GENETICA: A Computer Language That Supports General Formal Expression With Evolving Data Structures

Lefteris Virirakis

***Abstract*—This paper presents a general problem-solving method combining the principles of artificial intelligence and evolutionary computation. The problem-solving method is based on the computer language GENETICA, which stands for "Genetic Evolution of Novel Entities Through the Interpretation of Composite Abstractions." GENETICAs programming environment includes a computational system that evolves data abstractions, viewed as genotypes of data generation scenarios for a GENETICA program, with respect to either confirmation or optimization goals. A problem can be formulated as a GENETICA program, while the solution is represented as a data structure resulting from an evolved data generation scenario. This approach to problem solving offers: 1) generality, since it concerns virtually any problem stated in formal logic; 2) effectiveness, since formally expressed problem-solving knowledge can be incorporated in the problem statement; and 3) creativity, since unpredictable solutions can be obtained by evolved data structures. It is shown that domain specific languages, including genetic programming ones, that inherit GENETICAs features can be developed in GENETICA. The language G-CAD, specialized to problem solving in the domain of architectural design, is presented as a case study followed by experimental results.**

***Index Terms*—Evolutionary designing, formal logic, genetic algorithms, genetic programming (GP), problem solving.**

I. Introduction

A. Creativity, Effectiveness, and Generality in Evolutionary Computational Methods

AN IMPORTANT property of evolutionary computation, referred to as "creativity," is the ability to generate novel objects whose structure and functionality is unexpected in terms of previously available knowledge. A major challenge to the evolutionary computational methods is to achieve a general and effective representation of the objects to be evolved. Generality refers to the widest applicability of the method in different problem domains. Effectiveness refers to the best use of existing problem specific knowledge which improves the performance of the computational resources by reducing the search space. Effectiveness has been considered as contradictory with both creativity and generality.

Classical genetic algorithms (GAs) represent to-be-evolved objects as fixed-length binary strings, the latter viewed as genotypes. This is a general but poor representation, severely constraining both creativity and effectiveness. Enhancement of the representation has been attempted by either tree-structured genotypes reflecting concept hierarchies [1, pp. 13, 18–20], or genotypes interpreted to logical expressions [2]. Specialized programming environments make possible the development of domain specific GA applications [3], while incorporation of domain specific knowledge in a GA has been attempted in recent methods [4]. The latter approaches promote effectiveness to the detriment of both creativity and generality.

Genetic programming (GP) [5] considers computer programs as the objects to be evolved, which allows both creativity and generality since virtually anything can be represented as a computer program. This is also a major step toward effectiveness since solution representation knowledge can be incorporated in the components of evolving programs. However, GP does not provide an equally effective representation of the problem to be solved: in classical GP the user is only allowed to define the elements of the programming language (i.e., the function set and the terminal set), the fitness function and potentially some aspects of the program architecture [5, pp. 33–36]. Some additional control of the problem representation has been attempted through syntax specifications potentially reflecting problem specific knowledge. Such attempts include "strong" typing [6], [7] and "polymorphic" typing [8], [9], where type matching and unification, respectively, make possible the representation of syntax constraints. Constraint-based [10] and logic attribute grammars [11] further enhance knowledge representation through the syntax. Logic programming allows an even more general and effective solutions' representation [12, p. 43], which is based on a highly expressive formal language (i.e., the first-order logic). Effectiveness has been pursued in GP by means of both advanced data structures [13] and advanced formal modes of expression, e.g., quantifiers [14, p. 160] (expressions "FOR-ALL" and "EXISTS"), [15, p. 425] (expressions "MAP" and "FILTER"), recursion [5, pp. 147–154], [15] and "high-order" formulas, i.e., formulas relating other formulas, the latter viewed as terms [9, p. 418] (λ abstractions), [15, p. 424], [8, p. 234].

The idea of using the expressive power of a formal language could be extended to the whole problem representation. Problem specific knowledge, which could be represented through a formal language, may include many more things than knowledge expressed through the syntax of the solution representation language. A simple example concerns confirmation problems, namely problems of which the solution is evaluated in a Boolean manner: a formal expression, which represents

Manuscript received December 9, 2002; revised April 17, 2003.

The author is with the National Technical University of Athens, Athens 11362, Greece (left@genetica-informatics.org).

Digital Object Identifier 10.1109/TEVC.2003.816581

1089-778X/03$17.00 © 2003 IEEE

problem specific knowledge and describes a potential solution, may be syntactically correct but yet disconfirmed.

The expressive power of first-order logic supports the problem representation in a class of evolutionary methods where data structures, which reflect the syntax of a formal language, are considered as the objects to be evolved. Examples include 1) evolving bit-vectors, inspired by model-theoretic knowledge representation, which have been used in machine learning [16] (systems GIL and COGIN), [17, pp. 223, 224], [18], [19] and in proving logical arguments [20]; 2) evolving tree-structured data tracing a formal syntax, which have been used in abstract interpretation [21] ("abstract substitutions") and in theorem proving [22] ("features"); and 3) formal logic GP methods developed within the framework of constructive induction [23] and theorem proving [24]. Due to the generality of expression offered by first-order logic, all the methods referenced above are general—each one in its own application domain—and more effective than their nonevolutionary counterparts. They also can cope with confirmation problems, since abstract data structures can be optimized with respect to general confirmation criteria. However, these methods are designed specifically for learning, proving or interpreting tasks, which severely limits their potential creativity.

The idea of combining the expressive potential of formal logic in the problem representation (not only in the solution representation) with the creative potential of evolution in the problem solution (in different problem-solving tasks) has been tested in methods that give interpretations of evolutionary techniques in highly expressive environments: domain specification, problem formulation, and solution representation are performed within the environment, while the computational procedure is performed within the evolutionary technique. Relevant applications include the integration of a GP solver with an artificial intelligence (AI) environment [14], the integration of evolutionary algorithms (EAs) with an object-oriented environment [25], and the integration of GP or EAs with fuzzy logic [26], [27]. The potential of this approach depends on the degree of automation of the interpretation process and the user control allowed. A relevant branch of investigation concerns the integration of evolutionary techniques with highly expressive environments not based on formal logic. Such environments have been implemented as evolving neural networks [28], [29].

B. Problem Solving Based on Evolving Data Structures

Typical problem solving in the computer represents the problem as a computer program and the solutions as data structures. This representation is considered expressive enough for general problem-solving tasks if the computer language where the program is written includes at least the partial recursive functions [30, pp. 181–182]. A problem representation based on first-order logic [30, pp. 6–9], which makes easier the incorporation of problem-solving knowledge, is encountered in AI languages. An even more effective representation is achieved in computer languages that implement high-order modes of expression by considering parts of the program as data structures. Data structures within different computer languages may include various types of data such as arrays, lists, property lists, objects, data bases, or even computer programs.

Problem solving can be viewed as a process of construction and evaluation of data structures during the program execution. Construction of different data structures creates the search space while evaluation selects the elements of the solution space. The elementary decisions, on which the data construction is based, concern the selection of an element of either a countable class (represented as a list, an array or an integer interval) or a continuous class (represented as a real interval). The selection may be either deterministic (e.g., the PROLOG-like exhaustive enumeration generating the search space in AI applications) or stochastic (e.g., the random selections encountered in stochastic optimization or genetic methods). The data structures may be evaluated either in a Boolean manner (e.g., the PROLOG-like confirmation criteria encountered in AI applications) or in a numerical manner (e.g., optimization criteria represented by numerical functions), where the evaluation procedure constitutes part of the program.

Consider a computer language where each nondeterministic elementary decision made during the program execution is converted to a deterministic one by a piece of data, opaque to the programmer, referred to as a gene: given the context where the decision is to be made, a gene deterministically defines the result of the decision. Then, a proper gene structure, viewed as a genotype, deterministically defines a data generation scenario which may lead either to the construction of a potential solution or to the disconfirmation of the problem statement. In the former case, the data generation scenario can be evaluated by a program function which maps potential solutions to numerical values. In the latter case, the data generation scenario can be evaluated as an effort to confirm the problem statement. In both cases, the evaluation provides a measure of fitness. Consider a computational system which constructs genotypes, evaluates fitness, and applies evolutionary optimization by performing genetic operations on a population of genotypes. The genotypes' evolution constitutes a search for data generation scenarios that solve the problem.

Extending both the research on evolving data structures and the idea to give interpretations of evolutionary techniques in highly expressive environments (see Section I-A.) the evolutionary approach to problem solving presented in this paper aims both to generality and effectiveness. Generality is supported since any mode of expression practically used in computer languages could be used in the problem representation. Effectiveness is supported since a highly expressive formal statement of problem-solving knowledge can be incorporated in the problem representation, in the AI fashion. Unlike other evolutionary approaches to formal logic, this approach does not constrain creativity since solutions of unpredictable structure and functionality, represented as data structures, could emerge by evolution. An important aspect of the proposed approach is the possibility to consider evolving data structures as computer programs. This makes possible to develop GP methods where problem specific knowledge could be fully incorporated in the problem representation.

C. GENETICA

GENETICA is a computer language integrated in a programming environment that includes an evolutionary computational

system. Given a GENETICA program, the computational system performs successive executions of the program, where each execution leads to a different data generation scenario. Differences between data generation scenarios are caused by the nondeterministic elementary decisions occurring during the program execution. These decisions depend on genes both created and structured by the computational system at run time. Specific gene structures constitute genotypes each one deterministically defining a data generation scenario. The successive program executions performed by the computational system produce a population of genotypes. The computational system evaluates the performance of each data generation scenario in either confirming a specific formula or optimizing the value of a specific function within the program. The evaluation provides a fitness value for the genotype of the data generation scenario. The computational system evolves the population by performing genetic operations on high-fitness genotypes, estimating the fitness of the genotypes that result from the genetic operations, and substituting low fitness genotypes in the population. The best fitness genotype of the population, after the evolution procedure, defines the data generation scenario that produces the solution. The solution is a data structure constructed by a specific formula within the program.

GENETICA includes the partial recursive functions, the logical operations, the basic arithmetic operations and the quantifiers. GENETICAs atomic terms are integers, reals, and symbols represented as strings. Nonatomic terms are lists—simple or nested—of atomic terms. Lists can be both constructed and processed in GENETICA by LISP-like formulas. GENETICAs formulas can be also treated as terms, which makes possible high-order modes of expression, since a class of formulas cause the execution of any formula whose name—viewed as a string—appears in their argument. All the modes of expression practically used in computer languages are either included or can be constructed in GENETICA, while practically all the data structures can be represented as nested lists.

A prototype PC version of GENETICAs programming environment, which includes the compiler, the evolutionary computational system and a user interface, has been developed in Visual C++. This version has been used for the implementation and the production of the results presented in this paper.

The remainder of this paper is organized as follows.

- Section II briefly presents the language GENETICA, including an example of GENETICA code. The aim of the presentation is to clarify the features of the language that are important to the understanding of both the expressive potential of GENETICA and the problem-solving method.
- Section III presents the evolutionary computational system within GENETICAs programming environment. The presentation includes the definition of the genotype of a data generation scenario, the generation method and the functionality of genotypes, the genetic operations, and the fitness evaluation method, while a unifying description of the evolutionary computational process concludes the presentation.
- Section IV presents a large scale GENETICA application concerning the development of the language G-CAD, which is specialized to problem solving in the domain of architectural design. The presentation includes an introduction to a general method for developing domain specific languages and implementing GP in GENETICA, a description of G-CAD viewed as a case study of the language development method, and a unifying description of the architectural design problem-solving method.
- Section V presents the formulation and the solution of an architectural design problem as a case study of the architectural design problem-solving method introduced in Section V. The presentation includes the formulation of the problem statement, the adjustment of the basic computational parameters within GENETICAs environment, and the results produced. This section ends with a comparison of the G-CAD design method with other evolutionary design methods, which reveals the most important features of GENETICA-based problem solving.
- Section VI presents concluding remarks on the approach to problem solving presented in this paper.

II. Description of the Language GENETICA

A. General Features of GENETICAs Formulas

The notion of a GENETICA "formula" includes all the operations, the functions, and the relations that either constitute GENETICAs atomic elements or can be constructed in GENETICA. GENETICAs formulas have distinct input and output terms. Given the input terms, the formula can be "called," i.e., the program represented by the formula can be executed. The output terms emerge as a result of the call. Relations are represented by a class of formulas returning no output value. These formulas will be referred to as "nonconstructive" formulas, whereas any other formula will be referred to as a "constructive" formula.

Atomic formulas in GENETICA include arithmetic operations, list processing functions, and both inclusion and equality relations. Specific GENETICAs atomic formulas are semantically equivalent with the basic arithmetic and set-theoretic operations and relations of the Zermelo–Fraenkel language $\mathrm{L_1Set}$ and the Peano language $\mathrm{L_1Ar}$ [30, p. 6]. The confirmation of an atomic formula depends only on the input terms.

The argument of a special class of atomic formulas, referred to as "high-order" formulas, consists of a list having the form $(F_N, t_1, \ldots t_n)$, where F_N is the name of a formula (atomic or not) viewed as a symbol. When a "high-order" formula is called, the formula named F_N is called with input terms $t_1, \ldots t_n$, while the output terms of the call are included in the output terms of the "high-order" formula call. The call to the formula named F_N, resulting from the "high-order" formula call, will be referred to as the "forced call." Any nonatomic formula making reference to a "high-order" formula is also defined to be a "high-order" formula (recursive definition). Non "high-order" formulas will be referred to as "first-order" formulas.

Specific atomic constructive formulas have a single output term, which is a random element of either a countable class viewed as a list, or a continuous class viewed as a real interval, where either class is defined by the input terms. These formulas

will be referred to as "multiple confirmation" formulas in the sense that, given the input vector value, different output values confirm the formula. Any nonatomic first-order formula making reference to a "multiple confirmation" formula is also defined to be a "multiple confirmation" formula (recursive definition). Nonmultiple confirmation first-order formulas will be referred to as "single confirmation" formulas.

The standard value *false* substitutes each output term of a constructive first-order atomic formula in the case of disconfirmation.

B. Terms and Data Structures

Atomic terms in GENETICA are integers, reals, and symbols. The empty list and the standard value *false* are also considered as atomic terms. If $t_1, \ldots, t_n$ are terms then the list $(t_1, \ldots, t_n)$ is also a term. This recursively defines any nested list to be a term. GENETICAs terms have formula scope and duration, i.e., they can be used only in the formula where they have been declared and only during an execution of a call to the formula. However, any list-term can be registered in memory pointed by an integer-pointer and stay there until it is explicitly discarded. This is the case of a pointer list. Pointer lists have arbitrary duration and can be accessed by any formula where their integer-pointer has been passed. Data structures in GENETICA are represented as lists and pointer lists constructed by the list processing formulas.

C. Syntax and Semantics: Composition and Functionality of Nonatomic Formulas

A formula definition in GENETICA is a list having the form (F_N, *con*, S_{inp}, Ref_{lst}, S_{out}), where F_N is the name of the formula, *con* is a connective, S_{inp} is the input section as a list of names of input terms, Ref_{lst} is the reference section as a list of references to formulas which are called when the formula named F_N is called, and S_{out} is the output section as a list of names of output terms. A formula reference is a list having the form (Ref_N, Ref_{Sin}, Ref_{Sout}), where Ref_N is the name of the referenced formula, while Ref_{Sin} is the input section and Ref_{Sout} is the output section of the reference as lists of the names of the referenced formula's input and output terms, respectively. Any symbol in the input section of a formula reference should be included either in S_{inp} or in the output section of a previous formula reference in Ref_{lst}, while any symbol in S_{out} should be included in the output section of a formula reference in Ref_{lst}. A program in GENETICA is a list of formula definitions.

A program can be represented as a tree structure, referred to as the "syntax tree," where each formula definition is represented as a node having child nodes the definitions of the referenced formulas. Atomic formulas are represented as terminals (leaf nodes). The formula of which the definition is represented as the root of the syntax tree represents the problem statement. This formula will be referred to as the "root formula."

GENETICA includes the connectives *and*, *or*, *not*, *app*, *chs*, *rec*, and *opt*, which are described as follows.

