International Journal of Computer & Information Science, Vol. 5, No. 1, March 2004

Deductive and Inductive Methods for Program Synthesis

Jaan Penjam and Jelena Sanko[0]
Institute of Cybernetics, Estonia

Abstract

The paper discusses simple functional constraint networks and a value propagation method for program construction. Structural synthesis of programs is described as an example of deductive approach to program construction. An inductive method for program synthesis utilizing stochastic optimization algorithms is introduced to complement value propagation techniques.

Keywords: Structural synthesis of programs, functional constraint networks, differential evolution.

1. Introduction

Formal specifications that define required system properties play a central role in software engineering. They can be used for reasoning about complex programs on a suitable abstract level. Formal models are inescapable when programs are automatically derived or verified. The most widely used techniques for automatic program construction are based on stepwise refinement of a specification until actual program code is derived. In this paper, we will consider different approaches for program construction where programs are obtained by means of special mathematically based manipulations, without construction of intermediate level specifications. In particular, we will briefly introduce a method based on proof search, known as structural synthesis of programs (SSP), and on the other hand, a technique based on the optimization of a specific fitness function related to the task to be solved.

SSP has been implemented in a couple of commercial and some experimental systems that are either of scientific or pedagogical value. SSP's most advanced implementation in the system NUT is described in several papers, see [12, 3]. The SSP method is a *deductive approach* for program synthesis carried out via special logical inference rules from a general *relational specification* of a problem. This specification is often called the *computational model* of the problem and contains only information about the applicability of functions for computing values of variables.

Computational models are treated as functional constraint networks in this paper. A functional constraint network provides a simple data-flow between the problem variables, see Fig. 1. For a detailed explanation of the used notations see Section 2.

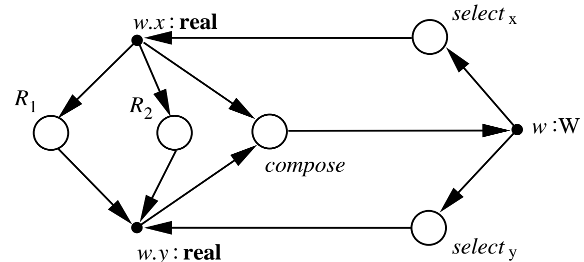


Figure 1. Computational model

A user-friendly object oriented specifications can be built on the top of constraint networks. For example, the network in Fig. 1 corresponds to the class declaration

```
Class W:(var x,y:real;
         rel   R: 2*x - y = 0);
```

that defines the structure of the objects of class W and an equation binding the components of these objects. The constraints in Fig. 1, that could be extracted from this class specification, contain functions (subroutines) implementing structural relationship and functions representing computational relations (solving equations, reusing of existing program modules etc). Here $select_x$ and $select_y$ give components of a compound object and $compose$ unites variables in a composite object. Functions $y = R_1(x) = 2x$ and $x = R_2(y) = x/2$ provide roots of the equation R.

Several specification languages have been implemented in NUT. Each of these languages could be used at an appropriate level of system design. A snapshot in Fig. 2 demonstrates a class declaration (window *Surveillance*), visual specification of the problem (window *Surveillance: scheme*), and a program synthesized by means of SSP (window *Algorithm*).

We will design an *inductive approach* for automatic program construction based on the same relational specification language (computational model) that is augmented by experimental data that describe desirable input-output behavior of the program to be composed. A common specification would bridge deductive and inductive methods of program construction, and combining these two types of techniques might provide more general and effective procedures for automation

[0]Institute of Cybernetics at TUT,
Akadeemia tee 21,
12618 Tallinn, Estonia
{jaan | jelena}@cs.ioc.ee

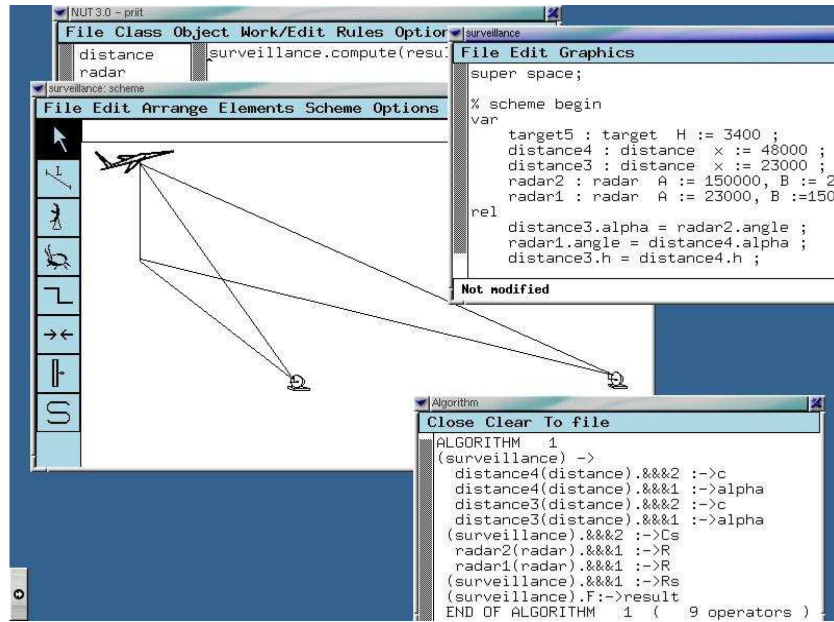


Figure 2. User interface of NUT system

of software development. This would simulate human reasoning, where deductive inference steps are interleaved with drawing conclusions from samples of experimental data.

