ to be a list of random symbols registered in **T**, where the random selection procedure respects L-syntax constraints relating **con**, $\mathbf{S_{inp}}$, and the first **n** elements of the reference section of **F**.

   b2) Define $\mathbf{F_{N_R}}$ to be the name of either a random atomic L-formula or a random L-formula in $\mathbf{P_L}$. Define $\mathbf{S_{inp_R}}$ to be a list of random symbols registered in **T**, where the random selection procedure respects both:

   - the requirement that the list of values where the list of symbols is mapped by **T** is type-compatible to the input of the L-formula.
   - L-syntax constraints relating **con**, $\mathbf{S_{inp}}$, and the first **n** elements of the reference section of **F**.

c) Let **V** be the vector value where $\mathbf{S_{inp}}$ is mapped by **T**. If **con** is a L-quantifier referring to a list-element, name it $\mathbf{q_{LST}}$, of **V** and $\mathbf{e_i}$ is the $\mathbf{i^{th}}$ element of $\mathbf{q_{LST}}$ then define $\mathbf{V_{inp_R}}$ to be the list derived from **V** when $\mathbf{q_{LST}}$ is substituted by $\mathbf{e_i}$. If **con** is not a L-quantifier then define $\mathbf{V_{inp_R} = V}$.

The directly caused L-formula call is realized through a call to **Execute** (line 27). This call returns $\mathbf{P_L}$ as the (potentially revised) L-code, $\mathbf{F_{N_R}}$ as the (potentially revised) name of the L-formula called and $\mathbf{V_{out_R}}$ as the output vector value of the directly caused call.

The G-formula **Add** redefines $\mathbf{V_{out_{lst}}}$ (line 28) by appending $\mathbf{V_{out_R}}$ as an element at the end of $\mathbf{V_{out_{lst}}}$.

If $\mathbf{R = ( )}$ (line 29), which means that the $\mathbf{n^{th}}$ reference of **F** is undefined, then the G-formula **Create_Reference_Output_Section** (line 30) returns $\mathbf{S_{out_R}}$ as a list of new symbols (not used before in the L-code), where the size of $\mathbf{S_{out_R}}$ equals the size of $\mathbf{V_{out_R}}$, while the G-formula **Revise_Reference_Section**

(line 31) redefines $\mathbf{P_L}$ by substituting the $\mathbf{n^{th}}$ reference, within the reference section of **F**, with the list $\mathbf{(F_{N_R}, S_{inp_R}, S_{out_R})}$.

The G-formula **Revise_Table** (line 33) registers in **T** the one-to-one mappings between the symbols in the output section of the $\mathbf{n^{th}}$ reference of **F** and the homologous values in $\mathbf{V_{out_R}}$, if the L-connective of **F** indicates that the output terms of a reference can be used as input terms in succeeding references.

### *1.3 Fitness evaluation considerations*

The fitness of a $\mathbf{P_G}$ execution should depend only on the fitness of the resultant $\mathbf{P_L}$ execution (determined in the lines 4, 7, 14, 27 of Table 1) and on the G-formulae involved in the L-code generation (lines 11, 15, 26 of Table 1). Any other G-formula call appearing in Table 1 should be made "always true" (see: *GENETICA_Documentation.PDF*, § 3.2.1.d, p.14).

The **do-until** loop (lines 23 to 34 of Table 1) has an important role in the construction of the fitness of the $\mathbf{P_L}$ execution: the loop's fitness should depend on the L-connective of **F** (see § 1.2.b). This loop could be implemented as a high order G-formula, name it **Loop**, which calls a loop-executing G-formula depending on the L-connective of **F.** For instance, name $\mathbf{Loop_{con}}$ the loop-executing G-formula that corresponds to the L-connective **con**. If **con** is either the L-**and** or the universal L-quantifier then $\mathbf{Loop_{con}}$ is constructed with the universal G-quantifier **app** (see *G-CAD_Documentation.PDF*: § 2.3), whereas if **con** is either the L-**or** or the existential L-quantifier then $\mathbf{Loop_{con}}$ is constructed with the existential G-quantifier **chs**.

Considering the optimization problems, note that a L-**opt** connective cannot be constructed by GENETICA's **opt**, since the latter only constructs the root G-formula. However it can be simulated by a non atomic G-formula whose fitness depends on the comparison of **0** with the inversed magnitude under maximization. Alternatively, a variation of $\mathbf{P_G}$ could include an **opt** root formula, as shown in section 2.