- *and*, *or*, and *not* are the basic logical operations. Formula definitions constructed with either *and* or *or* include an arbitrary number of references while formula definitions constructed with *not* include only one formula reference. Each referenced formula is called once. Each output term of a formula F constructed with *and* is represented in the output section of the F definition. A formula F constructed with *or* has a single output term which is the list of the terms whose names appear both in the output sections of the confirmed references and in the output section of the F definition; the order of the terms in the list respects the order of the respective term names in the output section of the F definition. Formulas constructed with *not* have no output terms.
- *app* (apply) and *chs* (choose) are the universal and the existential quantifier, respectively. The reference section of a formula definition constructed with either *app* or *chs* includes a single reference which is called once for each element of an input list, labeled as L_{in}. The elements of L_{in} represent the term referred to by the quantifier. The output vector values of the confirmed calls constitute the elements of an output list, labeled as L_{out}, where the order of the elements in L_{out} respects the order of the corresponding elements in L_{in}. Each formula constructed with *app* is confirmed if and only if all the calls to the referenced formula are confirmed. Each formula constructed with *chs* is confirmed if and only if at least one call to the referenced formula is confirmed.
- *rec* (recursion) represents the conditional recursion. The reference section of a formula definition constructed with *rec* includes two references: the first represents the termination condition while the second returns terms used as input terms for the recursive call. A call to a *rec* formula causes a call to the first referenced formula. If this call is confirmed then the *rec* formula is also confirmed, otherwise the second referenced formula is called. If the latter call is confirmed then the recursive call to the rec formula is performed, otherwise the *rec* formula is disconfirmed. Specific output terms of the *rec* formula also constitute input terms of the recursive call.
- *opt* (optimize) only constructs the root formula in optimization problems. The reference section of a formula definition constructed with *opt* includes two references, each one called once: the first reference returns terms which represent a potential solution, while the second reference evaluates these terms and returns a positive quantity which is to be maximized. The *opt* formula is confirmed, if and only if the first reference is confirmed.

An important property of GENETICAs connectives, derived from the previous definitions, is that the output values of the referenced formula calls emerging due to a call to a nonatomic formula F deterministically define the output values of the F call. This makes certain that the output values produced by the first-order atomic formula calls deterministically define the output values of all the formula calls occurring during the program execution, i.e., deterministically define a data generation scenario.

A full presentation of the syntax of GENETICAs prototype version, which is based on the principles presented here, is included in the file Genetica_Documentation.pdf, available in www.genetica-informatics.org.

TABLE I
GENETICA PROGRAM THAT STATES A PROBLEM HAVING BOTH CONFIRMATION AND OPTIMIZATION GOALS. F_0 IS THE ROOT FORMULA HAVING INPUT TERMS a_1, a_2, AND a_3, AND OUTPUT TERMS x_1, x_2, AND x_3. x_4 IS THE MAGNITUDE TO BE MAXIMIZED

```
(F0   opt    (a1, a2, a3)
   (
      (F1    (a1, a2, a3)  (x1, x2, x3))
      (F2    (x1, x2, x3)  (x4))
   )
   (x1, x2, x3)
)

(F1   and    (b1, b2, b3)
   (
      (mem   (b1)          (y1))
      (mem   (b2)          (y2))
      (sml   (y1, y2)      ())
      (in    (y1, b3)      (y3))
   )
   (y1, y2, y3)
)

(F2   and    (c1, c2, c3)
   (
      (F3    (c1, c2, c3)  (z1))
      (mem   (z1)          (z2))
   )
   (z2)
)

(F3   or     (d1, d2, d3)
   (
      (sqr   (d2)          (w1))
      (mns   (d3, d1)      (w2))
   )
   (w1, w2)
)
```

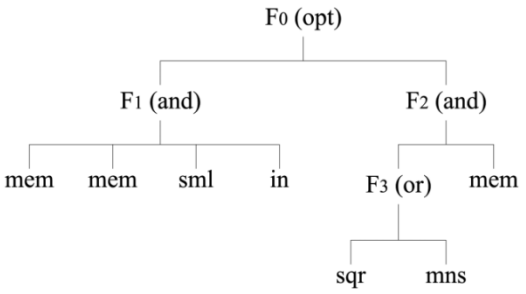


Fig. 1. Syntax tree of the program presented in Table I. The root node represents the root formula while each other node represents a formula referenced by the parent node formula. Nonterminal nodes represent nonatomic formulas, whereas terminal nodes represent atomic formulas. The connective that constructs each nonatomic formula appears after the formula's name in parentheses.

D. An Example of GENETICA Programming

Consider the following statement of a simple problem having both confirmation and optimization goals.

- Let a_1 and a_2 be lists of real numbers; let a_3 be a real number.
- Select an element of a_1, labeled as x_1, and an element of a_2, labeled as x_2.
- Verify that $x_1 < x_2$ (confirmation goal).
- Verify that $x_1 \leq a_3$ and select a number, labeled as x_3, in the real interval $[x_1, a_3]$ (confirmation goal).
- Consider the list (x_1, x_2, x_3) as a potential solution.
- Define x_4 to be either the square root of x_2 or the difference $x_3 - x_1$.
- Maximize x_4 (optimization goal).

The problem statement is presented as a GENETICA program in Table I. The program demonstrates the syntax introduced in Section II-C. The syntax tree is presented in Fig. 1.

The atomic formulas used in the GENETICA program are described here.

- The multiple confirmation formula *mem*, which has a single input term t, is confirmed if and only if t is a nonempty list and returns a randomly chosen element of t.
- The nonconstructive formula *sml*, which has two input terms t_1 and t_2, is confirmed if and only if both t_1 and t_2 are reals, where $t_1 < t_2$.
- The multiple confirmation formula *in*, which has two input terms t_1 and t_2, is confirmed if and only if both t_1 and t_2 are reals, where $t_1 \leq t_2$, and returns a random real in the interval $[t_1, t_2]$.
- The single confirmation formula *sqr*, which has a single input term t, is confirmed if and only if t is a positive real number and returns the square root of t.
- The single confirmation formula *mns*, which has two input terms t_1 and t_2, is confirmed if and only if both t_1 and t_2 are real numbers and returns t_1 minus t_2.

The root formula is the formula F_0 which includes two references: a reference to the formula F_1, which constructs a potential solution represented by the variables x_1, x_2, and x_3, and a reference to the formula F_2, which calculates the value of the magnitude to be maximized (see Section II-C: connective *opt*) represented by the variable x_4. The formula F_3, referenced in the F_2 definition, returns the list of the possible values of the magnitude to be maximized.

F_1 assigns to b_1, b_2, and b_3 the values of a_1, a_2 and a_3, respectively; to y_1 a random element of the list b_1 (first reference to the atomic formula *mem*); to y_2 a random element of the list b_2 (second reference to *mem*); confirms that $y_1 < y_2$ (reference to the atomic formula *sml*) and, if $y_1 \leq b_3$, assigns to y_3 a random real in the interval $[y_1, b_3]$ (reference to the atomic formula *in*). The values of y_1, y_2, and y_3 are assigned to the variables x_1, x_2, and x_3 in F_0.

F_2 assigns to c_1, c_2, and c_3 the values of x_1, x_2, and x_3, respectively; to z_1 the list of the possible values of the magnitude to be maximized (reference to F_3) and to z_2 a random element of the latter list (reference to *mem*). The value of z_2 is assigned to the variable x_4 in F_0.

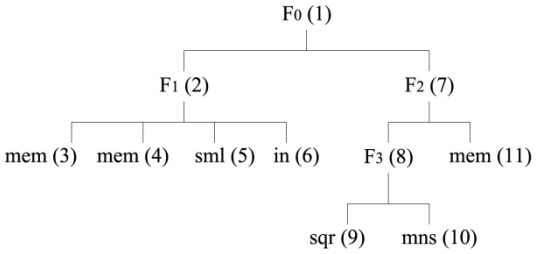


Fig. 2. Call tree of an execution of the program presented in Table I. The root node represents a call to the root formula, while each other node represents a call directly caused by the parent node call. The name of the formula called is followed by the order of the call in parentheses. The calls are performed in depth-first order with respect to the call tree.

F_3 assigns to d_1, d_2, and d_3 the values of c_1, c_2, and c_3, respectively; to w_1 the square root of d_2 (reference to the atomic formula *sqr*); to w_2 the difference $d_3 - d_1$ (reference to the atomic formula *mns*) and constructs the list (w_1, w_2), or the sublist specified by the confirmation state of the referenced formulas (see Section II-C: connective *or*). The constructed list is assigned to z_1 in F_2.

III. Computational System of GENETICAs Environment

A. Genotype of a Formula Call

1) Introduction to the Genotype of a Formula Call:

A formula call is a program execution attempting to confirm the formula, compute the output values, and potentially optimize them, given the input values. When a nonatomic formula F is called, all the formulas referenced in the definition of F are also called. The latter calls will be referred to as calls "directly caused" by the F call. Since directly caused calls may also be calls to nonatomic formulas, new calls may be directly caused by them. All these calls (directly or indirectly caused by the F call) will be referred to as calls "recursively caused" by the F call.

The recursively caused calls can be represented as nodes in a tree structure, where each parent node "directly causes" the child nodes. This structure will be referred to as the "call tree." The call tree presented in Fig. 2 is produced during the execution of the program presented in Table I. This call tree is isomorphic with the syntax tree presented in Fig. 1, i.e., each node in the one tree has a homologous node in the other. However, call trees are not always isomorphic with the respective syntax trees, due to the following reasons:

- a call subtree rooted at a quantifier node can be developed in arbitrary width (i.e., number of child nodes), depending on the size of the list whose elements represent the term referred to by the quantifier;
- a call subtree rooted at a recursion node can be developed in arbitrary depth (i.e., length of the route connecting the subtree's root node with a terminal node), depending on the number of the recursions.

Arbitrary subtree development also occurs at nodes that represent calls to high-order atomic formulas.

The data generation process occurring during a formula call determines both the confirmation state and the output values of the call and all the recursively caused calls. Data generation is a nondeterministic process since it is sensitive to variations ultimately originated in the nondeterministic behavior of the multiple confirmation atomic formulas (see Section II-A) whose calls appear as terminals in the call tree. As a consequence, different program executions, which realize different calls to the same formula, may result in different data generation scenarios each one having a different degree of success with respect to the confirmation and/or optimization goals.

Each data generation scenario can be represented as a tree structured object that copies the structure of a call tree and has terminal data that deterministically define the behavior of the homologous terminals in the call tree. These data will be referred to as "genes." Tree structured objects having genes as terminal nodes could be considered as "genotypes" of data generation scenarios if their representation satisfies the following conditions:

1) representation includes every possible data generation scenario, including scenarios disconfirming the formula called: this makes certain the sufficiency of the representation;
2) each object represents one and only one data generation scenario: this makes certain that the information defining a data generation scenario is preserved in the representation;
3) partially similar objects represent partially similar data generation scenarios: this makes possible effective genetic operations on the objects;
4) information not affecting the data generation is excluded from the representation: this makes the representation compact, while reduces the probability of ineffective genetic operations.

Every tree structured object the terminals of which satisfy the gene representation conventions, will be referred to as a "gene structure." It is clear that not all the gene structures could be considered as genotypes: an arbitrary tree structure could not be assigned to a call tree in advance, because the structure of a call tree depends on data generated at run time. Only specific gene structures, depending on both the syntax tree of the formula called and the input vector value of the call, match the structure of call trees. These gene structures define specific data generation scenarios and constitute genotypes. The genotype of a data generation scenario will be also referred to as the genotype of a call, since each call represents a specific data generation scenario.

Generation, interpretation, and variation procedures concerning genotypes are briefly described here.

- Genotypes are generated at run time. A node of the genotype's tree structure is generated each time a recursively caused call is performed, while a random gene is generated and attached as a terminal each time a multiple confirmation atomic formula is called.
- During the interpretation phase, which is also performed at run time, the genotype's tree is aligned to the (isomorphic) call tree, while the genes determine the result of the homologous calls to the multiple confirmation atomic formulas. The data generation scenario encoded in the genotype is realized.

- Variations on genotypes are caused by genetic operations. Such operations create gene structures that are not necessarily genotypes. These gene structures are transformed to genotypes via modifications preserving gene information. The transformation, which is performed at run time, combines interpretation and generation actions. Specifically, the subtrees of a gene structure are manipulated in three ways: those with homologous isomorphic subtrees in the call tree are preserved and interpreted; those without are cropped, while those missing are generated. Subtrees of the gene structure are missing when subtrees in the call tree have no homologous isomorphic subtrees in the gene structure. The transformation of a gene structure to a genotype will be referred to as "reconstruction."

The computational system of GENETICAs environment provides a unified procedure performing genotype generation, interpretation and reconstruction. Generation and interpretation are considered as special cases of reconstruction, specifically, the following.

1) Generation occurs by the reconstruction of an elementary gene structure, which is represented as a tree having only the root node. As a consequence, reconstruction leads to the generation of the whole genotype's tree.
2) Interpretation occurs when the gene structure to be reconstructed constitutes a genotype. In this case, the gene structure remains intact, while reconstruction leads to interpretation.

A call to an arbitrary formula F maps an input vector value V_{in} to an output vector value V_{out}, given a gene structure s. During the call, a reconstruction procedure maps s to the genotype g of the call. As a consequence, the pair (s, V_{in}) is mapped to the pair (g, V_{out}). This mapping is an instance of a map referred to as a "genetic map" for F. The definitions of both the reconstruction and the data generation procedure derive from the general definition of the genetic map for a GENETICA formula. This definition is given in the next paragraphs followed by illustrative examples.

2) Gene Structures and Gene Interpretation Functions:

Real values defined in the interval [0, 1] are considered as "genes." A "gene structure" is either a single gene or a (potentially nested) list whose innermost lists are either empty lists or one-element lists each one including a single gene. For instance, the list $s = ((((\ ),(0.8)),(0.7),(0.2)),((0.3),(0.6)),(\ ))$ is a gene structure. Nested lists can be represented as tree structures, where inclusion relations are represented as parent-child relations and innermost lists are represented as terminals. The tree representation of the list s is shown in Fig. 3.

If F is a multiple confirmation atomic formula (see Section II-A), then there is a function GIF_F, referred to as the "gene interpretation function" for F, mapping $[0,1] \times R_{V\text{in}}$ into $[0,1] \times R_{V\text{out}}$, where $R_{V\text{in}}$ is the space of the input vector values that confirm F, and $R_{V\text{out}}$ is the space of the output vector values for F. GIF_F has the following properties.

1) $\forall\ u \in [0,1]$, $\forall\ V_{\text{in}} \in R_{V\text{in}}$: $[GIF_F(u, V_{\text{in}}) = (w, V_{\text{out}}) \Rightarrow V_{\text{out}}$ is the output vector specified by the definition of F for the input vector $V_{\text{in}}]$.

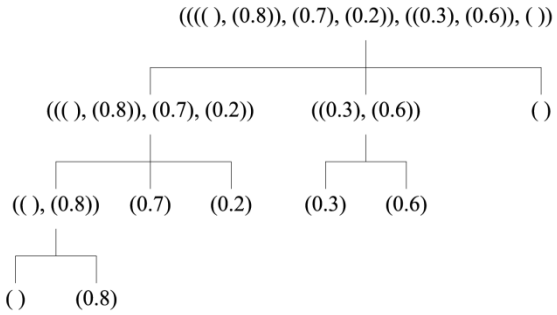


Fig. 3. Tree representation of the list $((((\ ),(0.8)),(0.7),(0.2)),((0.3),(0.6)),(\ ))$. Inclusion relations are interpreted as parent-child relations.

2) $\forall\ u \in [0,1]$, $\forall\ V_{\text{in}} \in R_{V\text{in}}$: $[GIF_F(u, V_{\text{in}}) = (w, V_{\text{out}}) \Rightarrow GIF_F(w, V_{\text{in}}) = (w, V_{\text{out}})]$.
3) $\forall\ u_1, u_2 \in [0,1]$, $\forall\ V_{\text{in}} \in R_{V\text{in}}$: $[GIF_F(u_1, V_{\text{in}}) = (w_1, V_{out1}) \wedge GIF_F(u_2, V_{\text{in}}) = (w_2, V_{out2}) \Rightarrow [w_1 = w_2 \Leftrightarrow V_{out1} = V_{out2}]]$.

For instance the atomic formulas *in* and *mem*, both presented in Section II-D, having input vectors (t_1, t_2) and (t) respectively, have the following gene interpretation functions, both having the aforementioned properties:

$$GIF_{\text{in}}\left((u,(t_1,t_2)\right) = (u,(t_1 + u\cdot(t_2 - t_1)))$$
$$GIF_{mem}\left(u,(t)\right) = \left(\frac{i+0.5}{k},(e_{i+1})\right)$$

where

— $t = (e_1, \ldots e_k)$, i.e., t is a list of k elements labeled as $e_1, \ldots e_k$;
— $u \neq 1 \Rightarrow i$ is the integer part of the product $u \cdot k$;
— $u = 1 \Rightarrow i = k - 1$.