The next section presents a simple case of computational models and SSP in terms of constraint satisfaction problem as in [13]. In Section 3, program construction is described as an optimization task. A way of encoding data flow information and a problem specification by the real-value function is introduced and analyzed from the perspectives of finding its extreme points. In the last sections, possibilities for generalization of the introduced method are reviewed and examples of synthesis of an "optimal program" using the differential evolution method are given [10].

2. Functional constraint networks and program synthesis

2.1 Definitions

Suppose that our universe (model) contains the variables needed for solving a problem and functional relations between these variables. To solve the problem we shall try to build a program that contains functions from this given model and operates on the variables from this model.

Throughout the paper, an underlined symbol $\underline{X}$ stands for a finite list of elements of the set X. The number of elements in the list $\underline{X}$ is denoted by $|\underline{X}|$. The set of all finite lists over X is denoted by $X^\star$. Concatenation of lists α and β is denoted by α, β. A list containing just one element will be identified with its element.

For any variable x, a non-empty set $D(x)$, called its *domain*, is given. Function D can be defined also for lists of variables as the Cartesian product of domains of its elements: $D(x_1, \ldots, x_n) = D(x_1) \times \ldots \times D(x_n)$.

We consider *constraints* as *relations of finite arity* on the universal domain specifying which combinations of values of program variables are acceptable. For example, in program synthesis, the input-output relations of the expected program act as constraints. A relation can be defined either extensionally by plainly enumerating the tuples it contains, or intensionally by some effective characterization of its extension. Formally, constraints can be written as formulae in first-order predicate calculus, where variables correspond to (individual) variables and predicates correspond to constraints: unary predicates determine domains of variables and other predicates provide relationships between tuples of variables.

For a finite non-empty set X of variables and a finite set of constraints $\mathcal{R}$, a triple $\langle X, \mathcal{R}, C\rangle$ is called *a constraint network schema*, where $X \bigcap \mathcal{R} = \emptyset$ and $C : \mathcal{R} \longrightarrow X^\star$ defines the var-list for every constraint.

For any constraint $R \in \mathcal{R}$, a function I is called its *interpretation* if $I(R) \subseteq D(C(R))$. Assuming the notations above, a *constraint network* is a quintuple $\langle X, \mathcal{R}, C, D, I\rangle$.

A *valuation* of the constraint network $\langle X, \mathcal{R}, C, D, I\rangle$ is a function V that maps any variable $x \in X$ into elements of the universal domain, i.e $V(x) \in D(x)$. A valuation V satisfies the network, and is called its *solution*, if interpretations of all its constraints hold at the values of their var-list, i.e. solution is any element of the intersection

$$\bigcap_{R\in\mathcal{R}} I(R)(V(C(R))).$$

For example, consider a constraint network $\langle\{x\}, \{R_1, R_2\}, C, D, I\rangle$, where $D(x) = \mathbb{N}$, $R_1 : 0 \le x \le 1$

and $R_2 : 1 \leq x \leq 7$. If $I(R_1)(V(x)) = \{0, 1\}$, and $I(R_2)(V(x)) = \{1, 2, ..., 7\}$ then the solution of given network is $I(R_1)(V(C(R_1))) \cap I(R_2)(V(C(R_2))) = \{1\}$.

Two constraint networks are *equivalent* if their solutions are the same.

Constraint satisfaction problem (*CSP*) is given by a constraint network $\langle X, \mathcal{R}, C, D, I\rangle$ and a non-empty set $X_a \subseteq X$ of *asked* variables. The restriction of a solution of the constraint network onto X_a is called a solution of the CSP.

In principle, one may want to find all solutions of a CSP (i.e. to solve the CSP "completely"). However, as the amount of usable computing resources is always limited, in practice *partial constraint satisfaction problem* is considered , which goal is to find just one solution or an optimal solution or at least a good solution according to some given objective function defined in terms of the variables of X. Often the CSP is approximated by setting additional restrictions to the formulas representing expressions and to the interpretation of functional and predicate symbols, and simplifying the task in such a way. Two popular restrictions are: constraints may contain no functional symbols, and domains of potential values of problem variables are finite.

In the framework of SSP, *functional constraint networks* or *computational models* are a specialization of constraint network, where constraints can only be interpreted as functional relations. A $(m+n)$-ary relation R on $D_1 \times D_2$ is *functional* if it is a graph of a function $f : D_1 \longrightarrow D_2$, i.e. if

$$\forall \underline{x} \in D_1 . \exists ! \underline{y} \in D_2 . R(\underline{x}, \underline{y}) \quad .$$

Functional relations are usually defined intentionally in the form $\{\underline{x}, \underline{y} \mid f[\underline{x}] = \underline{y}\}$, where $f[\underline{x}]$ is a term in an interpreted language, which all free variables appear in $\underline{x}$. We will use the notation $\lceil f \rceil$ for the graph of a function f.

Definition 1 *A* functional constraint network schema (FCNS) *is a constraint network schema $\langle X, \mathcal{R}, C\rangle$, where C is a pair of functions $\langle C_{in}, C_{out}\rangle$, such that $C_{in}, C_{out} : \mathcal{R} \longrightarrow X^\star$. $C_{in}(R)$ and $C_{out}(R)$ are called* input-var-list *and* output-var-list *of R, respectively.*

We will write $C(R) = \underline{X} \longrightarrow \underline{Y}$ as a shortcut notation instead of $C_{in}(R) = \underline{X}$ and $C_{out}(R) = \underline{Y}$.