### *1.4 Implementation issues concerning domain specific and GP languages developed in GENETICA*

As shown in the previous paragraphs, the execution of the program $\mathbf{P_L}$ is an interpretation of the execution of the program $\mathbf{P_G}$. If all the L-formulae referenced in $\mathbf{P_L}$ are also defined in $\mathbf{P_L}$ then $\mathbf{P_L}$ remains intact during the $\mathbf{P_G}$ execution, since the G-formulae altering the L-code are never activated, while the whole data generation process in $\mathbf{P_L}$ is defined by the genetic list (GL) (see: *GENETICA_Documentation.PDF*, § 3.1) of the $\mathbf{P_G}$ execution. In this case **L** could be considered as a GENETICA-like (non-GP) language where the code is fixed, whereas the data structures evolve reflecting the GL evolution which is controlled by the computational system of GENETICA's environment. If $\mathbf{P_L}$ includes undefined L-formulae references, represented as empty lists in the reference section of L-formulae definitions, then both the code generation process and the data generation process in $\mathbf{P_L}$ are defined by the GL of the $\mathbf{P_G}$ execution. In this case **L** could be considered as a GP language where both the code and the data structures evolve; the "function set" is the set of the atomic L-formulae and the L-formulae fully defined in $\mathbf{P_L}$,

while the "terminal set" is extracted from the input vector value for $\mathbf{P_L}$ (named $\mathbf{V_{inp}}$ in Table 1). Infinite development of a L-program can be prevented by any suitable control mechanism, for instance the controls used in recursion-enabled GP e.g. [3] (pp.31-32).

A GP method could be implemented by means of a L-formula that includes an undefined reference to the L-formula that constructs a solution and a reference to a fully defined solution evaluation L-formula. Hybrid GP and GENETICA-like methods could be implemented in a L-program that includes partially defined L-formulae, i.e. L-formulae including both defined and undefined references. Incomplete problem specific knowledge can be represented by partially defined L-formulae which are to be fully defined by GP-like evolution.

## 2 The development of the language G-CAD in GENETICA

### *2.1 Introduction to a GENETICA program that implements G-CAD*

The architectural design language G-CAD is described in the file *G-CAD_Documentation.PDF*. It is assumed that the reader is familiar with the terminology introduced in this file.

G-CAD is a non-GP GENETICA-like language where the code is fixed while data structures evolve. The structure of G-CAD resembles the structure of the language **L** introduced in § 1.1. However G-CAD differs from **L** in the following points:

- A formula definition in G-CAD, i.e. the definition of a Unit creation formula (*G-CAD_Documentation.PDF*, § 1.8, 2.3), has no type section: the type compatibility is a responsibility of the programmer, since G-CAD is not a GP language.
- A formula definition in G-CAD has a property section where values used during the G-CAD program execution are registered.
- Terms not included in the G-CAD code, i.e. the "implicit terms" (see *G-CAD_Documentation.PDF*, § 2.1), affect the execution of a G-CAD program.
- G-CAD does not include an optimization connective.
- The G-CAD connectives **G_OR** and **G_ONE** (i.e. the logical **OR** and the existential quantifier) cause calls that depend on random selections: when a **G_OR** formula is called, a referenced formula randomly selected from the reference section of the **G_OR** formula is called, while when a **G_ONE** formula is called, a randomly selected element of a list is included in the input values of the referenced formula's call.

G-CAD has been implemented as a GENETICA program named $\mathbf{P_{G_{G\text{-}CAD}}}$ which is based on the program $\mathbf{P_G}$ presented in § 1.2. $\mathbf{P_{G_{G\text{-}CAD}}}$ has input a G-CAD program, name it $\mathbf{P_L}$, and the input values for $\mathbf{P_L}$, while it has output the output of $\mathbf{P_L}$. The input of $\mathbf{P_L}$ includes (see: *G-CAD_Documentation.PDF*, § 3.1):

a) a World Unit consisted of standard (non evolvable) sub-Units
b) descriptions of Units to be evolved.

A description of a Unit to be evolved includes:

- a Spatial Map **M**, which represents the environment where the Unit is to be developed
- the name $\mathbf{F_N}$ of the Unit's definition formula which is included in $\mathbf{P_L}$
- the input vector value for the Unit's definition formula.