3) Genetic Maps and Genetic Lists:

For any formula F there is a class of maps, referred to as "genetic maps" for F, mapping $S \times R_{V\text{in}}$ into $S \times R_{V\text{out}}$, where S is the space of the gene structures, $R_{V\text{in}}$ is the space of the input vector values for F, and $R_{V\text{out}}$ is the space of the output vector values for F. Let M_F be the generic element of the class of the genetic maps for F. If $(s, V_{\text{in}}) \xrightarrow{M_F} (g, V_{\text{out}})$ then s will be referred to as the "call guide" of a call to F, while g will be referred to as the "genetic list" (GL) of the call. The GL constitutes the genotype of the call. The genetic mapping process, which takes (s, V_{in}) to (g, V_{out}), is recursively defined by the following rules:

a) if F is a first-order atomic formula, then:
  a1) if F is a single confirmation formula (see Section II-A), then define g and V_{out} to be, respectively, an empty list and the output vector value specified by the definition of F for the input vector V_{in};
  a2) if F is a multiple confirmation formula (see Section II-A), then:
    a2.1) if F is confirmed, then:
    a2.1.1) if $s = (v)$, where $v \in [0,1]$, then define $u = v$;
    a2.1.2) if $\forall v \in [0,1]$: $s \neq (v)$ then define u to be a random value in [0, 1];

a2.1.3) define $g = (w)$, where $GIF_F(u, V_{\text{in}}) = (w, V_{\text{out}})$;

a2.2) if F is not confirmed then define g to be an empty list and define V_{out} by assigning the standard value *false* to each output term of F.

b) if F is a nonatomic formula, then:

name s_i and $g_i(i = 1, \ldots n)$, respectively, the "call guide" and the GL of the i^{th} call directly caused by the F call, where n is the number of the calls directly caused by the F call;

b1) if s is not a list, then define $s_i(i = 1, \ldots n)$ to be an empty list;

b2) if s is a list and $k \geq 0$ is the size of s, then:

b2.1) if $k > 0$ and $s = (e_1, \ldots e_k)$, then define $s_j = e_j(j = 1, \ldots min(k, n))$;

b2.2) if $n > k$, then define $s_j(j = k + 1, \ldots n)$ to be an empty list.

b3) if at least one $g_i(i = 1, \ldots n)$ is not an empty list, then define $g = (g_1, \ldots g_n)$;

b4) if $g_i(i = 1, \ldots n)$ is an empty list, then define g to be an empty list;

b5) define V_{out} to be the output vector value specified by the connective in the definition of F, given the confirmation state and the output vector values of the directly caused calls (see Section II-C).

c) if F is a high-order atomic formula then define the "call guide" of the "forced call" (see Section II-A) equal to s and then define g equal to the GL of the "forced call"; define V_{out} to be the output vector value specified by the definition of F, given the output vector value of the "forced call."

The definition of the genetic map for F presented here satisfies all the specifications introduced in Section III-A1 for the data generation scenario's genotype, which is represented here as a GL, while includes all the generation, interpretation and reconstruction procedures introduced in the same paragraph. Specifically, the following.

1) Given the input values for a formula call, all the syntax-consistent data generation scenarios, including disconfirmation scenarios, are represented by GLs. Any GL—and the respective data generation scenario—could emerge if the "call guide" is an empty list.
2) Given the input values for a formula call, a GL specifies one and only one data generation scenario. Specifically, if a GL of a F call is used as a "call guide" in a new F call with the same input values then
   - the call trees of the two F calls are identical;
   - the confirmation state, the input values, the output values and the GLs of these calls and all the recursively caused calls represented by homologous nodes in the two call trees are identical.
3) If s is the "call guide" of a F call, then every gene that belongs to s and affects data generation has a homologous gene in the GL of the call. The latter gene, appearing as a homologous terminal in the tree representation of a different "call guide," generates similar data if interpreted under similar context. Missing subtrees in the tree representation of s, indicated by call subtrees having no homologous isomorphic subtrees in s, are developed in the GL, whereas redundant subtrees, i.e., s subtrees having no homologous isomorphic subtrees in the call tree, are cropped in the GL.
4) If F is a single confirmation formula (see Section II-A) then the GL of any call to F is an empty list.

4) An Example of Genetic Mapping:

An example of genetic mapping, i.e., the procedure that takes a "call guide" to a GL and realizes a data generation scenario, is given in Tables II(a) and II(b). Each table presents a different execution of the program presented in Table I. In both Table II(a) and (b), it is supposed that the gene structure $s = ((((\,),(0.8)),(0.7),(0.2)),((0.3),(0.6)),(\,))$, also presented in Section III-A2, is the "call guide" of the call to the root formula F_0, while the variables a_1, a_2, and a_3 (see Table I) are assigned the values $(5.5, 2, 9)$, $(2, 3, 4, 5)$, and 8, respectively.

In both Table II(a) and (b), successive rows present successive phases of the program execution. The columns, in the order they appear in each table, present, respectively: 1) the order of the current call (appearing also in the call tree presented in Fig. 2); 2) the name of the formula called; 3) the "call guide" of the call; 4) random values generated by the computational system due to genetic mapping rule a2.1.2 (see Section III-A3); 5) the call's GL derived from the "call guide" via the genetic mapping process; 6) the values of newly defined variables; and 7) the GL of call 1, i.e., the call to the root formula. The asterisks represent GLs elements not yet defined in the current computational phase.

Both data generation and GL construction, appearing in both Table II(a) and II(b), are determined by the application of genetic mapping rules (see Section III-A3) on the calls occurring during the program execution. The genetic mapping rules applied to each call depends on the rule-firing conditions (as defined in Section III-A3) satisfied by the call.

An explanation of the genetic mapping process presented in Table II(a) is given in the following lines. The reader should recall both the program presented in Table I and the genetic mapping rules presented in Section III-A3.

When the formula F_0 is called (call 1), the GL of the call, if not empty (it is empty only if the call satisfies the premise of genetic mapping rule b4, Section III-A3), is expected to have two elements. This is determined by rule b3, as two calls are directly caused by call 1.

Call 2, which is a call to the formula F_1 with input terms $b_1 = (5.5, 2, 9)$, $b_2 = (2, 3, 4, 5)$, and $b_3 = 8$, is the first call directly caused by call 1. Due to rule b2.1 applied to call 1, the "call guide" of call 2 is the first element of the "call guide" of call 1: that is, $(((\,),(0.8)),(0.7),(0.2))$. Due to rule b3 applied to call 2, the GL of call 2 (if not empty) is expected to have four elements, as four calls are directly caused by call 2.

Call 3, which is a call to the atomic formula *mem* (see Section II-D) with input term $b_1 = (5.5, 2, 9)$, is the first call directly caused by call 2. Due to rule b2.1 applied to call 2, the "call guide" of call 3 is the first element of the "call guide" of call 2: that is $((\,)(0.8))$. At this point, due to rule a2.1.2 applied to call 3, a random number, say 0.61, is generated. The GL and the output value of call 3 are defined to be (0.5) and 2, respectively, due to rule a2.1.3, as $GIF_{\text{mem}}(0.61, (5.5, 2, 9)) = (0.5, (2))$ (see Section III-A2: GIF_{mem} definition).

TABLE II
TWO DIFFERENT DATA GENERATION SCENARIOS, PRODUCED BY DIFFERENT GENETIC MAPS, FOR THE PROGRAM PRESENTED IN TABLE I ARE PRESENTED IN TABLE 2a AND 2b. IN BOTH TABLES SUCCESSIVE ROWS PRESENT SUCCESSIVE PHASES OF THE PROGRAM EXECUTION, WHILE EACH COLUMN PRESENTS THE STATE OF A DIFFERENT ENTITY OF THE COMPUTATIONAL PROCESS

Call	Formula called	"Call guide"	Random values	GL of the current call	Terms defined	GL of call 1
1	F_0	((( ), (0.8)), (0.7), (0.2)), ((0.3), (0.6)), ( ))		(*, *)	$a_1 = (5.5, 2, 9)$ $a_2 = (2, 3, 4, 5)$ $a_3 = 8$	(*, *)
2 (starting)	F_1	((( ), (0.8)), (0.7), (0.2))		(*, *, *, *)	$b_1 = (5.5, 2, 9)$ $b_2 = (2, 3, 4, 5)$ $b_3 = 8$	((*, *, *, *), *)
3	mem	(( ), (0.8))	0.61	(0.5)	$y_1 = 2$	(((0.5), *, *, *), *)
4	mem	(0.7)		(0.625)	$y_2 = 4$	(((0.5), (0.625), *, *), *)
5	sml	(0.2)		( )		(((0.5), (0.625), ( ), *), *)
6	in	( )	0.15	(0.15)	$y_3 = 2.9$	(((0.5), (0.625), ( ), (0.15)), *)
2 (ending)	F_1			((0.5), (0.625), ( ), (0.15))	$x_1 = 2$ $x_2 = 4$ $x_3 = 2.9$	(((0.5), (0.625), ( ), (0.15)), *)
7 (starting)	F_2	((0.3), (0.6))		(*, *)	$c_1 = 2$ $c_2 = 4$ $c_3 = 2.9$	(((0.5), (0.625), ( ), (0.15)), (*, *))
8 (starting)	F_3	(0.3)		(*, *)	$d_1 = 2$ $d_2 = 4$ $d_3 = 2.9$	(((0.5), (0.625), ( ), (0.15)), ((*, *), *))
9	sqr	0.3		( )	$w_1 = 2$	(((0.5), (0.625), ( ), (0.15)), ((( ), *), *))
10	mns	( )		( )	$w_2 = 0.9$	(((0.5), (0.625), ( ), (0.15)), ((( ), ( )), *))
8 (ending)	F_3			( )	$z_1 = (2, 0.9)$	(((0.5), (0.625), ( ), (0.15)), (( ), *))
11	mem	(0.6)		(0.75)	$z_2 = 0.9$	(((0.5), (0.625), ( ), (0.15)), (( ), (0.75)))
7 (ending)	F_2			(( ), (0.75))	$x_4 = 0.9$	(((0.5), (0.625), ( ), (0.15)), (( ), (0.75)))

(a)

Call	Formula called	"Call guide"	Random values	GL of the current call	Terms defined	GL of call 1
1	F_0	((( ), (0.8)), (0.7), (0.2)), ((0.3), (0.6)), ( ))		(*, *)	$a_1 = (5.5, 2, 9)$ $a_2 = (2, 3, 4, 5)$ $a_3 = 8$	(*, *)
2 (starting)	F_1	((( ), (0.8)), (0.7), (0.2))		(*, *, *, *)	$b_1 = (5.5, 2, 9)$ $b_2 = (2, 3, 4, 5)$ $b_3 = 8$	((*, *, *, *), *)
3	mem	(( ), (0.8))	0.68	(0.833)	$y_1 = 9$	(((0. 833), *, *, *), *)
4	mem	(0.7)		(0.625)	$y_2 = 4$	(((0. 833), (0.625), *, *), *)
5	sml	(0.2)		( )		(((0. 833), (0.625), ( ), *), *)
6	in	( )	0.15	( )	y_3 = false	(((0. 833), (0.625), ( ), ( )), *)
2 (ending)	F_1			((0.833), (0.625), ( ), ( ))	$x_1 = 9$ $x_2 = 4$ x_3 = false	(((0. 833), (0.625), ( ), ( )), *)
7 (starting)	F_2	((0.3), (0.6))		(*, *)	$c_1 = 9$ $c_2 = 4$ c_3 = false	(((0. 833), (0.625), ( ), ( )), (*, *))
8 (starting)	F3	(0.3)		(*, *)	$d_1 = 9$ $d_2 = 4$ d_3 = false	(((0. 833), (0.625), ( ), ( )), ((*, *), *))
9	sqr	0.3		( )	$w_1 = 2$	(((0. 833), (0.625), ( ), ( )), ((( ), *), *))
10	mns	( )		( )	w_2 = false	(((0. 833), (0.625), ( ), ( )), ((( ), ( )), *))
8 (ending)	F_3			( )	$z_1 = (2)$	(((0. 833), (0.625), ( ), ( )), (( ), *))
11	mem	(0.6)		(0.5)	$z_2 = 2$	(((0. 833), (0.625), ( ), ( )), (( ), (0.5)))
7 (ending)	F_2			(( ), (0.5))	$x_4 = 2$	(((0. 833), (0.625), ( ), ( )), (( ), (0.5)))

(b)

Call 4, which is a call to the atomic formula *mem* with input term $b_2 = (2, 3, 4, 5)$, is the second call directly caused by call 2. Due to rule b2.1 applied to call 2, the "call guide" of call 4 is the second element of the "call guide" of call 2: that is (0.7).

Due to rules a2.1.1 and a2.1.3, both applied to call 4, the GL and the output value of call 4 are defined to be $(0.625)$and 4, respectively, as $GIF_{\rm mem}(0.7,(2,3,4,5)) = (0.625,(4))$ (see Section III-A2: $GIF_{\rm mem}$ definition).

Call 5, which is a call to the atomic formula *sml* (see Section II-D) with input terms $y_1 = 2$ and $y_2 = 4$, is the third call directly caused by call 2. Due to rule b2.1 applied to call 2, the "call guide" of call 5 is the third element of the "call guide" of call 2: that is (0.2). Due to rule a1 applied to call 5, the GL of call 5 is an empty list.

Call 6, which is a call to the atomic formula *in* (see Section II-D) with input terms $y_1 = 2$ and $b_3 = 8$, is the fourth call directly caused by call 2. Due to rule b2.2 applied to call 2, the "call guide" of call 6 is $(\ )$, as the "call guide" of call 2 has only three elements. At this point, due to rule a2.1.2 applied to call 6, a random number, say 0.15, is generated. Due to rule a2.1.3 applied to call 6, the GL and the output value of call 6 are defined to be $(0.15)$and 2.9, respectively, as $GIF_{\rm in}(0.15,(2,8)) = (0.15,(2.9))$ (see Section III-A2: $GIF_{\rm in}$ definition).

At this point, due to rule b3 applied to call 2, the GL of call 2 is defined to be the list of the GLs of the directly caused calls, i.e., the calls 3, 4, 5, and 6: that is $((0.5),(0.625),(\ ),(0.15))$.

Call 7, which is a call to the formula F_2 with input terms $c_1 = 2$, $c_2 = 4$ and $c_3 = 2.9$, is the second call directly caused by call 1. Due to rule b2.1 applied to call 1, the "call guide" of call 7 is the second element of the "call guide" of call 1: that is $((0.3),(0.6))$. Due to rule b3 applied to call 7, the GL of call 7 (if not empty) is expected to have two elements, as two calls are directly caused by call 7.

Call 8, which is a call to the formula F_3 with input terms $d_1 = 2$, $d_2 = 4$ and $d_3 = 2.9$, is the first call directly caused by call 7. Due to rule b2.1 applied to call 7, the "call guide" of call 8 is the first element of the "call guide" of call 7: that is (0.3). Due to rule b3 applied to call 8, the GL of call 8 (if not empty) is expected to have two elements, as two calls are directly caused by call 8.

Call 9, which is a call to the atomic formula *sqr* (see Section II-D) with input term $d_2 = 4$, is the first call directly caused by call 8. Due to rule b2.1 applied to call 8, the "call guide" of call 9 is the first element of the "call guide" of call 8: that is, 0.3. Due to rule a1 applied to call 9, the GL of call 9 is an empty list. The output value of call 9, which is assigned to w_1, is 2 (the square root of d_2).

Call 10, which is a call to the atomic formula *mns* (see Section II-D) with input terms $d_3 = 2.9$ and $d_1 = 2$, is the second call directly caused by call 8. Due to rule b2.2 applied to call 8, the "call guide" of call 10 is an empty list. Due to rule a1 applied to call 10, the GL of call 10 is an empty list. The output value of call 10, which is assigned to w_2, is 0.9 (d_3 minus d_1).

At this point, the GL of the call 8 is defined to be an empty list, due to rule b4, as both GLs of the directly caused calls, i.e., the calls 9 and 10, are empty lists. The output value of the call 8, which is assigned to z_1, is (2, 0.9). This is the list of the values assigned to w_1 and w_2, respectively, (see Section II-C: connective *or*), as both calls directly caused by call 8 are confirmed.

Call 11, which is a call to the atomic formula *mem* with input term $z_1 = (2, 0.9)$, is the second call directly caused by call 7. Due to rule b2.1 applied to call 7, the "call guide" of call 11 is the second element of the "call guide" of call 7: that is (0.6). Due to rules a2.1.1 and a2.1.3, both applied to call 11, the GL and the output value of call 11 are defined to be (0.75) and 0.9, respectively, as $GIF_{\rm mem}(0.6,(2,0.9)) = (0.75,(0.9))$ (see Section III-A2: $GIF_{\rm mem}$ definition).

At this point, due to rule b3 applied to call 7, the GL of the call 7 is defined to be the list of the GLs of the directly caused calls, i.e., the calls 8 and 11: that is $((\ ),(0.75))$. Due to the same rule applied to call 1, the GL of call 1 is defined to be the list of the GLs of calls 2 and 7: that is $(((0.5),(0.625),(\ ),(0.15)),((\ ),(0.75)))$.