Often the best comprehensible representation of FCNS is its drawing as a directed bipartite graph, where nodes of one sort correspond to variables, nodes of the other sort correspond to constraints and arcs correspond to directed bindings between variables and constraints. Fig. 3 shows the FCNS, where $X = \{x, y, z, u, v\}$, $\mathcal{R} = \{R\}$ and $C(R) = x, y, z \longrightarrow u, v$.

Definition 2 *A* functional constraint network (FCN) *is a quintuple $\langle X, \mathcal{R}, C, D, I\rangle$, where $\langle X, \mathcal{R}, C\rangle$ is a functional constraint network schema, D is a function mapping any variable $x \in X$ into its domain, and I is an interpretation mapping any $R \in \mathcal{R}$ into a functional relation on $D(C_{in}(R), C_{out}(R))$, i.e $I(R) = \lceil R \rceil$.*

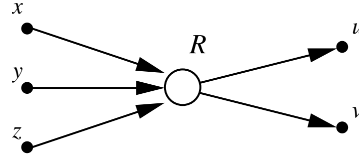


Figure 3. Graph representation of FCNS

NUT performs partial constraint satisfaction by means of algorithms for SSP [4]. One can find a way how specifications written in NUT are translated into FCN suitable for application of different satisfaction techniques in [12, 13].

2.2 Value propagation

Value propagation along a constraint network is based on the direct usage of constraints as functional dependencies. The underlying idea of value propagation is in finding an induced functional constraint on $\longrightarrow \underline{X_a}$.

Value propagation technique can be formally considered as a sequence of equivalent transformations of the FCN. Every such transformation augments the FCN by a new constraint R' derived from the constraints existing in the FCN originally or have been added by previous steps of the propagation. If the FCN is augmented by the constraint R' such that $C(R') = \longrightarrow \underline{X_a}$, then $I(R')(V(\underline{X_a}))$ will provide one possible solution of the CSP. If no new constraints can be added by the transformations, then process will be stopped, and the CSP has no solutions on this FCN (FCNS), independently even from the interpretation of constraints used.

In the simplest case, when a CSP is being solved for the "first order" FCN $\langle X, \mathcal{R}, C, D, I\rangle$, the following three augmentation transformations are needed:

Composition For two constraints $R_1, R_2 \in \mathcal{R}$ such that $C(R_1) = \underline{Z} \longrightarrow \underline{X}$, $C(R_2) = \underline{X} \longrightarrow \underline{Y}$ and $I(R_1) = \lceil f \rceil$, $I(R_2) = \lceil g \rceil$, the FCN is augmented by a constraint R', such that $C(R') = \underline{Z} \longrightarrow \underline{Y}$ and $I(R') = \lceil g \circ f \rceil$.

Selection For a constraint $R \in \mathcal{R}$ such that $C(R) = \underline{Z} \longrightarrow \underline{X}, \underline{Y}$, where $|\underline{X}| = m_1$, $|\underline{Y}| = m_2$ and $I(R) = \lceil f \rceil$, the FCN is augmented by a constraint R', such that $C(R') = \underline{Z} \longrightarrow \underline{X}$ and $I(R') = \lceil \text{select}^{m_1+m_2}_{1,...,m_1} \circ f \rceil$, where $\text{select}^{m_1+m_2}_{1,...,m_1}(x_1, ..., x_{m_1}, y_1, ..., y_{m_2}) = (x_1, ..., x_{m_1})$.

Merging For two constraints $R_1, R_2 \in \mathcal{R}$ such that $C(R_1) = \underline{Z} \longrightarrow \underline{X}$, $C(R_2) = \underline{Z} \longrightarrow \underline{Y}$ and $I(R_1) = \lceil f \rceil$, $I(R_2) = \lceil g \rceil$, the FCN is augmented by a constraint R', such that $C(R') = \underline{Z} \longrightarrow \underline{X}, \underline{Y}$ and $I(R') = \lceil (f, g) \rceil$.

Fig. 4 illustrates the meaning of these transformations. Augmented constraints are represented by dashed lines.

Proposition 1 *Composition, selection and merging are equivalent transformations of FCNs. See proof in [13].*

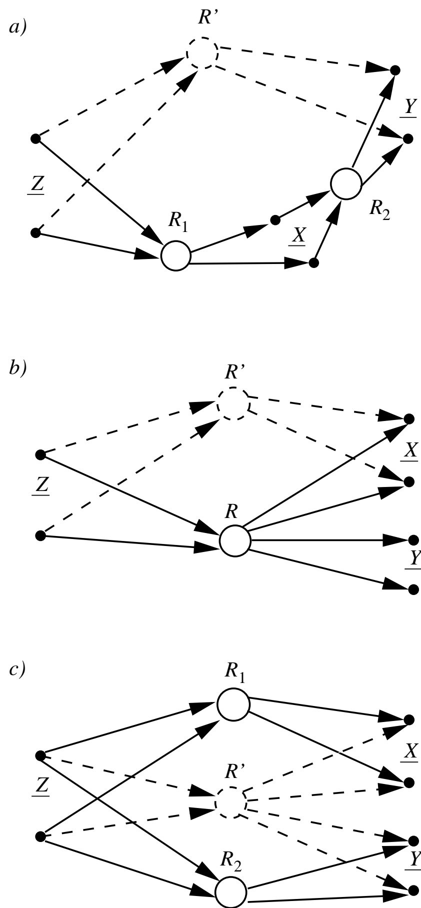


Figure 4. Transformations of FCNS: a) composition, b) selection, c) merging

It is also worth mentioning that value propagation on FCN has a good explanation in terms of constructive logic. All augmentation steps can be regarded as valid inference rules of intuitionistic propositional calculus. So the composition transformation of the FCN is a version of implication elimination, and selection and merging correspond, respectively, to elimination and introduction of conjunction. Thus, the SSP can be smoothly treated as proof search in the theory corresponding to a given CSP. This is a motivation why the value propagation is often called a *deductive synthesis of programs*.