Standard Units and Units to be evolved represent static and dynamic elements of a solution respectively. The solution itself is represented as a Unit to be evolved.

During the $\mathbf{P_L}$ execution, which reflects the $\mathbf{P_{G_{G\text{-}CAD}}}$ execution, the Units to be evolved are successively developed and imprinted, each one to the respective Spatial Map, via a call to the respective Unit definition formula. Developed Units are successively appended to the World Unit. Each developing Unit may refer (through instancing) to any sub-Unit of the so far developed World Unit. This makes possible self-replication procedures within the World Unit's structure, which allow fractal-like growth of the World Unit.

The Unit developed last, which corresponds to the last Unit description in the input of $\mathbf{P_L}$, is considered as the solution of the design problem. The output of $\mathbf{P_L}$, which is also the output of $\mathbf{P_{G_{G\text{-}CAD}}}$, includes the developed World Unit, the Spatial Map where the solution Unit has been imprinted and the explicit (in G-CAD syntax) output of the solution Unit definition formula.

The basic structure of $\mathbf{P_{G_{G\text{-}CAD}}}$, presented in the next paragraph, is based on the structure of $\mathbf{P_G}$ presented in § 1.2. The features of $\mathbf{P_{G_{G\text{-}CAD}}}$ not encountered in $\mathbf{P_G}$ are summarized here:

- An optimization formula has been introduced in $\mathbf{P_{G_{G\text{-}CAD}}}$, since G-CAD does not include an optimization connective.
- G-CAD formulae definitions, appearing as terms in $\mathbf{P_{G_{G\text{-}CAD}}}$, include a property section.
- Code generation and term type calculation procedures are obsolete in $\mathbf{P_{G_{G\text{-}CAD}}}$ since G-CAD is not a GP language.
- Implicit and explicit G-CAD terms are separately processed in $\mathbf{P_{G_{G\text{-}CAD}}}$.
- The random selections performed by the connectives **G_OR** and **G_ONE** are programmed in $\mathbf{P_{G_{G\text{-}CAD}}}$.

### *2.2 Presentation of the GENETICA program that implements G-CAD*

A semi-formal description of the program $\mathbf{P_{G_{G\text{-}CAD}}}$, introduced in the previous paragraph, is presented in Table 2. The description respects the notation of Table 1.

The root formula in $\mathbf{P_{G_{G\text{-}CAD}}}$ is the G-formula **Optimize** (line 1) which is constructed with the G-connective **opt**. The solution construction formula is the G-formula **Create_Units** (line 2), while the solution evaluation formula is the G-formula **Last** (line 3).

```
1   Optimize (P_L, u_w, E_lst)
2       Create_Units (P_L, u_w, E_lst) → (u_w, V_out)
3       Last (V_out) → V_to_be_maximized
4       Return (u_w, V_out)

5   Create_Units (P_L, u_w, E_lst)
6       For_Each_Element ((F_N, M_inp, V_inp), E_lst) {
7           Execute (P_L, F_N, (M_inp, u_w, u_w, V_inp)) → (M_out, V_out)
8       }
9       Return (u_w, V_out)

10  Execute (P_L, F_N, (M_inp, u_w, u_p, V_inp))
11      if (Atomic (F_N)) then {
12          if (First_Order (F_N)) then {
13              Call (F_N, (M_inp, u_w, u_p, V_inp)) → (M_out, V_out)
14          } else {
15              Prepare_Forced_Call (V_inp) → (F_F, V_inp_F)
16              Execute (P_L, F_F, (M_inp, u_w, u_p, V_inp_F)) → (M_out, V_out)
17          }
18      } else {
19          Create_Child_node (V_inp, u_w, u_p) → u_c
20          Execute_References (P_L, F_N, (M_inp, u_w, u_c, V_inp)) → (M_out, V_out_lst)
21          Define_Output (P_L, F_N, V_out_lst) → V_out
22      }
23      Return (M_out, V_out)

24  Execute_References (P_L, F_N, (M, u_w, u_c, V_inp))
25      Initialize_Table (P_L, F_N, V_inp) → T
26      V_out_lst ← ( )
27      i ← 0
28      do {
29          Get_Call_Order  (P_L, F_N, i) → i
30          Get_Reference (P_L, F_N, i) → (R, n)
31          Prepare_Execute_Call (P_L, F_N, R, T, i) → (F_N_R, V_inp_R)
32          Execute (P_L, F_N_R, (M, u_w, u_c, V_inp_R)) → (M, V_out_R)
33          Add (V_out_lst, V_out_R) → V_out_lst
34          Revise_Table (P_L, F_N, T, n, V_out_R) → T
35      } until (Termination_condition (P_L, F_N, T, i))
36      Return (M, V_out_lst)

37      Definitions of atomic G-CAD formulae follow …
```