Note that terminals (0.7) and (0.6) of the "call guide" of call 1 affect the data generation scenario presented here, while if the GL of call 1 is reused as the "call guide" of call 1 then the contents of the fifth, sixth, and the seventh column of Table II(a) will remain intact, while the fourth column will be empty, as the premise for the generation of random values (rule a2.1.2) will never be satisfied: the "call guides" of the calls to multiple confirmation atomic formulas, i.e., the calls 3, 4, 6, and 11, will be identical with the respective GLs, due to genetic mapping rules a2.1.1 and a2.1.3 and the property 2 of the gene interpretation function (see Section III-A2).

Table II(a) presents a data generation scenario where call 1 and all the recursively caused calls are confirmed. Table II(b) differs from Table II(a) in that the random number 0.68, instead of 0.61, is generated at call 3. This leads to a completely different data generation scenario: call 3 assigns to y_1 the value 9 (i.e., the third element of the call's input list) instead of 2, since $GIF_{\rm mem}(0.68,(5.5,2,9)) = (0.833,(9))$ (see Section III-A2: $GIF_{\rm mem}$ definition). This causes disconfirmation of the calls 5 and 6. As a consequence, y_3 is assigned the value *false*. Due to the following substitutions, the variables x_3, c_3, and d_3 are also assigned the value *false*, which disconfirms call 10. Given the syntactical structure of the program, the calls disconfirmed by the latter data generation scenario are the calls 1, 2, 5, 6, and 10. Again, if the GL of call 1 is reused as the "call guide" of call 1, both the data generation scenario and the confirmation conditions for all the recursively caused calls will be the same, while the GL will remain intact.

B. Genetic Operations on Genetic Lists

Genetic operations on GLs resemble specific GP genetic operations since both GLs and computer programs have a tree structure. A significant difference concerns the manipulation of semantic information. A program-node represents a specific function and, therefore, it carries semantic information. A GL node, however, inherits semantic information only from its position in the GL tree, as each node position in the GL tree is mapped to a specific formula represented in the syntax tree. As a consequence, GP genetic operations that alter the position of nodes in a tree structure, can not be used for GLs. Potential loss of semantic information, due to the latter operations, has also been discussed within the framework of GP [31, p. 1093], [11].

The computational system of GENETICAs environment provides three genetic operations on GLs: "homologous crossover" (see also [11], [13, p. 35] ("context preserving crossover"), [31, p. 1093], [32], and [33]), "subtree mutation" (see also GP subtree mutation: [5, pp. 43–44], [13, p. 34], and "Gaussian mutation" (see also [17, p. 225]. The lists resulting from genetic operations on GLs are gene structures which are mapped to new GLs via genetic mappings (see Section III-A3).

GENETICAs homologous crossover resembles the "uniform crossover" described in [34] (pp. 294–295) and [35] (pp. 1163–1164): two parent GL-trees are aligned, the "common region" (i.e., the largest tree structure of which the nodes have homologous nodes in both parent trees: see also [33, p. 105], [34, p. 294]) is identified and different nodes of the "common region" specify pairs of homologous nodes of the parent trees. Paired nodes are swapped. GENETICAs crossover is not necessarily uniform, since the expected depth of the nodes to be swapped in the parent trees can be regulated by a probability function. Within a GL, high-depth nodes, resulting in small building blocks, are usually associated to low-level semantic properties, whereas low-depth nodes, resulting in larger building blocks, are usually associated to higher-level semantic properties. Crossover depth control has been used in GP applications [5, p. 43], while the influence of the depth of a genotypic change on fitness has been discussed in [36].

In GENETICAs subtree mutation random nodes in a GL-tree are substituted by empty lists. The missing parts of the resultant list are randomly redeveloped in a new GL via the genetic mapping (see Section III-A3). Like in crossover, the expected depth of the mutated node can be regulated by a probability function.

GENETICAs Gaussian mutation changes real values within a GLs terminal nodes. The probability of each new value is defined by a normal distribution maximized at the original value. Gaussian mutation has been used also in GP [13, p. 35] ("mutating constants at random") to enhance optimization of numerical parameters [37], in evolutionary programming to enhance local search [38, pp. 754–755] (Solis and Wets method) and in continuous global optimization [39, p. 932].

C. Fitness of a Formula Call

1) Introduction to the Fitness of a Formula Call:

The goal in confirmation problems is to confirm the root formula, whereas in optimization problems the goal is triple: 1) confirm the formula that produces a potential solution (see Section II-C, connective *opt*); 2) confirm the formula that evaluates the potential solution; and 3) find a solution which maximizes the solution evaluation magnitude, in this order of importance. Each potential call to the root formula, namely each potential data generation scenario that may occur given the input values for the root formula, has a specific degree of success with respect to the aforementioned goals. The fitness of the call is a formal expression of what we have called "degree of success." A critical issue, which is investigated in the remainder of this paragraph, is to specify the notion "degree of success" with respect to the confirmation goals.

Consider the call tree of a call to a nonatomic formula F. Substitute each nonterminal node n in the call tree with the connective that constructs the formula whose call is represented by n. Substitute each terminal node t in the call tree with the truth value of the call represented by t. The resulting tree will be referred to as the "confirmation tree." For instance, the confirmation tree derived from the call tree presented in Fig. 2, with respect to the genetic mapping presented in Table II(b), is presented in Fig. 4.

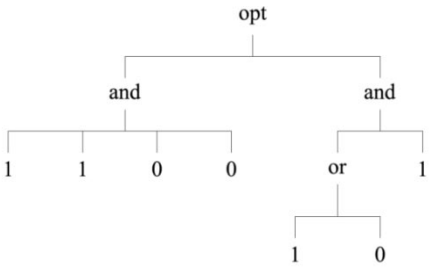


Fig. 4. Confirmation tree of the program presented in Table I, with respect to the genetic mapping presented in Table II(b). The values 0 and 1, appearing as terminals, represent, respectively, disconfirmation and confirmation of the homologous atomic formula calls appearing in the call tree presented in Fig. 2.

Let c be an arbitrary call represented in the call tree. Given the logical structure represented by the respective confirmation tree, c can be assigned a value denoting the minimum number of the first-order atomic formula calls of which the confirmation state should change in order to confirm c. This value, which can be viewed as a measure of "distance from confirmation" for c, can be recursively computed given the connectives and the truth values represented in the confirmation tree. For instance an *and* node of the confirmation tree has "distance from confirmation" the sum of the "distances from confirmation" of its child nodes, while an *or* node has "distance from confirmation" the minimum "distance from confirmation" of its child nodes. Intuitively, the "distance from confirmation" of a call is a negative expression of the "degree of success" of the call with respect to the goal of confirmation and could be considered as a measure of fitness of the call.

The aforementioned approach to fitness resembles approaches adopted in some evolutionary methods that consider statistical properties of the portions of data positively and negatively affecting confirmation, such as "conditional probability distribution" [2, p. 189], "gain ratio" [23, p. 147], "fact separation" [22, p. 4], "step-wise" and "exhaustive" evaluation [40, p. 1213], "positive and negative examples" [18, p. 643], [41], etc. It also resembles approaches related to the field of model based representation of knowledge. A "model" for a logical expression is a complete truth assignment for the propositional variables that appear in the expression [42, p. 131]. In [43], it has been argued that the ratio of the models that confirm premises and conclusion over the models that confirm premises could serve as validity criteria for inference in probabilistic and nonmonotonic reasoning. Given the input terms, a GENETICA formula, labeled as F, represents a logical statement. The truth assignment represented by the terminals of the confirmation tree of an F call could be viewed as a model confirming premises for F. Then, each F call can be represented as a model. Let C be the set of all the possible F calls, given the input terms, and C_T the set of the confirmed F calls. Let C_{d1} and C_{d2} be the sets of the F calls having "distances from confirmation" less or

TABLE III
DEFINITION OF THE FITNESS LISTS OF THE CALLS TO NONATOMIC FORMULAE. LET (d_k, d'_k) $(k = 1, \ldots n)$ BE THE FITNESS LIST OF THE kTH CALL DIRECTLY CAUSED BY A CALL TO A NONATOMIC FORMULA F, WHERE n IS THE NUMBER OF THE CALLS DIRECTLY CAUSED BY THE F CALL. THE FITNESS LIST OF THE F CALL IS PRESENTED WITH RESPECT TO THE CONNECTIVE IN THE DEFINITION OF F

Connective	Fitness List
and, app	$(d_1 + \ldots + d_n,\ \min(d'_1, \ldots d'_n))$
or, chs	$(\min(d_1, \ldots d_n),\ d'_1 + \ldots + d'_n)$
not	$(d'_1,\ d_1)$
rec	$(\min(d_1, d_3),\ \min(d'_2, d'_3))$ if $d_1 \neq 0$ and $d_2 = 0$ $(d_1,\ d'_1)$ if $d_1 = 0$ or $d_2 \neq 0$
opt	$(d_1 + c + \frac{1}{1+Q}, d'_1)$ where: $c = 1$ if $d_1 \neq 0$ $c = 1 - \frac{1}{1+d_2}$ if $d_1 = 0$ $Q = 0$ if $c \neq 0$ Q equals the quantity to be maximized if $c = 0$

equal to d_1 and d_2, respectively, where $d_2 < d_1$. The relation $C_T \subseteq C_{d2} \subset C_{d1} \subseteq C$ holds. Due to this relation, the ratio of the confirmed F calls (i.e., the elements of C_T) is higher in C_{d2} than it is in C_{d1}. As a consequence, C_{d2} is better than C_{d1}, with respect to the validity criteria proposed in [43].

Note that when a formula F is called, similar calls to the same first-order atomic formula, each one affecting the "distance" from F confirmation, may be encountered in the call tree. As a consequence, syntactically different formulas may result in different "distances" even if they are semantically equivalent. For instance, if F_A is a disconfirmed first-order atomic formula then the "distance from confirmation" of a F_A call is one, whereas the "distance from confirmation" of the semantically equivalent (F_A *and* F_A) call is two. This is not a problem in GENETICA since, unlike GP, the syntax is fixed and no different GENETICA programs have to be compared. Besides, some user control of the fitness becomes possible through the syntax; e.g., the expression (F_A *and* F_A *and* F_B)—although not recommended since it can be replaced by an equal fitness expression that does not indicate a double F_A call—means that the confirmation of F_A is twice as important as the confirmation of F_B. Syntactic control of the fitness is presented in Section III-C4.

2) Definition of the Fitness of a Formula Call:

We define two magnitudes characteristic of a call to a formula F (therefore, characteristic of the GL of the F call): "distance from confirmation," denoted by d, and "distance from disconfirmation," denoted by d'. The latter magnitude represents "distance from confirmation" of the negation of F. The list (d, d') is defined to be the fitness list of the F call. Specifically, we define (0,1) and (1,0) to be the fitness list of a first-order atomic formula call in the case of confirmation and disconfirmation, respectively. The fitness list of a call to a nonatomic formula is defined in Table III. The fitness list of a call to a high-order atomic formula is defined to be equal to the fitness list of the forced call (see Section II-A).

It is easy to prove by induction that the "distance from confirmation" of a call to a formula F constructed with one of the connectives *and*, *or*, *not*, *app*, and *chs*, always represents the minimum number of the recursively caused first-order atomic formula calls whose confirmation state should change in order to confirm F.

An explanation should be given in the case of recursion (connective *rec*). Let (C_1, C_2, C_3) be the list of the possible calls directly caused by the F call, i.e., C_1 is the call to the termination condition (see Section II-C, connective *rec*), C_2 is the call to the formula that produces arguments of the recursive F call, and C_3 is the recursive F call. The equality $T_F = (T_1 \text{ or } (T_2 \text{ and } T_3))$ holds, where T_F denotes the truth value of the initial F call and $T_i (i = 1, 2, 3)$ denotes the truth value of the call C_i. Due to the correspondence of logical operations to operations on "distances from confirmation," introduced for the connectives *and* and *or* in Table III, the fitness list of the F call should be $(min\,(d_1,\ d_2 + d_3), d'_1 + \min\,(d'_2,\ d'_3))$. Note that, due to the definition of the connective *rec* in Section II-C, C_2 is realized if and only if C_1 is disconfirmed, while C_3 is realized if and only if C_2 is both realized and confirmed. By setting the "distance from confirmation" equal to infinity (maximum possible value) and the "distance from disconfirmation" equal to zero for the nonrealized calls, the latter definition of the fitness list becomes equivalent to the definition given in Table III.

Note that in optimization problems, i.e., problems where the root formula is constructed with *opt*, the "distance from confirmation" of a root formula call is artificially increased by two quantities: the first, named c in Table III, depends on the "distance from confirmation" of the solution evaluation formula, while the second, represented by $1/(1 + Q)$ in Table III, depends on the value of the solution evaluation magnitude. Due to this technique, the objective of minimizing the value of the first element of the fitness list respects the priorities of the optimization objectives, since this value lies in the real intervals $(2, \infty)$, (1,2), and (0,1), respectively, in the cases of: 1) disconfirmation

TABLE IV
FITNESS EVALUATION FOR THE DATA GENERATION SCENARIOS PRESENTED IN TABLE II(a) AND (b) IS, RESPECTIVELY, PRESENTED IN TABLE IV(a) AND (b). IN TABLE IV(a) AND (b), f_i, d_i, AND $d'_i (i = 1, \ldots 11)$ DENOTE, RESPECTIVELY, THE FITNESS LIST, THE "DISTANCE FROM CONFIRMATION," AND THE "DISTANCE FROM DISCONFIRMATION" OF THE iTH CALL IN THE CALL TREE (FIG. 2) OF THE PROGRAM PRESENTED IN TABLE I

$(d_3, d'_3) = (d_4, d'_4) = (d_5, d'_5) = (d_6, d'_6) = (d_9, d'_9) = (d_{10}, d'_{10}) = (d_{11}, d'_{11}) = (0, 1)$

$f_2 = (d_2, d'_2) = (d_3 + d_4 + d_5 + d_6, \ \min(d'_3, d'_4, d'_5, d'_6)) = (0 + 0 + 0 + 0, \ \min(1, 1, 1, 1)) = (0, 1)$

$f_8 = (d_8, d'_8) = (\min(d_9, d_{10}), \ d'_9 + d'_{10}) = (\min(0, 0), \ 1 + 1) = (0, 2)$

$f_7 = (d_7, d'_7) = (d_8 + d_{11}, \ \min(d'_8, d'_{11})) = (0 + 0, \ \min(2, 1)) = (0, 1)$

$f_1 = (d_1, d'_1) = \left(d_2 + c + \frac{1}{1+Q}, \ d'_2\right)$

$d_2 = 0 \Rightarrow c = 1 - \frac{1}{1+d_7} = 1 - \frac{1}{1+0} = 0$

$c = 0 \Rightarrow Q = 0.9$ (the value of x_4 in TABLE II(a)

$\Rightarrow \quad f_1 = \left(0 + 0 + \frac{1}{1+0.9}, \ 1\right) = (0.526, \ 1)$

(a)

$(d_3, d'_3) = (d_4, d'_4) = (d_9, d'_9) = (d_{11}, d'_{11}) = (0, 1), \qquad (d_5, d'_5) = (d_6, d'_6) = (d_{10}, d'_{10}) = (1, 0)$

$f_2 = (d_2, d'_2) = (d_3 + d_4 + d_5 + d_6, \min(d'_3, d'_4, d'_5, d'_6)) = (0 + 0 + 1 + 1, \min(1, 1, 0, 0)) = (2, 0)$

$f_8 = (d_8, d'_8) = (\min(d_9, d_{10}), d'_9 + d'_{10}) = (\min(0, 1), 1 + 0) = (0, 1)$

$f_7 = (d_7, d'_7) = (d_8 + d_{11}, \min(d'_8, d'_{11})) = (0 + 0, \min(1, 1)) = (0, 1)$

$f_1 = (d_1, d'_1) = \left(d_2 + c + \frac{1}{1+Q}, \ d'_2\right)$

$d_2 = 2 \neq 0 \Rightarrow c = 1$

$c = 1 \neq 0 \Rightarrow Q = 0$

$\Rightarrow \quad f_1 = \left(2 + 1 + \frac{1}{1+0}, \ 0\right) = (4, 0)$

(b)

of the solution production formula; 2) confirmation of the solution production formula and disconfirmation of the solution evaluation formula; and 3) confirmation of both the solution production formula and the solution evaluation formula.

The fitness evaluation procedure is illustrated in Tables IV(a) and (b) with respect to the data generation scenarios presented in Tables II(a) and (b), respectively. The first line of both Table IV(a) and (b) presents the fitness lists of the atomic formula calls, while the remaining lines present the fitness calculation for the nonatomic formula calls. The fitness calculation for a call to a nonatomic formula F is based on the substitution of the fitness values (i.e., the elements of the fitness list) by the expressions specified in Table III for the connective constructing F. For instance, the fitness list $f_2 = (d_2, d'_2)$ of call 2 equals $(d_3 + d_4 + d_5 + d_6, \min(d'_3, d'_4, d'_5, d'_6))$ because of the following.