2.3 Synthesis of programs on FCNS

SSP is using a simple and efficient value propagation technique for solving a CSP on the FCN. An interesting observation is that SSP transformations do not require interpretation I to construct a new constraint. This means that the value propagation process can be planned in advance, at the level of FCNS. This fact can be used for designing a very efficient program synthesizer, but one has to take into account the possibility that a program built only by the information of schema may break down during the run-time and will not solve a CSP.

There are several value propagation algorithms that implement different strategies for choosing next transformations to be performed. Most of these algorithms use greedy strategies that augment FCNS only if at least one new variable will be computed (a variable that had no value so far). The most efficient program synthesis algorithm for "first order" FCNS was designed by Dikovsky in 1982. That algorithm has time complexity $\mathcal{O}(L)$, where L is the number of arcs in the FCNS graph. The algorithm and analysis of its complexity both are discussed in [13].

To illustrate the techniques presented , the following toy example of FCNS is used throughout the paper. Let the set of asked variables be $X_a = \{t\}$ and $C(R_1) = x \longrightarrow y$, $C(R_2) = x \longrightarrow y$, $C(R_3) = x \longrightarrow z$, $C(R_4) = z \longrightarrow v$, $C(R_5) = v \longrightarrow z, u$, and $C(R_6) = y, u \longrightarrow t$.

Fig. 5 demonstrates the result of applying several augmentation steps for solving the CSP. Assuming the notation $I(R_i) = r_i$ here, the given FCNS is extended by constraints (in the order of their appearance) R'_1, R'_3, R'_4, R'_5, R'_{5s} and R'_6, which have the following interpretations

$$r'_1 = \lceil r_1 \circ r_0 \rceil$$
$$r'_3 = \lceil r_3 \circ r_0 \rceil$$
$$r'_4 = \lceil r_4 \circ r'_3 \rceil$$
$$r'_5 = \lceil r_5 \circ r'_4 \rceil$$
$$r'_{5s} = \lceil \mathrm{select}_2^{1+1} \circ r'_5 \rceil$$
$$r'_6 = \lceil r'_1 \circ r'_{5s} \rceil$$

This sequence of transformations has a simple translation into some programming language as follows

```
begin
    x:=R0();
    y:=R1(x);
    z:=R3(x);
    v:=R4(z);
    (u,z):=R5(v);
    t:=R6(y,u);
end
```

The fact that all programs constructed as above are sequences of assignments, allows us, in turn, to shorten our notation. Let us use the sequence of indices of related constraints as a representation of the program. In the example above, the synthesized program can be represented as $\langle \mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6} \rangle$ or **013456**. It is easy to see that carrying out the transformation rules in another order results in a different program. For this FCNS, SSP may give programs like **023456**, **034156** or **0134545456**, and they all can solve the problem. Note, that the last program cannot be obtained by greedy algorithms that use every constraint at most once.

Value propagation has certain limitations, such as, only one possible solution of CSP is found, synthesized program may break down during its running, *consistency* of the FCN (all constraints are satisfied by the solution) cannot be guaranteed etc. That is the price for efficiency. In FCNs, where

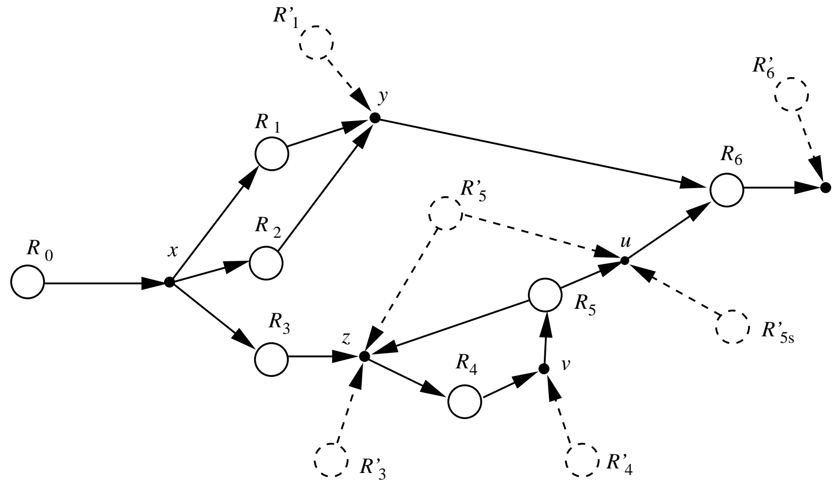


Figure 5. An example of program synthesis

interpretation guarantees consistency SSP is a powerful method for getting truly correct programs that meet their specifications. This has been used for automatization of computations in several engineering fields [12, 11]. In practice, FCN is generalized to allow variables to take higher-order (relational or functional) values [13]. The corresponding logic calculus is complete in the intuitionistic propositional calculus, the fact that was first stated by G. Mints [4].