TABLE 2

Outline of the GENETICA program $\mathbf{P_{G\text{-}CAD}}$, which implements the language G-CAD, with respect to the notation introduced in Table 1.

The input of **Optimize** includes:

1. A G-CAD program $\mathbf{P_L}$.
2. A pointer $\mathbf{u_w}$ to a World Unit which initially includes only standard (non evolvable) Units.
3. A list $\mathbf{E_{lst}}$ of descriptions of Units to be evolved. Each element of $\mathbf{E_{lst}}$ is a list having the form $\mathbf{(F_N, M, V_{inp})}$, where $\mathbf{F_N}$ is the name of the $\mathbf{P_L}$ formula defining the Unit, **M** is a Spatial Map where the Unit is to be imprinted, and $\mathbf{V_{inp}}$ is an input vector value for the formula named $\mathbf{F_N}$.

Let $\mathbf{(F_{Nlast}, M_{last}, V_{inplast})}$ be the last element of $\mathbf{E_{lst}}$. **Optimize** returns the terms $\mathbf{u_w}$ and $\mathbf{V_{out}}$, where $\mathbf{V_{out}}$ is the output vector value of the formula named $\mathbf{F_{Nlast}}$. The term $\mathbf{u_w}$, which is the pointer to the World Unit, remains intact during the **Optimize** call; however the World Unit itself changes, as evolving Units are successively constructed and appended to it. The Unit constructed by the last element of $\mathbf{E_{lst}}$ is considered as the solution of the design problem. The last element of $\mathbf{V_{out}}$ is considered as the value to be maximized in the case of optimization. The G-formula **Last** (line 3) just returns the last element of $\mathbf{V_{out}}$.

Let **F** be the G-CAD formula having name $\mathbf{F_N}$, where the **F** definition is registered in $\mathbf{P_L}$.

Within the **Create_Units** definition (line 5), the loop **For_Each_Element** (lines 6 to 8), which is constructed with the universal G-quantifier **app**, calls the G-formula **Execute** once for each element of $\mathbf{E_{lst}}$. The element passed to each loop execution is represented by the list $\mathbf{(F_N, M_{inp}, V_{inp})}$ appearing as an input term for the formula **For_Each_Element**. The G-formula **Execute** calls **F** with input vector value $\mathbf{(M_{inp}, u_w, u_w, V_{inp})}$, while $\mathbf{(M_{out}, V_{out})}$ is the output vector value of the call (both the input and the output terms for a G-CAD formula have been presented in *G-CAD_Documentation.PDF*: § 2.1). The pointer $\mathbf{u_w}$ and the output vector value $\mathbf{V_{out}}$ emerging at the last loop execution—where the last element of $\mathbf{E_{lst}}$ has been passed—are returned by **Create_Units**.

The definitions of both the G-formulae, **Execute** (line 10) and **Execute_References** (line 24), presented in Table 2 differ from the definitions of the homonymous G-formulae presented in Table 1 in the following points:

1) The input vector value of a L-formula to be called, named $\mathbf{V_{inp}}$ in the definition of **Execute** in Table 1, has been substituted by $\mathbf{(M_{inp}, u_w, u_p, V_{inp})}$ which is the input vector value of a G-CAD formula as described in *G-CAD_Documentation.PDF*: § 2.1. Respectively the output vector, named $\mathbf{V_{out}}$ in Table 1, has been substituted by $\mathbf{(M_{out}, V_{out})}$ which is the output vector value of a G-CAD formula.
2) Procedures executed if $\mathbf{F_N}$ is either an empty list or the name of a G-CAD formula having undefined references, have been removed. Both $\mathbf{F_N}$ and $\mathbf{P_L}$ do not appear as output terms of G-formulae since they are never redefined.
3) A reference to the G-formula **Create_Child_node** has been added (line 19). This formula is called if **F** is a non atomic formula, i.e. if it is a formula constructing a Unit. **Create_Child_node** initializes the new Unit as a list that includes the standard value **2**, denoting "Unit" (the representation of a Unit is defined in *G-CAD_Documentation.PDF*: § 1.8), and the Unit's property list which is an element of $\mathbf{V_{inp}}$. Then appends the initialized Unit as an element at the end of the list pointed by $\mathbf{u_p}$ and returns a pointer $\mathbf{u_c}$ to