- Call 2 is a call to F_1 (see Fig. 2), which is a nonatomic formula constructed with *and* (see Table I); consequently, the "distance from confirmation" (first element of the fitness list) and the "distance from disconfirmation" (second element of the fitness list) of call 2 are, respectively, equal to the sum of the "distances from confirmation" and the minimum "distance from disconfirmation" of the directly caused calls (see Table III).
- The calls directly caused by call 2 are the calls 3, 4, 5, and 6 (see Fig. 2).

3) "Fuzzy" Extension to the Definition of Fitness:

GENETICA includes a nonconstructive atomic formula named *eq*, which stands for equality. Input terms for *eq* are two arbitrary GENETICAs terms. *eq* is confirmed if and only if the input terms are equal. Due to the definitions given in the previous paragraph, the "distance from confirmation" of a call to *eq* should be either one or zero, since *eq* is a first-order atomic formula. We extend this definition by defining the "distance from confirmation" d_{eq} of a call to *eq* as a function of the input terms of the call, with values in the real interval [0, 1). This function, which represents a measure of structural and quantitative difference between the input terms, is formally presented in Section 3.2.1.b in the file Genetica_Documentation.pdf, available in www.genetica-informatics.org. If the input terms are equal then $d_{eq} = 0$ and *eq* is confirmed. d_{eq} could be interpreted as a fuzzy truth value for *eq*.

Given the definition of the fitness lists for nonatomic formula calls, presented in Table III, the extended definition of "distance

from confirmation," introduced here for the atomic formula *eq*, results in a generalization of the notion of "distance from confirmation" given in Section III-C1. Specifically, if C_F is a call to a formula F then the smallest set S of the recursively caused first-order atomic formula calls whose confirmation state should change in order to confirm C_F, should be viewed as a fuzzy set. S includes all the first-order atomic formula calls recursively caused by C_F. Each element C_A of S has "membership value":

- the "distance from confirmation" of C_A if confirmation of C_A contributes positively to confirmation of C_F;
- the "distance from disconfirmation" of C_A if confirmation of C_A contributes negatively to confirmation of C_F.

The size of S, which represents the "distance from confirmation" of C_F, is viewed as the sum of the "membership values" of its elements.

4) Syntactic Control of the Fitness:

GENETICA provides direct control of the magnitude which is to be maximized in optimization problems, through the definition of the solution evaluation formula (see Section II-C, connective *opt*). As mentioned in Section III-C1, control is also possible on "distance from confirmation" through the syntax. The latter kind of control is enhanced by both a formula property named "always true" and specific atomic formulas provided by GENETICA. "Always true" formulas are forced to be confirmed while the fitness list of a call to an "always true" formula is defined to be the list (0, ∞). Several atomic formulas are defined to be "always true," while any formula can be converted to "always true" if called through an "always true" high-order atomic formula. The fitness list (0, ∞) constitutes a neutral element with respect to the operation that computes the fitness list in the case of a formula F constructed with the connective *and* (see Table III). As a consequence, the referenced in the definition of F formulas that affect the fitness of a F call can be determined through the syntax.

Syntactic control of the fitness is further enhanced both by an "always true" high-order atomic formula, returning the fitness list of the forced call as an output value, and by a single confirmation atomic formula, of which call's the fitness list is explicitly defined by an input term (the possibility to define explicitly the fitness list extends the definitions given in Section III-C2). As a consequence, GENETICA provides access to the fitness list of any formula call. The fitness list can be both processed and defined as a term.

D. Computational Process

The computational process can start when both a GENETICA program and the input values for the root formula within the program have been defined. It includes the phase of the population initialization and the phase of the execution of computational cycles. Various parameters of the computational system can be adjusted by the user.

1) Population Initialization:

An initially empty list P is defined and calls to the root formula are made. For each call, a GL is constructed via a genetic mapping having the empty list as a "call guide" (see Section III-A3), while a triplet (g, f, s) is defined, where g is the call's GL, f is the call's fitness list, and s is a magnitude having initial value 1. The triplet will be referred to as a "species," while s will be referred to as the "species size." Informally, a "species" represents a set of identical individuals, while the "species size" represents the size of the set. If the GL of a new "species" is identical to the GL of a "species" already existing in P, then the "species size" of the existing "species" increases by one, while the size of P remains intact. Otherwise, the new "species" is appended to P. As a consequence, no "species" having identical GLs exist in P. The elements of P are ordered by fitness, with the best-fitness element at the end. P will be referred to as the "population."

2) Execution of Computational Cycles:

In each computational cycle:

a) The computational system creates three lists, referred to as the "selection lists," each one associated to a different genetic operator. The elements of the "selection lists" are "species" of the population. The selection of the "species" forming a "selection list" is done randomly with respect to a probability function, referred to as the "selection function," applied to the population. The selection probability of each element of the population depends only on its order in the population and the "selection function."

b) The GL of each "species," within each "selection list," is transformed by the respective genetic operator to a gene structure. Each gene structure is used as the "call guide" of a root formula call and is then mapped to a new GL via the genetic mapping process (see Section III-A3). The new GL defines a new "species" referred to as a "test."

c) For each "test" that includes a GL similar to the GL included in a "species" of the population, the "species size" of the latter "species" increases by one. High-fitness "tests" from the remaining ones form an "innovation list," while low-fitness "species" of the population form an "extinction list." The elements of the former list substitute the elements of the latter list in the population.

The computational process ends either with the finding of an acceptable solution or with the transition of the population to a state where no "species" of better fitness can be produced.

3) Parameters of the Computational System:

The probability of finding an acceptable solution and the computation time are roughly proportional to the total number of tests performed. The search dynamics depend on parameters of the computational system, which can be adjusted before the beginning of or during the computational process, affecting both the expected computation time and the search localization. These parameters are reported here (a detailed presentation is included in the file Genetica_Documentation.pdf, available in www.genetica-informatics.org):

a) size of the population;

b) number of tests per computational cycle;

c) sizes of the "selection lists"; these sizes determine the computational method, e.g., a relative increase of the size of the "selection list" associated to the crossover, the subtree mutation or the Gaussian mutation operation would turn the computational method, respectively, to a GA, a stochastic search method or an evolutionary strategy;

d) form of the "selection functions";

e) definition of the "innovation list";
f) definition of the "extinction list";
g) desirable percentage, number or probability of the terminal nodes of a GL affected by each genetic operation;
h) distribution of the probability of a mutated node with respect to the depth in a GL, in subtree mutations and crossovers;
i) standard deviation of the probability density function in Gaussian mutations;
j) contribution of the "species size" to the selection probability of a "species": the selection probability of a "species" can be multiplied by s^x, where s is the "species size" and x is a real variable.

IV. Development of an Architectural Design Language in GENETICA

A. Development of Domain Specific Languages and Implementation of GP in GENETICA

A general and effective approach to domain specific problem solving is to describe both the problem and the available knowledge about 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 computer's machine language. GENETICA, which has the expressive capabilities required, supports the development of high-level formal languages. The benefit is that GENETICAs evolutionary computational system becomes available in the latter languages. GP versions of high-level languages become possible if programs in the latter languages are treated as evolving data structures in GENETICA.

Let L be a high-level formal language which is to be developed in GENETICA. Let the prefixes "G" and "L" denote an element of GENETICA and an element of L, respectively. The development of L could be based on the following tenets.

- L-terms can be constructed as potentially composite G-terms.
- Atomic L-formulas can be constructed as potentially nonatomic G-formulas.
- L-connectives can be constructed as nonatomic high-order G-formulas having as input the L-formulas connected by the L-connective.

Specifically, L can be implemented as a G-program, labeled as P_G, where the input vector for the root G-formula in P_G includes an arbitrary L-program, labeled as P_L, and an input vector value for the root L-formula in P_L, while the output vector value for the latter formula, resulting from the execution of P_L, is included in the output of the root G-formula. Since the execution of P_L is an interpretation of the execution of P_G, any data generation scenario in P_L is an interpretation of a corresponding data generation scenario in P_G, while any confirmation or optimization goal in P_L is an interpretation of a corresponding goal in P_G. As a consequence, the evolutionary computational system provided by GENETICAs environment becomes available in P_L through P_G.

P_L can be modified or even developed during the execution of P_G, since it is treated as a data structure in P_G. Specifically, missing L-formulas (i.e., L-formulas referenced but not defined in P_L) can be constructed at run time, while redundant L-formulas (i.e., L-formulas defined but not referenced in P_L) can be referenced by L-formulas emerging at run time. Data processing in P_G, resulting in the development of P_L, would be controlled by the computational system within GENETICAs environment. This makes possible a general GP method, where domain specific applications could be developed, since domain specific knowledge can be encoded in incomplete L-programs (i.e., L-programs with missing L-formulas) 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 classical 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; consequently, it can be evolved as part of the solution. GENETICA-based GP is characterized by a genotype-to-phenotype mapping since both programs and data structures (phenotypes) are ultimately defined by evolving GLs (genotypes). This brings it closer to nonclassical GP approaches such as developmental GP [44]–[46] and grammatical evolution [47], [48].

A general method for developing domain specific and GP languages in GENETICA, with respect to the principles presented here, is formally presented in the file Genetica-based_Languages.pdf, available in www.genetica-informatics.org.

B. Architectural Design Problem Solving and the Language G-CAD

The field of problem solving in architectural design is very demanding both in expressive and computational resources. Early methods in this field date back to the 70s [49], [50]. Methods developed in the 80s followed the logic programming paradigm [51]–[53], while methods developed in the 90s followed the evolutionary paradigm [54]–[57]. Some of the latter methods deal with creative designing [58]–[60]. Since problems in architectural design are not only hard to solve but even difficult to formulate, design methods have been tested in either very restrictive or very abstract cases. It is a challenge to develop a general architectural design language effective in real scale problems.

The language G-CAD, which stands for "GENETICA-CAD," is specialized to problem solving in the domain of architectural design. G-CAD has been developed in GENETICA, following the principles presented in Section IV-A. A formal presentation of the G-CADs development method is included in Section II of the file Genetica-based_Languages.pdf, available in www.genetica-informatics.org.

G-CAD is not a GP language: solutions are not programs, but evolvable data structures governed by fixed G-CAD code. The basic specifications of G-CAD derive from object-oriented open-ended design enabled with general knowledge representation [61] and from creative evolutionary design based on de-

velopmental procedures and embryogenies [59, pp. 384–385], [60], [62]. These are summarized here.

- Encapsulate design knowledge in objects having both internal properties and communication channels.
- Allow general knowledge representation within objects.
- Enable objects with context sensitive behavior based on object-to-object interactions.
- Allow both top-down and bottom-up interactions in the object hierarchy.
- Allow behavior-driven selfassembly of objects.

A brief description of G-CAD is given in the next paragraphs. A detailed presentation is included in the file G-CAD_documentation.pdf, available in www.genetica-informatics.org.

C. Description of the Language G-CAD

1) G-CAD Terms and Objects:

G-CAD shares GENETICAs terms. Additional composite terms in G-CAD, which are represented as lists are described here.

a) A "property list" is a list of pairs, where each pair has the form (*property_name*, *property_value*).

b) A "spatial map" represents a configuration of spatial attributes on the Cartesian plane by mapping regions of the Cartesian plane to "property lists," where properties represent spatial attributes.

c) A "query" represents a logical statement on a "property list." Given a "spatial map," a "query" specifies on the Cartesian plane the region whose properties satisfy the "query."

d) An OCS, which stands for "object's coordinate system," is a local coordinate system specifying the position, orientation and mirroring condition of a design object on the Cartesian plane. When an object is mirrored the X axis of its OCS is reversed, while the Y axis remains intact.

e) A "primitive" is an elementary design object, geometrically represented as a rectangle. The "primitive's" formal representation includes a "property list" where the geometric features, the OCS and the spatial attributes of the "primitive" are registered. A "primitive" can be "imprinted" to a "spatial map." Due to the imprinting, the "spatial map" is revised to assign to the region occupied by the "primitive" the spatial attributes of the "primitive," while assignments already represented in the "spatial map" remain intact.

f) Tree structured design objects the terminals of which are "primitives." Each tree structured object has its own "property list," where the object's OCS is always included. All the nodes of the tree structure constitute subobjects. A tree structured object can be either an original object, referred to as a "unit," or a replica, referred to as an "instance." An "instance" points to a "unit," while represents a copy of this "unit" subjected to the position, rotation, and mirroring transformation that takes the "unit's" OCS to the "instance's" OCS. Any tree structured object can be "imprinted" to a "spatial map." Imprinting the object is equivalent to imprinting the object's terminal "primitives."

g) A "positioning list" includes specifications for the positioning of a design object on the Cartesian plane, given a "spatial map." Specifically, a "positioning list" includes two "queries" ; the application of these "queries" on the "spatial map," respectively, specifies a "positioning region" and an "adjacent region" on the Cartesian plane. The object should be placed having the X axis of its OCS at the boundary of these regions and the Y axis showing at the side of the "positioning region," while it should respect orientation and mirroring constraints also included in the "positioning list."

2) G-CAD Formulas:

a) General features of the G-CAD formulas: G-CAD formulas have distinct input and output terms, following GENETICAs conventions. A class of G-CAD formulas, referred to as "primary formulas," create design objects, i.e., "primitives," "units" or "instances." The remaining G-CAD formulas, referred to as "secondary formulas," perform either arithmetic or logical processing producing terms not representing design objects.

The input of a formula that creates a design object always includes a "spatial map" representing the environment where the object is to be created, while the output includes the "spatial map" resulting after the object's imprinting.

"Primitives" and "instances" are created by atomic formulas, whereas "units" are created by nonatomic formulas. The tree structure of a "unit" reflects the "unit's" creation formula call tree.

b) Atomic G-CAD formulas: The atomic formulas in G-CAD include the formula creating a "primitive," the formula creating an "instance," a high-order formula, and the atomic secondary formulas. These are briefly described here.

b1) The input for the formula creating a "primitive" includes a "spatial map," the "positioning list," the "property list" and dimension specifications for the "primitive" to be created. Within the "property list," a specific property named APA, which stands for "acceptable positioning area," is a region definition "query." The formula is confirmed if and only if the "primitive" is included in the Cartesian plane's region defined by the application of the APA "query" to the input "spatial map."

b2) The input for the formula creating an "instance" includes a "spatial map," a "unit," and also the "positioning list" and the "property list" of the "instance" to be created. Within the "property list," a specific property named IUS, which stands for "instanced unit selection," is a "unit" selection "query." The "unit" to be instanced is the first "unit" of which the "property list" satisfies the IUS "query." This first "unit" is encountered by depth-first search in the input "unit's" structure. The formula is confirmed if and only if both the IUS "query" is confirmed and each terminal "primitive" of the "instance" is included in the input "spatial map's" region specified by the "primitive's" APA property.

b3) The input for the high-order formula includes the name F_N and an input vector value V of a G-CAD formula,

while the output includes the output of the formula named F_N called with V.

b4) G-CAD secondary atomic formulas include the basic arithmetic and set-theoretic operations and relations, as well as list processing functions. A class of G-CAD secondary atomic formulas perform arithmetic and logic calculations on either "spatial maps" or design objects.

c) Nonatomic G-CAD Formulas: The definition of a nonatomic formula is a list having the form (F_N, *con*, S_{inp}, P, Ref_{lst}, S_{out}), where F_N is the name of the formula, *con* is a connective, S_{inp} is the input section as a list of the names of the input terms, P is the property section as the "property list" of the object created by the formula, Ref_{lst} is the reference section as a list of references to formulas which are called when the formula named F_N is called, and S_{out} is the output section as the list of the names of the output terms. A formula reference is a list having the form (Ref_N, Ref_{Sin}, Ref_{Sout}), where Ref_N is the name of the referenced formula, while Ref_{Sin} is the input section and Ref_{Sout} is the output section of the reference as lists of the names of the referenced formula's input and output terms, respectively. Any symbol in the input section of a formula reference should alternatively: 1) be included in S_{inp}; 2) constitute a property name in P; or 3) be included in the output section of a previous formula reference in Ref_{lst}. Any symbol in S_{out} should be included in the output section of a formula reference in Ref_{lst}. A G-CAD program is a list of definitions of nonatomic formulas.

The nonatomic formulas are constructed with the G-CAD connectives and_{G-CAD}, or_{G-CAD}, all_{G-CAD}, one_{G-CAD} and rec_{G-CAD}. The connectives and_{G-CAD}, all_{G-CAD} and rec_{G-CAD} are similar with the GENETICA connectives *and*, *app* and *rec*, respectively. or_{G-CAD} differs from GENETICAs *or* in that only one referenced formula, randomly selected from the reference section of the or_{G-CAD} formula, is called. one_{G-CAD}, which is the G-CAD existential quantifier, differs from GENETICAs *chs* in that the single referenced formula is called once for a randomly selected element of the list of which the elements represent the term referred to by the quantifier.