3. Program synthesis as an optimization task

3.1 Reformulating problem specification

Different solutions of CSP may give different values for the asked variables $X_a \subseteq X$. The users of systems are often unhappy about randomly chosen solution, they need the result best conforming to their needs or all solutions satisfying a certain level of "fitness".

There are two serious obstacles to accommodate this natural request. First, we need to have a *measure of fitness of solution* that should be an effectively computable function $s : \Pi \longrightarrow \mathbb{R}$, where Π is a set of all possible programs. By the *program* we mean a sequence of instructions, i.e. a sequence of transformations used for constructing the corresponding augmented constraints. Second, we require a method for finding the best programs in terms of the measure s or finding the subset of programs $\Pi = \{p \in \Pi | s(p) \geq l\}$ at a given level of fitness l, i.e. a method for finding extremal points of the real function. In general, these optimization tasks have no complete solutions and can be solved either for special cases or as approximations obtained by means of different heuristics.

In this paper, we do not discuss various fitness measures and their properties. However, to illustrate the method considered, let us assume that the fitness is given by a finite set of sample input-output data for the best program. In practice, it might be a set of measured or observed values of some physical phenomena, parameters or aspects of a complex process or a system that have to be modeled by the target program. In other words, measurements give an empirical correspondence (relation) between components of the system and we will treat those programs better fitting, which keep this correspondence during their run.

Example Let us consider a FCN $\langle X, \mathcal{R}, C, D, I\rangle$ the schema of which is presented in Fig. 5, a constraint R' added to the FCN by a series of augmentation steps and a finite extensionally defined constraint P, where $C(P) = C(R')$. We call P an *observed constraint*. The constraint R' matches a program implementing a functional dependency from x to t induced by the set of constraints $\mathcal{R}$. There is a number of different such added constraints R' corresponding to different augmentation sequences. Let us take such a program p that will produce for any $x \in D(x)$ the value of $t = p(x)$. Assume that the desired behavior of the program p should be as close as possible to an observable physical system with parameters x and t. Suppose that one has made n experiments with the system and measured these parameters. Results of the measurements give the observed constraint $P = \{(x_1, t_1), \ldots, (x_n, t_n)\}$, where x_i and t_i denote measured values of x and t during the same experiment ("at the same time"). A fitness measure is given to any point in the space of solutions of FCN by its distance to the observed constraint. Depending on the domain and the problem to be solved, different distance functions can be used. In our example, a square root error fitness measure (alias score) of the program p is used for a distance measure and computed as

$$s(p) = \frac{\sum_{i=1}^{n} (p(x_i) - t_i)^2}{n-1} + \alpha\, l(p) \ , \qquad (1)$$

where

- $l(p)$ – a component of a fitness measure, which measures the length of the program p by calculating the number of transitions through the FCN
- α – a small floating-point coefficient (e.g. $\alpha = 0.0001$). Note that $\alpha l(p)$ is a length penalty that is used to get better score for the shortest among the programs, which have the same input-output behavior. □

In order to take advantage of the existing optimization techniques and heuristics elaborated for search of extremum of functions of a real variable, an effective coding of programs is required.

3.2 Coding of Program Structures

We also need to construct a function $h : [0, 1) \to \Pi$ that allows to reformulate the problem of program construction to finding the minimal value of the function $f(z) = s(h(z))$. The optimization technique used in the paper require that the function $h(z)$ should be defined for any value $z \in [0, 1)$. Continuity or differentiability of function $h(z)$ is not needed.

A naive way to construct a function $h(z)$ is to represent z in a positional representation system with a base m, where $m = |\mathcal{R}|$ and to use digits of its fractional part as labels of corresponding constraints $R \in \mathcal{R}$ used for transformation of the FCN. For example, if the base of representation system is 7 and $z_7 = 0.324516$, we would get immediately a sequence of the applicability of constraints, i.e the program $p = h(z_m) = h(0.324516) =$ **324516**.

However, this simple method cannot be used in practice as it produces infinite program codes for irrational numbers. It also can create sequences of constraints that do not satisfy the data-flow defined by the FCN, for example, sequence **5614** is not accepted by FCN presented in Fig. 5. In the latter case, one can introduce a large score for such programs and in this way to suppress them within the minimization process. However, this would result in a huge search space and a very inefficient process for problem solution. For our example, it is easy to check that acceptable sequences satisfying data-flow constraints only make about 1% of all possible sequences.

Function $h(z)$ can be modified so that any real number within the interval $[0, 1)$ would be mapped into an accepted program. We introduce a *state transition machine* $\mathcal{M}(P)$ representing all program control flows allowed for the observed constraint P accepted by our FCN (see Fig. 6).