the initialized Unit. The initialized Unit represents a child node of the Unit pointed by $\mathbf{u_p}$ within the World Unit's tree structure.

4) The last input term in the definition of the G-formula **Execute_References** (line 20), i.e. the list $\mathbf{(M_{inp}, u_w, u_c, V_{inp})}$, differs from the respective term in the definition of the G-formula **Execute**, i.e. the list $\mathbf{(M_{inp}, u_w, u_p, V_{inp})}$, in the third element: the Units pointed by $\mathbf{u_p}$ and $\mathbf{u_c}$ have a parent-to-child relation which makes possible the development of the World Unit's tree, through the recursive interactions between **Execute** and **Execute_References**.
5) The table **T** created by the G-formula **Initialize_Table** (line 25) represents a map which is defined by both:
   - the one-to-one mapping between the symbols in the input section of **F** and the homologous values in $\mathbf{V_{inp}}$ (this is the definition of **T** presented in Table 1)
   - the one-to-one mapping between property names and property values, defined in the property section of **F**.
6) The G-formula **Get_Call_Order** (line 29) increases **i** by one (as in the line 24 of Table 1) in the case of the G-CAD connectives **G_AND**, **G_ALL** and **G_REC**. In the case of a **G_OR** connective **Get_Call_Order** returns an integer representing the order of a random reference in the reference section of **F**. In the case of the G-CAD quantifier **G_ONE**, **Get_Call_Order** returns an integer representing the order of a random element of the list referred to by the quantifier.

Evolving random selections performed during the execution of $\mathbf{P_L}$ specify:

a) the position, orientation and mirroring condition of each terminal (Primitive or Instance: see *G-CAD_Documentation.PDF*, § 1.7, 1.8) of an evolving object, given the terminal's Positioning List (*G-CAD_Documentation.PDF*, § 1.9) and the Spatial Map (*G-CAD_Documentation.PDF*, § 1.3) where the terminal is to be imprinted (*G-CAD_Documentation.PDF*: § 2.2.a, 2.2.c)
b) the dimensions of each terminal Primitive of an evolving object, given the Primitive's OCS and the Spatial Map's region where the Primitive is allowed to be placed (see *G-CAD_Documentation.PDF*: § 2.2.a)
c) the output of the multiple confirmation G-CAD secondary formulae (the term "multiple confirmation formula" has been defined in *GENETICA_Documentation.PDF*: § 2.1 p.2)
d) the referenced G-CAD formula which is to be called in the case of a **G_OR** connective
e) the input of the referenced G-CAD formula in the case of a **G_ONE** connective

All the aforementioned random selections evolve due to GENETICA's computational system since they are performed via the multiple confirmation GENETICA formulae appearing in $\mathbf{P_{G_{G\text{-}CAD}}}$: these are the formulae that implement the G-CAD multiple confirmation atomic formulae in the cases (a), (b) and (c), and the formula **Get_Call_Order** (Table 2, line 29) in the cases (d) and (e).

REFERENCES

[1] T. Yu and C. Clack, “PolyGP: A polymorphic Genetic Programming system in Haskell,” in *Proceedings of the Third Annual Genetic Programming Conference*, J.R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba, R.L. Riolo, Eds. San Francisco, California: Morgan Kaufmann Publishers, 1998, pp. 416-421.

[2] Matthias Fuchs, “Evolving term features: first steps,” Automated reasoning project, Research School of Information Sciences and Engineering, Australian National University. Tech. Rep. TR-ARP-04-99, 1999. Available: http://arp.anu.edu.au:80/ftp/techreports/1999/TR-ARP-04-99.ps.gz

[3] W.B. Langdon, *Genetic Programming and data structures: Genetic Programming + data structures = automatic programming*. Boston, Dordrecht, London: Kluwer Academic Publishers, 1998.