3) An Integrated GENETICA and G-CAD Design Method:

A design problem formulation in G-CAD includes a G-CAD program, labeled as P_L, and the P_L's input, both represented as GENETICA lists. These lists constitute the input of a GENETICA program, labeled as $P_{G_{G-CAD}}$, that implements G-CAD based on the principles presented in Section IV-A. P_L's input includes:

a) "unit," referred to as the "world unit," which consists of standard (nonevolvable) subunits;
b) descriptions of "units" to be evolved, where each description includes:
   - a "spatial map" that represents the environment where the "unit" is to be developed;
   - the name F_N of the "unit's" definition formula, where the latter formula is included in 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 P_L execution, which reflects the $P_{G_{G-CAD}}$ 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 subunit 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 P_L, is considered as the solution of the design problem. The solution "unit's" output may include a magnitude to be maximized. The output of $P_{G_{G-CAD}}$, obtained from the execution of P_L, includes the developed "world unit" and the solution "unit's" output.

A CAD interface provides both graphic and text tools for the definition of input data for P_L, while it presents the solution as an architectural drawing. The problem formulation and solution procedure include the following steps.

1) Write the P_L code. Use the CAD interface to define the input for P_L. The interface automatically formulates both P_L and P_L's input as GENETICA lists.
2) Use GENETICAs environment to load $P_{G_{G-CAD}}$ and declare the lists formulated in the previous step as the input for $P_{G_{G-CAD}}$.
3) Adjust the computational parameters within GENETICAs environment and activate the computational process.
4) At the end of the computational process, use the CAD interface to convert the solution (i.e., the evolved $P_{G_{G-CAD}}$'s output which has the form of a GENETICA list) to an architectural drawing.

Random selections performed during P_L's execution specify the following:

a) position, orientation, and mirroring condition of each "primitive" or "instance" object within an evolving "unit," given both the "spatial map," where the object is to be imprinted and the object's "positioning list;"
b) dimensions of each "primitive" of an evolving "unit," given the "primitive's" positioning, the region, where the "primitive" should be included, and the "primitive's" dimensioning specifications;
c) output of the multiple confirmation G-CAD secondary formulas;
d) the referenced G-CAD formula, that is to be called in the case of a or_{G-CAD} connective;
e) input of the referenced G-CAD formula in the case of a one_{G-CAD} connective.

All the aforementioned random selections evolve according to GENETICAs computational system, as they are realized via the multiple confirmation GENETICA formulas appearing in $P_{G_{G-CAD}}$.

V. Case Study: Formulation and Solution of an Architectural Design Problem

A. Problem Formulation

1) Informal Problem Statement:

Design a small hotel's floorplan, with eight identical apartments, a corridor, and a stairwell. An apartment consists of a main room and a WC, while the main room should be adjacent to a corridor. The dimensions of the apartment's rooms should satisfy specific geometric constraints. The apartments can be combined in specific ways, while no apartment-to-apartment overlapping is allowed. The stairwell should be adjacent to the corridor without overlapping an apartment. Specific objects representing openings, furniture or bathroom fittings should be placed in an apartment. These objects should satisfy both object-to-object and object-to-apartment geometric relations. The area of the built space, i.e., the space occupied by either the apartments or the corridor, should be minimized.

2) Formal Problem Statement: The formal problem statement is outlined here. A detailed presentation is included in the file G-CAD_documentation.pdf, available in www.genetica-informatics.org.

Formally, the problem statement is the input of the program $P_{G_{G-CAD}}$, introduced in Section IV-C3. According to the definitions given in the same section, the input includes a G-CAD program, a "world unit" having standard (nonevolvable) subunits, and a list of descriptions of evolvable "units" defined in the G-CAD program.

a) Standard "units": The standard "units" (Fig. 5), included in the "world unit" of the hotel application, represent design objects having fixed structure and dimensions. These objects constitute standard elements of the desirable solution, such as openings, furniture, bathroom fittings, and a stairwell. Each standard "unit" consists of "primitives" having fixed dimensions and standard positioning in the "unit's" OCS. Each "primitive" represents a spatial attribute which is included in the "primitive's" "property list." Spatial attributes denote "physical space," i.e., space occupied by the physical object, "functional space," i.e., space that should be free to allow the use of the physical object, "internal space" and "external space" viewed as regions separated by an opening, etc. When the standard "units" are imprinted to a "spatial map" then the overlapping conditions of spatial attributes are determined by the *APA* properties of the "primitives" (see Section IV-C2b1).

The idea to consider design objects as assemblies of "physical" and "functional" spaces, where the functional spaces can overlap each other while no other overlapping is allowed, has been used in a PROLOG architectural design method named "CADRAM" [63]. This idea is generalized in G-CAD since arbitrary spatial attributes, having different overlapping relations, can be assigned to design objects.

b) Evolvable "units": The basic evolvable "units" of the hotel application are the "unit" *Type*, which represents the floorplan of an apartment type perceived as an assembly of two rooms, the "unit" *Type_Full*, which represents the floorplan of the same apartment type having openings, furniture, and bathroom fittings, and the "unit" *Hotel*, which represents the hotel's floorplan, i.e., the potential solution of the design problem. The "unit" *Type_Full* includes an "instance" of the "unit" *Type* and various subobjects that represent openings, furniture and bathroom fittings. The "unit" *Hotel* includes apartments represented by *Type_Full* "instances" and various subobjects formulating both a corridor that connects the apartments and a stairwell. The creation procedure of an evolvable "unit" is encoded in the "unit's" creation formula. The formulas creating the basic evolvable "units" of the hotel application are constructed with and_{G-CAD} (see Section IV-C2c). Consequently, all the referenced formulas are called, creating subobjects which are successively imprinted to the "spatial map" included in the "unit's" description (see Section IV-C3b). The creation procedure of the basic evolvable "units" is outlined below. Consider that each "unit" is named by the respective "unit" creation formula.

b1) The "unit" Type

The *Type* "unit" creation formula creates two "primitives," allocated as shown in Fig. 6, having the spatial attributes *MR* and *WC*, which stand for "Main Room" and "WC," respectively. *Type* is to be attached to a fixed width region having the spatial attribute *COR* which stands for "Corridor." The positioning of *Type* in relation to this region is presented in Fig. 6, where the dimension *C* represents the region's fixed width. The dimensions A, B, C, D, and E, presented in Fig. 6, satisfy the constraints

$$A_{\min} \le A \le A_{\max}$$
$$B = B_{\max} - (A - A_{\min}) \cdot \frac{B_{\max} - B_{\min}}{A_{\max} - A_{\min}}$$
$$D = A + B - C$$
$$WCA_{\min} \le WCA \le WCA_{\max}$$

where WCA is the area of the WC region, i.e., the product $B \cdot E$, while the subscripts min and max denote minimum and maximum acceptable values, respectively. These constraints are constructed via secondary formulas referenced by the *Type* "unit" creation formula. The variables $A_{\min}$, $A_{\max}$, $B_{\min}$, $B_{\max}$, C, $WCA_{\min}$, and $WCA_{\max}$ represent the input terms of the *Type* "unit" creation formula. They are assigned the values 350, 390, 170, 230, 130, 35 000, and 40 000, respectively, representing centimeters.

b2) The "unit" *Type_Full*

The *Type_Full* "unit" creation formula creates an "instance" of the "unit" *Type*, various "instances" of the standard "units" (Fig. 5) and two subunits representing furniture. The standard "units'" "instances" include two *Door* "instances" (representing the entrance door and the WC door, respectively), two *Bed* "instances," and one "instance" of each one of the "units" *Door_Balcony*, *WC_Window*, *Toilet*, *Basin* and *Shower*. Both *Type_Full*'s subunits, labeled as *Sitting_Group* and *Wardrobe*, respectively, are created by formulas referenced by the *Type_Full* "unit" creation formula. The *Sitting_Group* "unit" creation formula, which is constructed with or_{G-CAD} (see Section IV-C2c), creates either a *Sitting_Group_1* or a *Sitting_Group_2* "instance" (see Fig. 5), while the *Wardrobe* "unit" creation formula, which is constructed with and_{G-CAD}, creates two "primitives"

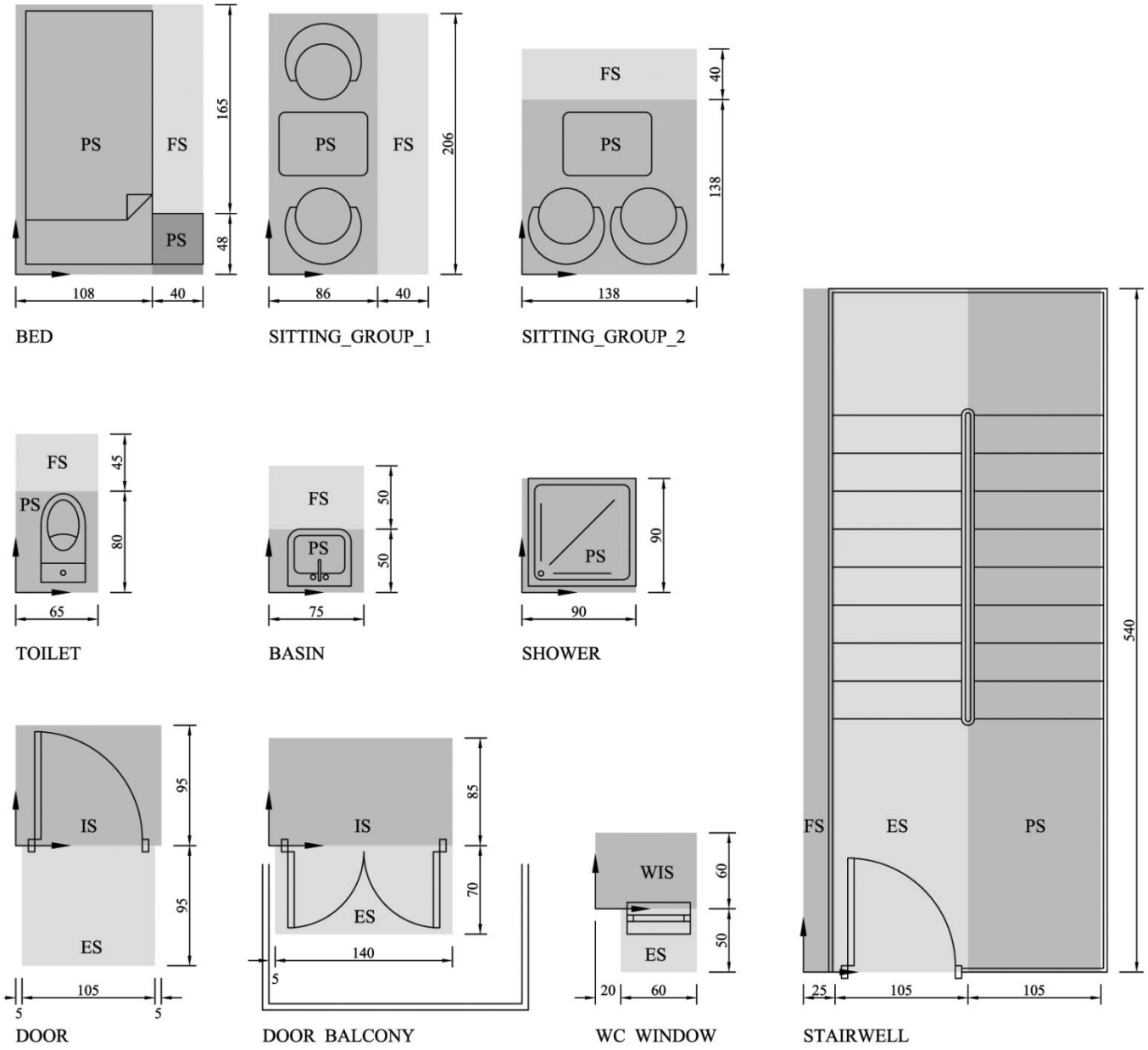


Fig. 5. Standard (nonevolvable) "units" within the "world unit" of the hotel application. Each standard "unit" includes "primitives" represented as shaded rectangles. Dimensions of "primitives" represent centimeters. Spatial attributes PS, FS, IS, ES, and WIS, each one assigned to a "primitive," denote "physical space," "functional space," "internal space," "external space," and "window internal space," respectively. The OCS of each "unit" appears as a system of two vectors, perpendicular to each other, where the horizontal vector represents the X axis. The name appearing under each "unit" constitutes the value of a property that represents the "unit's" identity. A specific property of each "unit" refers to a drawing that represents the "unit" as a physical object. This drawing is to be drawn by the CAD interface in the solution presentation phase.

that, respectively, represent the physical and the functional space of a wardrobe having fixed width and variable length, as shown in Fig. 7.

The positioning specifications for all the "instances" included in *Type_Full* are presented in Table V. These specifications are encoded within the "instances" "positioning lists" (see Section IV-C2b2). The APA "queries" (see Section IV-C2b1) of the "primitives" of the instanced "units" are presented in Table VI. The *PS* "primitive" of the "unit" *Wardrobe* satisfies the positioning specifications presented in Table V for the *Bed*, *Sitting_Group_1*, and *Sitting_Group_2* "instances," while the APA "queries" of both "primitives" of the "unit" *Wardrobe* are the same with the APA "queries" of the homonymous "primitives" of the aforementioned "instances" (see Table VI).

Examples of evolved *Type_Full* "units" are presented in Fig. 8 as architectural drawings.

b3) The "unit" *Hotel*

The "unit" *Hotel* has terminal objects eight *Type_Full* "instances," various "primitives" having the spatial attribute *COR* which stands for "corridor," and a *Stairwell* "instance." The *Hotel* "unit" creation formula includes a reference to a *Type_Full* "instance" creation formula, seven references to a "unit" creation formula labeled as *Place*, a reference to a *Stairwell*

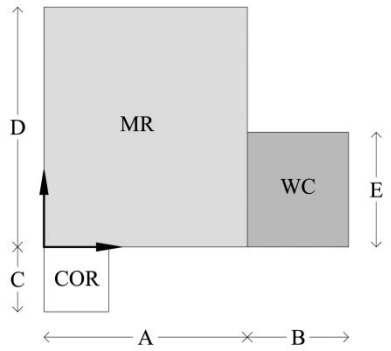


Fig. 6. "Primitives'" configuration within "unit" *Type* (shaded) in relation to a corridor's region (white), where the "unit" *Type* is to be attached. *Type*'s OCS appears as a system of two vectors perpendicular to each other, where the horizontal vector represents the X axis. Spatial attributes MR, WC, and COR denote "Main Room," "WC," and "Corridor," respectively. Dimensions A, B, D, and E vary satisfying specific geometric constraints, while dimension C is fixed.

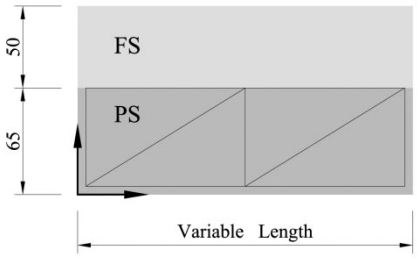


Fig. 7. "Primitives" of "unit" *Wardrobe* are represented as shaded rectangles. Dimensions of "primitives" represent centimeters. Spatial attributes PS and FS, each one assigned to a "primitive," denote "physical space," and "functional space," respectively. The OCS of the PS "primitive," which is also the *Wardrobe*'s OCS, appears as a system of two vectors, perpendicular to each other, where the horizontal vector represents the X axis. A specific property of the PS "primitive" refers to a drawing that represents the wardrobe as a physical object. This drawing is to be drawn by the CAD interface in the solution presentation phase.

"instance" creation formula, and a reference to a secondary formula that calculates area. *Place* selects randomly and implements one of the four positioning methods presented in Fig. 9.

Each positioning method is represented by a different "unit" creation formula referenced by *Place*. Each positioning method "unit" includes one or two *COR* "primitives" and a *Type_Full* "instance," all positioned in relation to the *Type_Full* "instance" created last. Overlapping conditions are determined by the APA properties of both the *COR* "primitives" and the terminal "primitives" of the "unit" *Type* (i.e., the "primitives" having the spatial attributes *MR* and *WC*, respectively). Due to these conditions, overlapping is only allowed between *COR* "primitives." The secondary formula referenced by the *Hotel* "unit" creation formula calculates the inversed value of the hotel's floorplan area, which is considered as the area of the region specified by the "query" (*COR* or *MR* or *WC*). The inversed value constitutes the magnitude to be maximized.

B. Solution Procedure and Results

The formal statement and the solution procedure of the problem presented in the previous paragraphs were based on the prototype version of GENETICAs programming environment, introduced in Section I-C, running in a 550 MHz CPU. After the formulation of the design problem (steps 1 and 2 in Section IV-C3) an initial configuration of the computational parameters was given within GENETICAs environment, while adjustments of these parameters were made during the solution procedure (step 3 in Section III-C3). The population size (see Section III-D3a was set to 100, while the number of tests per computational cycle (see Section III-D3b was set to 40 for the first cycle and 20 for the next cycles. Only subtree mutations were performed in the first cycle, while crossovers were introduced in the second cycle and Gaussian mutations in the 13th cycle (see Section III-D3).