The *states* of the machine represent sets of variables, the *arcs* correspond to transformations according to particular constraints. Here multiple transitions between the two states marked with different labels are drawn as one arrow labeled with the list of constraints. The states containing only variables from $C_{in}(P)$ and $C_{out}(P)$ are called *root* and *result* respectively. Any path from *root* to *result* is a sequence of constraints that solve the CSP. Thus, in our case, the problem statement is: $x \to\ t$ and, for example, the following accepted programs could be obtained: **13456**, **21324351561**,

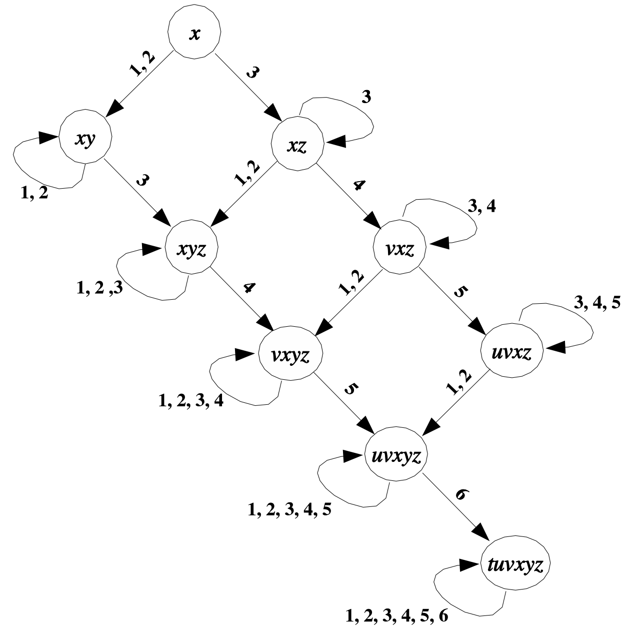


Figure 6. State transition graph $\mathcal{M}(P)$

234545456 etc. If the subroutines implementing constraints $\lceil R_1 \rceil, ..., \lceil R_6 \rceil$ do not have "side-effects" (interaction with user or environment of the system, computations depending on hidden parameters, cycles etc.) then the calling of such subroutines repeatedly does not change valuation of variables, and the first and the second sequences would produce the programs with the same input-output behavior.

object	components and meanings
model	state transition machine $\mathcal{M}(P)$
ε	empty program
root	problem inputs $C_{in}(P)$
result	problem outputs $C_{out}(P)$
R	*constraint*, where *R.id* is identifier of *R* *R.inputVar* is input-var-list of *R* *R.outputVar* is output-var-list of *R*
reachedState	set of computed variables
arcsOut	set of outgoing arcs from the *reachedState*
state	state of $\mathcal{M}(P)$, where *state.varSet* is a set of variables *state.rId* is a label of ingoing arc
statesSet	set of all possible expansion *state*s from the *reachedState*
path	sequence of *constraint*s
bounds	bounds of a "running" interval, where *bounds.L* is leftmost points of *bounds* *bounds.R* is rightmost points of *bounds*
iterNum	number of transitions through the $\mathcal{M}(P)$

Table 1. Objects explanation

Let us introduce now an algorithm for construction of a program p for z. The program p for the given z can be found by applying the following subroutine with the proper actual parameters: $p \leftarrow$ **roadNetworks**$(z, \mathcal{M}(P), C_{in}(P), C_{out}(P))$. An explanation of the used notations is brought in a Table 1, the library function **append**$(list, b)$, where $list = \langle a_1, \ldots, a_n \rangle$, returns a new list $\langle a_1, \ldots, a_n, b \rangle$.

```
function roadNetworks(z, model, root, result) is
1 : Initialization
  bounds.L ← 0;
  bounds.R ← 1;
  iterNum ← 0;
  state ← root;
2 : Finding a program p
  while result ⊈ state do
   2a : Finding all outgoing arcs
     arcsOut ← sToArcsOut(state, model);
   2b : Finding all possible expansion states
     statesSet ← sToStatesSet(model, state, arcsOut);
     if statesSet ≠ ∅ then
        h(z, state, statesSet, bounds, path);
        iterNum ← iterNum + 1;
     endif
     if iterNum = iterNumCritical then
         path ← ε; return(path);
     endif
  endwhile
  2c : Return a program p
  return(path);
endfunction
```

To find $arcsOut$ the following subroutine is used:

```
function sToArcsOut(reachedState, model) is
arcsOut ← ∅;
forall R ∈ model do
  if R.inputVar ⊆ reachedState
    then arcsOut ← arcsOut ∪ R.id;
  endif
enddo
return(arcsOut);
endfunction
```

Construction of $reachedState$ has the following form:

```
function sToStatesSet(model, reachedState, arcsOut) is
statesSet ← ∅;
forall R ∈ arcsOut do
    state.varSet ← reachedState ∪ R.outputVar;
    state.rId ← R.id;
    statesSet ← append(statesSet, state);
enddo
return(statesSet);
endfunction
```

Finding the sequence of transformations to be included into a program p is arranged while traversing the machine $\mathcal{M}(P)$. There is a narrowing technique to determine for a given z the interval to which it belongs. Any movement by an arc of $\mathcal{M}(P)$ means appending the corresponding transformation (i.e. constraint) to the program and narrowing the interval.

```
proc h (z, reachedState, statesSet, bounds, path) is
s ← #statesSet;
if s > 0 then ω0 ← (bounds.R − bounds.L) /s;
endif
forall i ∈ {1, ..., s} do
 if z ∈ [bounds.L + ω0 · (i − 1), bounds.L + ω0 · i)
   then state ← statesSet_i;
        reachedState ← state.varSet;
        path ← append(path, state.rId);
        bounds.L ← bounds.L + ω0 · (i − 1);
        bounds.R ← bounds.L + ω0 · (i);
 endif
enddo
endproc
```

To illustrate the construction of a program p for the state transition machine $\mathcal{M}(P)$ in Fig. 6, the interval $[0, 1)$ is divided into equal subintervals: $[0, 0.33)$, $[0.33, 0.66)$ and $[0.66, 1)$ that correspond to constraints identifiers **1**, **2**, and **3**, respectively. Thus, for $z = 0.88534$ it means that the first constraint in the program p should be **3**. This **3** transition leads to the state xz that also has three outgoing arrows. The interval $[0, 66, 1)$ will be consequently divided into three: $[0.66, 0.769)$, $[0.769, 0.878)$ and $[0.878, 1)$ corresponding to **1**, **2**, and **4**. Thus, the second transition in the program p should be **4**, etc.