The first confirmation of the root formula, leading to a solution, occurred in the third cycle after a total of 180 executions (including the population initialization phase) of the GENETICA program had been performed in 7 h of processing time. The solution produced has a 1 954 576 cm^2 built area. The final configuration of apartments, which is the best possible for the particular problem, occurred in the 10th cycle after a total of 320 executions had been performed in 12 h and 40 min of processing time. The solution produced has a 1 759 139 cm^2 built area. Minor improvements occurred in the next cycles where the search was gradually localized through the regulation of the computational parameters. The final solution, presented in Fig. 10, occurred in the 47th cycle after a total of 880 executions had been performed in 35 h and 40 min of processing time. The final solution has a 1 748 619 cm^2 built area.

Solutions were converted to architectural drawings by the CAD interface (see Section IV-C3). The conversion rules are presented here:

- draw thick walls at the boundaries between the *(MR or WC or COR)* region and the *(not (MR or WC or COR or ES))* region;
- draw thin walls at the boundaries between the *(MR and (not ES))* region and the *WC* region, the boundaries between the *(COR and (not ES))* region and the *(MR or WC)* region, and the boundaries between regions of different *Type_Full* "instances";
- draw the drawings associated to the objects presented in Figs. 5 and 7, with respect to the OCS of each object's "instance" within the solution.

GENETICAs environment allows for the visualization of the computational process by presenting the fitness values, the percentage of the new "species" (see Section III-D1) introduced in the population, and the "species'" distribution within the population, as functions of time which is represented by the number of computational cycles. The visualization of the computational process during the solution of the hotel design problem is presented in Fig. 11.

C. Comparison With Other Evolutionary Design Methods

Note the following features of the hotel application presented in the previous paragraphs.

1) The design problem is a real scale one: both the complexity of the problem and the level of the solution details are encountered by human architects.
2) The design problem has both confirmation and optimization goals, i.e., satisfying complex spatial relations and minimizing the area of the built space.

TABLE V

POSITIONING SPECIFICATIONS FOR THE "INSTANCES" OF THE STANDARD "UNITS" PRESENTED IN FIG. 5. "INSTANCES" ARE POSITIONED HAVING THE X AXIS OF THEIR OCS AT THE BOUNDARY OF TWO REGIONS, REFERRED TO AS "POSITIONING REGION" AND "ADJACENT REGION" RESPECTIVELY, WHILE THE Y AXIS SHOWS AT THE SIDE OF THE "POSITIONING REGION." THE ROWS OF THE TABLE REPRESENT "INSTANCES." THE COLUMNS NAMED "POSITIONING REGION" AND "ADJACENT REGION" PRESENT "QUERIES" SPECIFYING THE RESPECTIVE REGIONS FOR EACH "INSTANCE," WHILE THE REMAINING COLUMNS PRESENT ROTATION AND MIRRORING CONSTRAINTS FOR THE POSITIONING OF EACH "INSTANCE." EMPTY CELLS IN THE LATTER COLUMNS INDICATE UNCONSTRAINED POSITIONING

Instance	Positioning Region	Adjacent Region	Rotation Constraints	Mirroring Constraints
Door (Entrance)	MR	(not MR)	0^0	Not Mirrored
Door (WC)	WC	MR		
Door_Balcony	MR	(not (MR or WC))	180^0	
WC_Window	WC			
Bed	(MR and (not (IS or ES or PS or FS)))	(not MR)		
Sitting_Group_1				
Sitting_Group_2				
Basin	(WC and (not (IS or WIS or PS or FS)))	(not WC)		
Shower	(WC and (not (IS or PS or FS)))			
Toilet				
Stairwell	(not (COR or MR or WC or ES))	COR		

TABLE VI

APA VALUES OF THE "PRIMITIVES" OF THE STANDARD "UNITS" PRESENTED IN FIG. 5. THE APA VALUE OF A "PRIMITIVE" (i.e., THE VALUE OF THE APA PROPERTY WITHIN THE "PRIMITIVE'S" "PROPERTY LIST") IS A "QUERY" THAT SPECIFIES THE REGION WHERE THE "PRIMITIVE" SHOULD BE INCLUDED, GIVEN A "SPATIAL MAP." THE ROWS OF THE TABLE REPRESENT THE STANDARD "UNITS" PRESENTED IN FIG. 5. THE COLUMNS REPRESENT "PRIMITIVES," EACH ONE SPECIFIED BY A SPATIAL ATTRIBUTE. THE CELLS PRESENT THE APA VALUES OF EACH "UNIT'S" "PRIMITIVES"

Instanced Unit	IS	WIS	ES	PS	FS
Door	(MR or WC)		(COR or MR)		
Door_Balcony	MR		(not (COR or MR or WC))		
WC_Window		WC			
Bed				(MR and (not (IS or ES or PS or FS)))	(MR and (not PS))
Sitting_Group_1					
Sitting_Group_2					
Basin				(WC and (not (IS or WIS or PS or FS)))	(WC and (not PS))
Shower				(WC and (not (IS or PS or FS)))	
Toilet					
Stairwell			(not (COR or MR or WC or ES))		

3) Mutually depended qualitative attributes (e.g., selection and spatial configuration of the proper objects) and quantitative attributes (e.g., dimensions of objects) should be combined in a solution in order to achieve the aforementioned goals.

4) Design knowledge, which is represented in terms of formal logic, is encapsulated in the definitions of the design objects and determines their behavior. Both object-to-object (e.g., bed-to-wardrobe) and object-to-environment (e.g., bed-to-apartment or apartment-to-hotel)

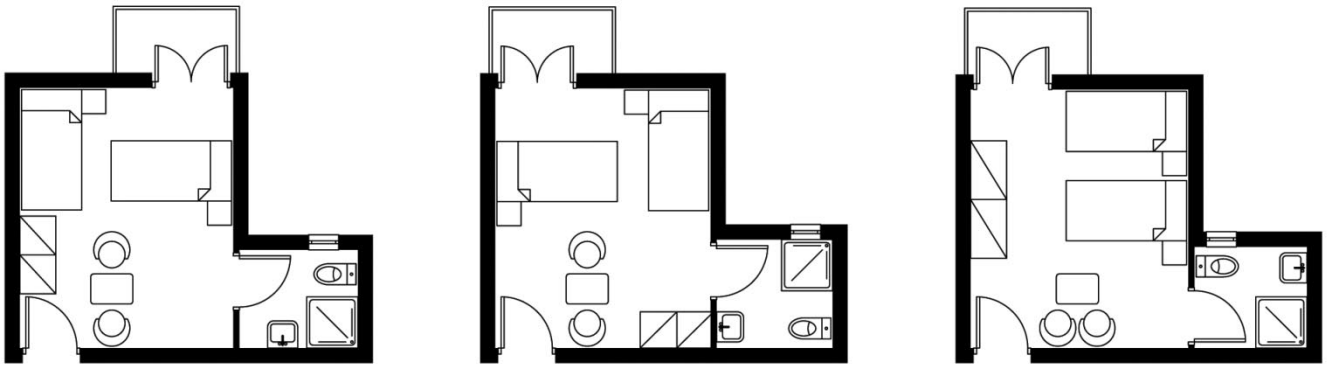


Fig. 8. Graphic representation of evolved Type_Full "units," appearing as architectural drawings.

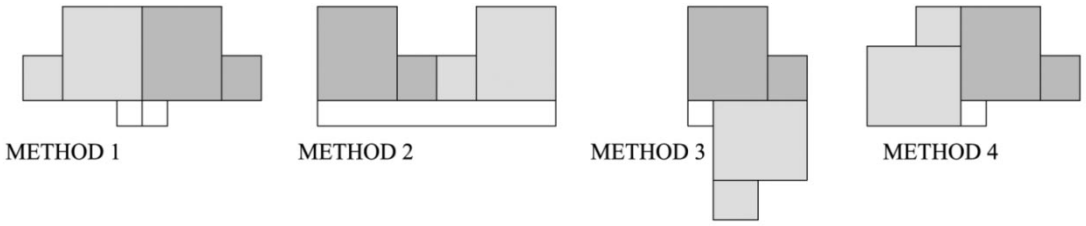


Fig. 9. Four methods for positioning a new apartment in relation to an existing one, where both apartments constitute "instances" of the "unit" *Type_Full*. The existing apartment appears dark gray whereas the new apartment appears light gray. Each positioning method is implemented by a "unit" that includes the *Type_Full* "instance" appearing light gray and also one or two COR "primitives" appearing white.

interactions emerge due to the objects' behaviors. These interactions activate self-assembly procedures which construct the solution in an unpredictable way, i.e., in a way not indicated by the knowledge encapsulated in the objects' definitions.

5) Three design scales are involved in the problem: a) specify the large scale configuration (apartments-corridor-stairwell); b) specify the dimensions and the openings of an apartment; and c) specify the objects' configuration within an apartment. Scale-to-scale dependencies occur, e.g., potential furniture configurations depend on the dimensions of an apartment and vise versa, the area of the built space depends both on the apartment's dimensions and on the large scale configuration, etc.
6) The hotel application was not developed as a hotel design method from scratch: the development was based on both expressive and computational resources derived from G-CAD which provides a general architectural design method. G-CAD itself was not developed from scratch: the development was based both on expressive and computational resources derived from GENETICA which—in common with its programming environment—provides a general problem formulation and solving method.

To the best knowledge of the author, the solution of architectural design problems combining features 1, 2, and 3 has not been attempted so far by other automatic design methods (evolutionary or not).

GP design methods have been tested in real scale problems (feature 1) in different domains, such as electrical circuits [5, pp. 383–906], [64]–[66], parameterized controllers [67], or metabolic pathways [68]. Due to their creative character, GP methods can cope with problems the solution of which requires a combination of both qualitative and quantitative attributes (feature 3), while qualitative attributes can be interpreted as confirmation goals (feature 2). However, a general representation of such goals cannot be included in the problem statement, due to the numerical nature of the fitness function. Unlike GP methods, the method presented here allows logical structuring of both confirmation and optimization goals in the problem statement. The method is "creative," since the combination of the solution attributes that satisfy these specifications is achieved by evolution, but the (already known) logical context that specifies the domain of the evolving entities constitutes part of the problem statement; as a consequence, the context does not have to be discovered by evolution and the computational cost of this discovery does not have to be paid. This promotes effectiveness: the results presented here, although not directly comparable with the results of different GP real scale design methods due to the domain difference, indicate that real scale design problems could be solved with a computational effort very small both in fitness evaluations and in processing time.

Knowledge encapsulation (feature 4) has been attempted by the design of the proper "genes" in various evolutionary design methods (e.g., [55, p. 345], [56, pp. 402–403]). A more effective encapsulation is encountered in the functions that constitute the function set of GP design applications (e.g., [64, p. 1085], [65, p. 1478], [66, p. 480], [67, p. 499]): unlike genes, functions perform arithmetic or logical operations while they cause elementary changes of the structure of evolving

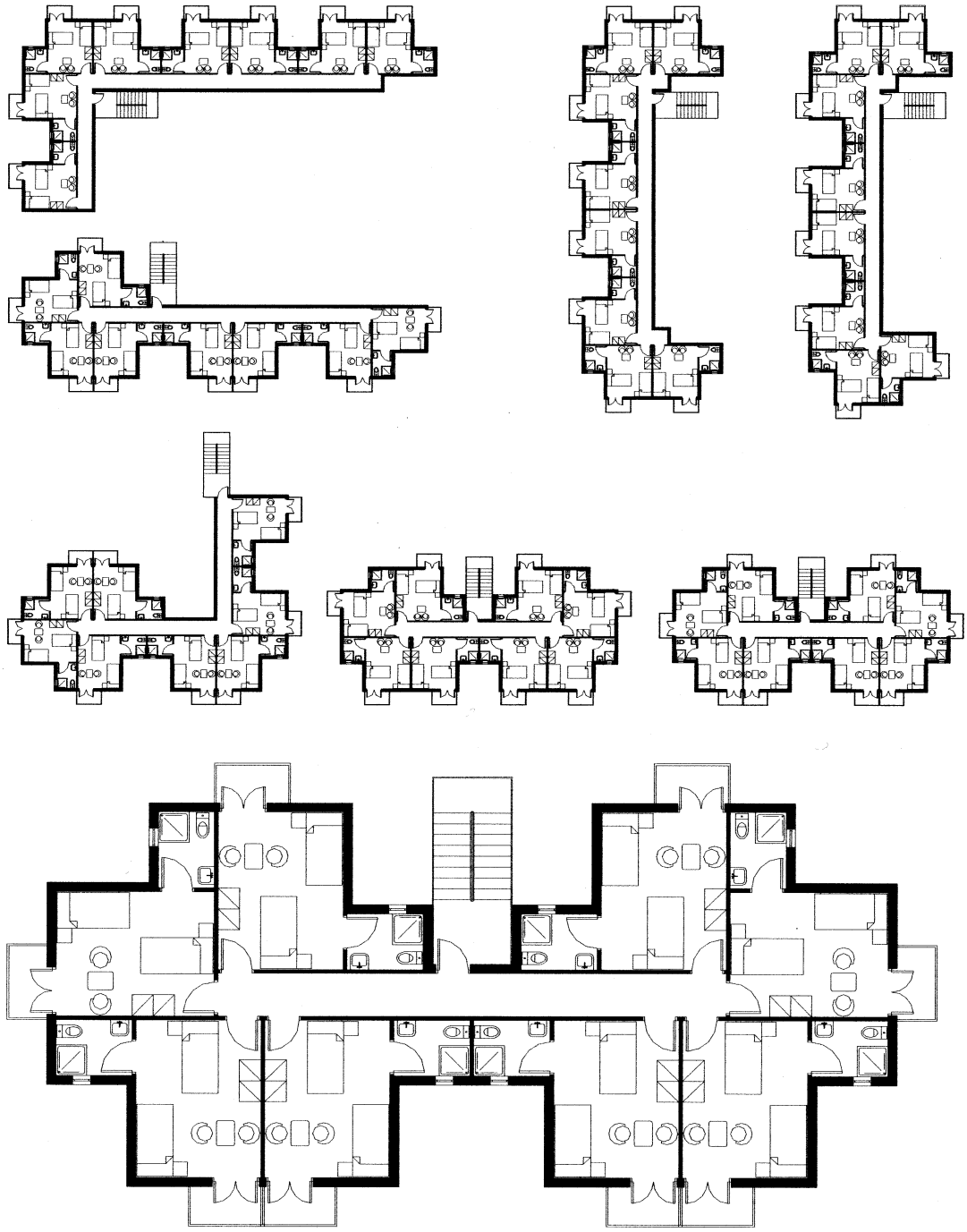


Fig. 10. Graphic representation of evolved *Hotel* "units" appearing as architectural drawings. The "units" are presented in increasing fitness order from the left to the right and from the top to the bottom. The best fitness "unit," occurred in the 47th computational cycle, appears magnified.

objects. Encapsulation in G-CAD objects enhances further the representation of problem specific knowledge by including the representation of goals, logical behaviors, construction rules, and complex data structures as object properties.

Due to their high computational complexity, multiscale design problems (feature 5) have been partitioned into different problems in various evolutionary methods (e.g., [55] and [56]). Reconstruction of the multiscale problem has been attempted by strictly bottom-up explicit embryogenies (e.g., [55]). This severely constrains the interactions between objects represented in different scales and consequently the method's ability to cope with scale-to-scale dependencies. The method presented here can cope with multiscale design problems without partitioning, due to the logical structuring of the problem which leads to search space reduction.

The hotel application, presented here, is a highly specialized one. Specialized applications are usually effective but may not be general. Their development cost is usually high. However, the hotel application is an instance of the class of the applications that can be developed in G-CAD. G-CAD provides both generality, within the domain of architectural design, and tools for the development of problem specific applications.

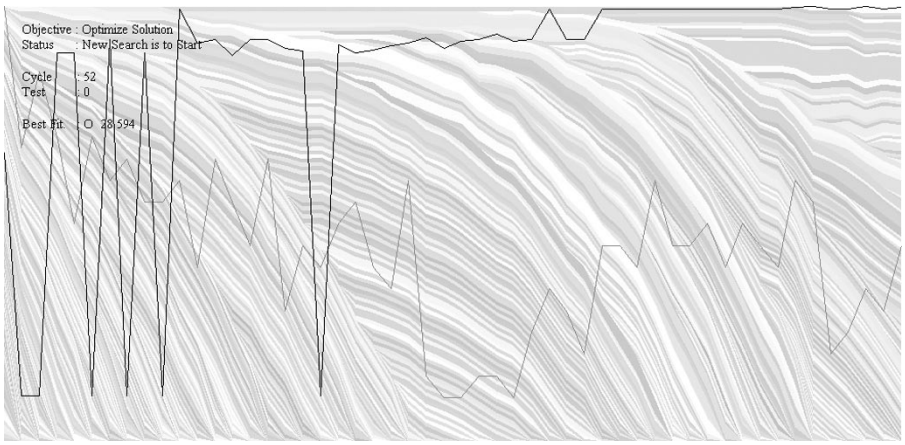


Fig. 11. Visualization of the computational process during the solution of the hotel design problem. Consider the bottom edge and the left edge of the textured area as the horizontal axis and the vertical axis of a diagram, respectively. The horizontal axis represents time in computational cycles. A range of 52 cycles is presented. The vertical axis represents three properties of the search state: 1) "Species" evolution, where "species" (see Section III-D1) are represented by grayscale zones ordered by fitness, with the best fitness "species" at the top of the diagram. The thickness of the zones represent relative "species sizes" (see Section III-D1). Critical innovations (i.e., new best fitness "species") emerge at the top of the diagram, while extinction takes place at the bottom of the diagram; 2) the best fitness encountered within each computational cycle is represented by a black line. The first best fitness value, appearing at the left, represents a high "distance from confirmation" of the root formula. Low best fitness values (lower left) represent a low "distance from confirmation." The remaining best fitness values correspond in cases where the root formula has been confirmed, while they represent the magnitude to be maximized. The latter magnitude has been increased by a factor of $5 \cdot 10^7$ in order to be clearly presented; and 3) the ratio of the "tests" that introduce new"species " in the population to all the "tests" performed during a computational cycle is represented by a gray line.

G-CAD is a domain specific language having itself a high cost of development. However, G-CAD is an instance of the class of the languages that can be developed in GENETICA. GENETICA provides both generality, within the domain of evolutionary problem solving, and tools for the development of domain specific languages. A similar step by step route from generality to specificity (feature 6) has not been reported in other evolutionary design methods, to the best knowledge of the author.

VI. Conclusion

The conclusion of this paper is that it is possible to develop a problem-solving method that provides evolutionary computational support to general problem formulations within some important areas of formal expression, including both first-order logic and high-order modes of expression. This makes possible to apply a problem-independent evolutionary solution procedure to a large class of problems having an accurate and potentially sophisticated formal statement. The problem-solving method can be implemented by a computer language integrated in a programming environment having a built-in evolutionary computational system. Data generation scenarios in the language can be viewed as interpretations of abstract structures evolved according to the computational system. The field of application of the proposed method includes general confirmation problems and also problems combining both confirmation and optimization goals. Within this approach the following issues could be investigated:

- integration of AI expressive power with the computational efficiency of evolutionary methods;
- development of domain specific languages supported by evolutionary heuristics;
- development of GP versions of the latter languages;
- development of creative evolutionary design applications having complex confirmation and optimization goals.

Acknowledgment

The author would like to thank V. Androulakis, P. Nikolaidis, K. Patapis, and J. Stavridis for very useful discussions and comments. P. Iatroudakis and A. Koumba kindly offered important comments and advice. V. Sakellariou offered valuable commends and information concerning the theory of mathematical logic, while he encouraged me to proceed with high-order modes of expression. The author wishes particularly to thank Prof. P. Ross (University of Edinburgh) and Assistant Prof. Y. Veneris (NTUA). Both provided critical comments and advice which helped me to give this paper its present form. Of course, none of the above could be held responsible for the views expressed in the present paper.

References

[1] L. Dekker and J. Kingdon, "Development needs for diverse genetic algorithm design," in *Genetic Algorithms in Optimization Simulation and Modeling*, J. Stender, E. Hillebrand, and J. Kingdon, Eds. Amsterdam, The Netherlands: IOS Press, 1994, pp. 9–26.

[2] S. Goonatilake and K. Feldman, "Genetic rule induction for financial decision making," in *Genetic Algorithms in Optimization Simulation and Modeling*, J. Stender, E. Hillebrand, and J. Kingdon, Eds., 1994, pp. 185–201.

[3] N. J. Radcliffe and P. D. Surry, "The reproductive plan language RPL2: Motivation, architecture and applications," in *Genetic Algorithms in Optimization Simulation and Modeling*, J. Stender, E. Hillebrand, and J. Kingdon, Eds., 1994, pp. 65–94.

[4] I. T. Christou and A. Zakarian, "Domain knowledge and representation in genetic algorithms for real world scheduling problems," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds. San Francisco, CA, 2000, pp. 690–696.

[5] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane, *Genetic Programming III Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann, 1999.

[6] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995.

[7] C. Harris, "Strongly typed genetic programming to promote hierarchy through explicit syntactic constraints," in *Proc. Late Breaking Papers at the Genetic Programming 1997 Conf.*, J. R. Koza, Ed.. Stanford, CA, 1997, pp. 72–80.

[8] N. F. McPhee, N. J. Hopper, and M. L. Reierson, "Impact of types on essentially typeless problems in GP," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds. San Francisco, CA, 1998, pp. 232–240.

[9] T. Yu and C. Clack, "PolyGP: A polymorphic genetic programming system in Haskell," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza , W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 416–421.

[10] L. Martin, F. Moal, and C. Vrain, "Declarative expression of biases in genetic programming," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds. San Francisco, CA, 1999, pp. 401–407.

[11] B. J. Ross, "Logic-based genetic programming with definite clause translation grammars," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf , J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., 1999, p. 1236.

[12] M. Keizer, V. Babovic, C. Ryan, M. O'Neill, and M. Catolico, "Adaptive logic programming," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds. San Francisco, CA, 2001, pp. 42–49.

[13] W. B. Langdon, *Genetic Programming and Data Structures: Genetic Programming + Data Structures=Automatic Programming*. Norwell, MA: Kluwer, 1998.

[14] I. Muslea, "A general-purpose ai planning system based on the genetic programming paradigm," in *Proc. Late Breaking Papers at the Genetic Programming 1997 Conf.*, J. R. Koza, Ed., pp. 157–164.

[15] T. Yu and C. Clack, "Recursion, lambda abstractions and genetic programming," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 422–430.

[16] A. Giordana and F. Neri, "Genetic algorithms in machine learning," *AI Commun.*, vol. 9, pp. 21–26, 1996.

[17] A. Giordana, F. Neri, L. Saitta, and M. Botta, "Integrating multiple learning strategies in first order logics," *Mach. Learn.*, vol. 27, no. 3, pp. 209–240, 1997.

[18] A. Tamaddoni-Nezhad and S. Muggleton, "Using genetic algorithms for learning clauses in first-order logic," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., pp. 639–646.

[19] P. G. K. Reiser and P. J. Riddle, "Evolving logic programs to classify chess-endgame positions," in *Proc. Simulated Evolution and Learning: Selected Papers/2nd Asia Pacific Conf. Simulated Evolution and Learning SEAL '98*, B. McKay, X. Yao, C. S. Newton, J.-H. Kim, and T. Furuhashi, Eds. New York, 1999, pp. 138–145.

[20] C. Chakraborti and K. K. N. Sastry, "The genetic algorithms approach for proving logical arguments in natural language," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 463–470.

[21] K. Musumbu, K. Barbar, and M. Nassif, "Evolution strategies to improve abstract interpretation algorithms for logic programming," in *Proc. Late Breaking Papers at the Genetic Programming 1997 Conf.*, J. R. Koza, Ed., 1997, pp. 165–173.

[22] M. Fuchs. (1999) Automated Reasoning Project. Research School of Inform. Sci. Eng., Australian Nat. Univ., Tech. Rep. TR-ARP-04-99. [Online]. Available: http://arp.anu.edu.au:80/ftp/techreports/1999/TR-ARP-04-99.ps.gz

[23] Y. J. Hu, "PolyGP: A genetic programming approach to constructive induction," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 146–151.

[24] M. Fuchs, D. Fuchs, and M. Fuchs, "Generating lemmas for tableau-based proof search using genetic programming," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1027–1032.

[25] J. Neves, M. Rocha, H. Rodrigues, M. Biscaia, and J. Alves, "Adaptive strategies and the design of evolutionary applications," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 473–479.

[26] P. Bentley, "'Evolutionary, my Dear Watson' investigating committee-based evolution of fuzzy rules for the detection of suspicious insurance claims," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., 2000, pp. 702–709.

[27] L. Sammartino, M. Simonov, M. Soroldoni, and A. G. B. Tettamanzi, "GAMUT: A system for customer modeling based on evolutionary algorithms," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., p. 758.

[28] F. Gruau, "Genetic synthesis of modular neural networks," in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed.. San Francisco, CA, 1993, pp. 318–325.

[29] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, pp. 1423–1447, Sept. 1999.

[30] Y. Manin, *A Course in Mathematical Logic*. New York: Springer-Verlag, 1977.

[31] W. B. Langdon, "Size fair and homologous tree genetic programming crossover," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1092–1097.

[32] F. D. Francone, M. Conrads, W. Banzhaf, and P. Nordin, "Homologous crossover in genetic programming," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1021–1026.

[33] R. Poli and N. F. McPhee, "Exact Schema theory for GP and variable length GAs with homologous crossover," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., pp. 104–111.

[34] R. Poli and W. B. Langdon, "On the search properties of different crossover operators in genetic programming," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 293–301.

[35] R. Poli, J. Page, and W. B. Langdon, "Smooth uniform crossover, sub-machine code gp and demes: A recipe for solving high-order Boolean parity problems," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1162–1169.

[36] C. Igel and K. Chellapilla, "Investigating the influence of depth and degree of genotypic change on fitness in genetic programming," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1061–1068.

[37] M. Evett and T. Fernandez, "Numeric mutation improves the discovery of numeric constraints in genetic programming," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 66–71.

[38] K. Chellapilla, H. Birru, and S. Sathyanarayan, "Effectiveness of local search operators in evolutionary programming," in *Proc. 3rd Annu. Genetic Programming Conf.*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, Eds., 1998, pp. 753–761.

[39] J. M. Yang and C. Y. Kao, "An evolutionary algorithm for continuous global optimization," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 930–937.

[40] P. Walsh, "A functional style and fitness evaluation scheme for inducting high level programs," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1211–1216.

[41] F. Divina and E. Marchiori, "Knowledge based evolutionary programming for inductive learning in first-order logic," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., p. 173.

[42] H. Kautz, M. Kearns, and B. Selman, "Horn approximations of empirical data," *Artif. Intell.*, vol. 74, pp. 129–145, 1995.

[43] E. Henry and J. R. Kyburg, "Combinatorial semantics: semantics for frequent validity," *Comput. Intell.*, vol. 13, no. 2, pp. 215–257, 1997.

[44] R. E. Keller and W. Banzhaf, "Genetic programming using genotype-phenotype mapping from linear genome into linear genotypes," in *Proc. 1st Annu. Conf. Genetic Programming 1996*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., 1996, pp. 116–122.

[45] ——, "The Evolution of genetic code in genetic programming," in *Proc. GECCO'99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1077–1082.

[46] ——, "Evolution of genetic code on a hard problem," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., pp. 50–56.

[47] M. O'Neill and C. Ryan, "Under the hood of grammatical evolution," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1143–1148.

[48] ——, "Grammar based function definition in grammatical evolution," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., pp. 485–490.

[49] C. M. Eastman, "Logical methods of building design: A synthesis and review," *DMG-DRS J. Design Res. Methods*, vol. 6, no. 3, pp. 79–87, 1972.

[50] W. J. Mitchell, "The theoretical foundation of computer-aided architectural design," *Environ. Planning B: Planning Design*, vol. 2, pp. 127–150, 1975.

[51] P. S. G. Swinson, "PROLOG: A relude to a new generation of CAAD," *Environ. Planning B: Planning Design*, vol. 15, pp. 335–343, 1983.

[52] R. D. Coyne and J. S. Gero, "Logic programming as a means of representing semantics in design languages," *Environ. Planning B: Planning Design*, vol. 12, pp. 351–369, 1985.

[53] ——, "Design knowledge and context," *Environ. Planning B: Planning Design*, vol. 12, pp. 419–442, 1985.

[54] L. Virirakis, "The continuous search space design method (CSSDM)," *Environ. Planning B: Planning Design*, vol. 20, pp. 617–643, 1993.

[55] M. A. Rosenman and J. S. Gero, "Evolving designs by generating useful complex gene structures," in *Evolutionary Design by Computers*, P. Bentley, Ed. London, U.K.: Morgan Kaufmann, 1999, pp. 345–364.

[56] A. G. De Silva Garza and M. L. Maher, "A process model for evolutionary design case adaptation," in *Artificial Intelligence in Design '00*, J. S. Gero, Ed. Dordrecht, The Netherlands: Kluwer, 2000, pp. 393–412.

[57] R. Jagielski and J. S. Gero, "A genetic programming approach to the space layout planning problem," in *CAAD Futures 1997*, R. Junge, Ed. Dordrecht, The Netherlands: Kluwer, 1997, pp. 875–884.

[58] J. S. Gero and V. Kazakov, "Adapting evolutionary computing for exploration in creative designing," in *Computational Models of Creative Designing*, J. S. Gero and M. L. Maher, Eds. Sydney, Australia: Key Centre of Design Computing and Cognition, Univ. Sydney, 1999, pp. 175–186.

[59] J. S. Gero, "Novel models in evolutionary designing," in *Proc. Simulated Evolution and Learning: Selected Papers/2nd Asia Pacific Conf. Simulated Evolution and Learning SEAL '98*, B. McKay, X. Yao, C. S. Newton, J.-H. Kim, and T. Furuhashi, Eds., pp. 381–388.

[60] J. S. Gero and X.-G. Shi, "Design development based on an analogy with developmental biology," in *Proc. CAADRIA'99*, J. Gu and Z. Wei, Eds. Shanghai, China, 1999, http://www.arch.su.edu.au/~john/publications/PDF2/99GeroShiCAADRIA.pdf, pp. 253–264.

[61] H. Lipson and N. P. Suh, "Toward a universal design-component architecture for design automation," in *Proc. ICAD2000 1st Int. Conf. Axiomatic Design*, Cambridge, MA, June 21–23, 2000, [Online]. Available: http://www.cs.brandeis.edu/~lipson/.

[62] P. Bentley and S. Kumar, "Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem," in *GECCO-99 Proc. Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 35–43.

[63] C. Kyriazopoulos, "The CADRAM program: An application for CAAD based on the primitives' allocation method," Diploma thesis, Dept. Architectural Eng., Nat. Tech. Univ. Athens (available from the Library of Dept. Architectural Eng.), 1995.

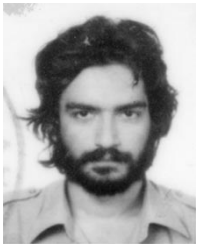
[64] J. R. Koza, M. A. Keane, F. H. Bennett, J. Yu, W. Mydlowec, and O. Stiffelman, "Searching for the Impossible using genetic programming," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1083–1091.

[65] F. H. Bennett, J. R. Koza, M. A. Keane, J. Yu, W. Mydlowec, and O. Stiffelman, "Evolution by means of genetic programming of analog circuits that perform digital functions," in *Proc. GECCO-99 Genetic and Evolutionary Computation Conf.*, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., pp. 1477–1483.

[66] J. R. Koza, M. A. Keane, J. Yu, and W. Mydlowec, "Automatic synthesis of electrical circuits containing a free variable using genetic programming," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., pp. 477–484.

[67] M. A. Keane, J. Yu, and J. R. Koza, "Automatic synthesis of both the topology and tuning of a common parameterized controller for two families of plants using genetic programming," in *Proc. GECCO-2000 Genetic and Evolutionary Computation Conf.*, D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., pp. 496–504.

[68] J. R. Koza, W. Mydlowec, and G. Lanza, "Automatic synthesis of both the topology and sizing of metabolic pathways using genetic programming," in *Proc. GECCO-2001 Genetic and Evolutionary Computation Conf.*, L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., pp. 57–65.



Lefteris Virirakis received the Architect-Engineer degree and the Ph.D. degree in engineering from the National Technical University of Athens (NTUA), Greece, in 1986 and 1991, respectively. The doctorate thesis was on a computer-aided automatic architectural design method within the framework of evolutionary computation.

He developed CAD applications for construction companies and software houses and three-dimensional rendering and animation presentations of architectural projects for construction companies and design studios. He lectured on CAD applications at the National Technical University of Athens (continuous education) from 1994 to 1995 and at commerical companies from 1994 to 1997. He lectured on applications of informatics to design methods in NTUA postgraduate studies programs from 1998 to 2003. He authored *The Continuous Search Space Design Method, Environment and Planning B: Planning and Design* (1993) and *CSSDM: A Computer Aided Architectural Design Method* (Technika Chronika, 1995).