Usually shorter codes are preferred, thus to avoid having a program with endless cycles or cycles with large number of iterations an extra condition $iterNum = iterNumCritical$ is used, where $iterNumCritical \gg m$. But there may also be other considerations why one program has priority over another. To specify these heuristics, we propose to add weights to transitions of the machine $\mathcal{M}(P)$. It is assumed

that bigger weights are attached to the transitions with higher preference and the sum of weights of all outgoing arcs of a state equals to one. So the size of a new interval is proportional to the weight attached to the arrow passed. To exclude "useless" transitions that do not produce new values to the components of the model, some weights may be zero. In our example, the arrows with zero weight are not shown and the weights of the remaining ones outgoing from one state are supposed to be equal (see Fig. 7).

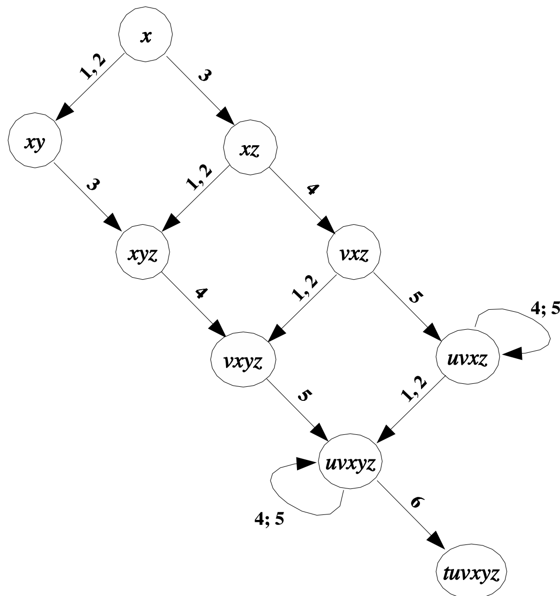


Figure 7. Weighted state transition graph

The notation **4; 5** means a composed transitions corresponding to the sequential application of constraints identified by **4** and **5**, which is only a cyclic running of the program transitions that change values of variables.

Proposition 2 *For the given FCN(FCNS) and a valid observable constraint P, there exists an effective procedure for construction of the state transition machine $\mathcal{M}(P)$.*

Proof. Let us assume that $n = |X|$, $m = |\mathcal{R}|$ and $k = |C_{in}(P)|$. Any state of the machine $\mathcal{M}(P)$ contains variables of the constraint P. Thus, $\mathcal{M}(P)$ will have at most 2^{n-k} states. For any state, at most m constraints have to be checked up, which may induce transitions from the state. Hence, a naive construction algorithm for the machine $\mathcal{M}(P)$ has exponential complexity.

□

Proposition 3 *Two different programs solving FCNS for an observable constraint P have different codes, i.e. $h(z) \neq h(z')$ implies $z \neq z'$.*

Proof. This is a direct consequence from the fact that the machine $\mathcal{M}(P)$ is deterministic. Indeed, for any state $s \in X^\star$ and a constraint R with $C_{in}(R) \subseteq s$, there are two exclusive cases – either $C_{out}(R) \subseteq s$ (the looping transition $s \longrightarrow s$ is included into $\mathcal{M}(P)$) or $C_{out}(R) \cap s \neq C_{out}(R)$ (there exists the transition from s to $t = C_{out}(R)$).

□

The construction of $\mathcal{M}(P)$ guarantees that it may cycle only locally (in one state). To avoid this kind of diverging computation of function $h(z)$, it is reasonable to modify the algorithm above with dynamic weights so that the probability of continuation via a looping transition would decrease with every cycle. The simplest way to cut down infinite computations is to assign a counter of cycles to any loop of the state transition graph $\mathcal{M}(P)$ and after a certain number of cycles to prune that transition by setting its weight to zero. Note that this alteration does not affect minimization if the limit for the number of cycles is chosen large enough. Longer paths in the transition graph mean also a larger number of transitions and this increases the score of the program due to the length penalty in the formula (1). Relying on this modification and the properties above it is easy to conclude that the following statement holds.

Corollary 3 *Function $f(z) = s(h(z))$ is a piecewise constant function computable for any $z \in [0, 1)$. The set of constant pieces of function f is enumerable.*

3.3 Optimization Techniques

The last corollary sets certain limits to the optimization techniques suitable for the minimization of function f. Methods based on continuity and differentiability of the object function cannot be used here. Stochastic optimization algorithms, such as random walking, simulated annealing or genetic programming might help instead.

To experiment with program construction by means of optimization techniques, evolutionary approach was chosen. Evolution Strategies (ESs) are algorithms, which imitate the principles of natural evolution as a method to solve the parameter optimization problems. Rainer Storn and Kenneth Price first introduced the new heuristic approach for minimizing possibly nonlinear and non-differentiable continuous space functions in 1995 [8]. Differential Evolution (DE) can be classified as *a simple Evolutionary Strategy* for optimizing real-valued multi-modal objective functions. The algorithm has proven to be efficient, effective, and the robust optimization technique that outperforms traditional Genetic Algorithms, especially in the case of problems containing continuous problem variables. The convergence rate of the floating-point encoding DE algorithm is more than 10 times higher than the convergence rate of the traditional binary encoded Genetic Algorithm [9]. DE has been successfully applied in various fields, one can find a number of applications of this algorithm in [7].

As with all evolutionary optimization algorithms, DE works with a population of solutions, not with a single solution for the optimization problem. More detailed information about how DE works can be found in [10]. For the most part

of our investigations the simple one-dimensional version of DE-algorithms *DE/rand/1* is used.

More precisely, DE is a specialized search method for our case that uses NP different values (called *entities*) $z_{i,G} \in [0,1)$, $i = 1,2,\ldots,NP$ for each generation G. DE generates a new generation of entities $G+1$ by performing sequentially three operations: *mutation*, *crossover* and *selection*.

Mutation For each entity $z_{i,G}$, a *perturbed entity* is generated according to $v_{i,G+1} = z_{r_1,G} + F(z_{r_2,G} - z_{r_3,G})$, where $r_1, r_2, r_3 \in \{1,2,\ldots,NP\}$ are randomly selected and satisfy: $r_1 \neq r_2 \neq r_3 \neq i$. $F \in [0,2]$ is a control parameter of the amplification of the difference vector $(z_{r_2,G} - z_{r_3,G})$.

Crossover The target entity $z_{i,G}$ is transformed into a trial entity

$$u_{i,G+1} = \begin{cases} v_{i,G+1} & \text{if} \quad r \leq CR, \\ z_{i,G} & \text{otherwise} \end{cases} .$$

$r \in [0,1]$ is the next in turn evaluation of a uniform random number generator and CR is the crossover constant from the interval $[0,1]$.

Selection If and only if the trial vector $u_{i,G+1}$ yields a better cost function value compared to $z_{i,G}$, it is accepted as a new parent entity for the following generation $G+1$. Otherwise, the target entity is retained to serve a parent entity in generation $G+1$ once again.

The evolution process converges to a generation where all entities have the same value of fitness function. Extracting distance and direction information from the population to generate random deviations results in an adaptive scheme with a good precondition to converge to global minimum of the object function.

The goal of this paper has been to give the idea on program construction. Finally, we shall give a report to some preliminary experiments in which we have implemented simple examples that illustrate this idea. Given an observed constraint P (i.e. input-output parameters of target program) and defining the interpretation of the model, the aim of the algorithm is to find a proper sequence of constraints that can solve a problem.

Example 1 Let us consider a FCN the schema of which is presented in Fig. 5 and deliver an optimal solution for the problem $P : x \longrightarrow t$. The desired input-output behavior of the program to be constructed is given by the observable constraint $P = \{(4.025, 215),\ (5.7, -1227),\ (6.4, 4285),\ (6.8, -3170),\ (7.1, -7852),\ (7.3, 1065),\ (7.4, 7952),\ (7.5, 13075),\ (7.6, 13192),\ (7.8, -5704),\ (7.95, -20838)\}$

Let us have the following interpretation of constraints

$I(R_1)$ = " `y := 2*x` "
$I(R_2)$ = " `y := 2+x` "
$I(R_3)$ = " `z := x/2` "
$I(R_4)$ = " `v := z/2` "
$I(R_5)$ = " `(u, z) := ((-v)^3,(-v)^4)` "
$I(R_6)$ = " `v := exp(y)*cos(u)` "

DE implemented in system NUT was used in the experiments. Using DE parameters $NP = 10$, $F = 0.8$ and $CR = 0.5$, several "optimal" solutions like 0.645527, 0.513938, 0.608810, 0.658636 that correspond to "**23456**" or 0.9999 that corresponds to "**34526**" were found. All these programs had a score of 0.114650. In fact, the exact solution was found very quickly. The minimization process converged on average by 4 generations (that is not surprising for a small toy example).

Non-optimal solutions, for example, are 0.968355 ($s(z) = 60.5 \cdot 10^{11}$) that correspond to "**3454516**" or 0.455363, 0.831913 ($s(z) = 14.9 \cdot 10^7$) that correspond to "**2345456**" and "**3245456**" respectively, or 0.764984, 0.252132 ($s(z) = 94.9 \cdot 10^{11}$) that correspond to "**31456**" and "**13456**" respectively. In Fig. 8 the difference of optimal and non-optimal solutions is illustrated.

□

Example 2 The result of optimization is rather fragile in the sense that small changes of the computational model or interpretation of relations would lead to the result of a completely different flame. In the sequel we have changed the model in Fig. 5 by adding one arc (from the node x to R_4) and given a new interpretation to the relations as follows

$I(R_1)$ = " `y := ln(x)` "
$I(R_2)$ = " `y := exp(-x)` "
$I(R_3)$ = " `z := x^3+6*x` "
$I(R_4)$ = " `v := 2*(x^2 + z)` "
$I(R_5)$ = " `(u, z) := (v+6,v/2)` "
$I(R_6)$ = " `v := 0,16*y*u/7,85^4` "

The problem was specified by the following bivariate numeric data set

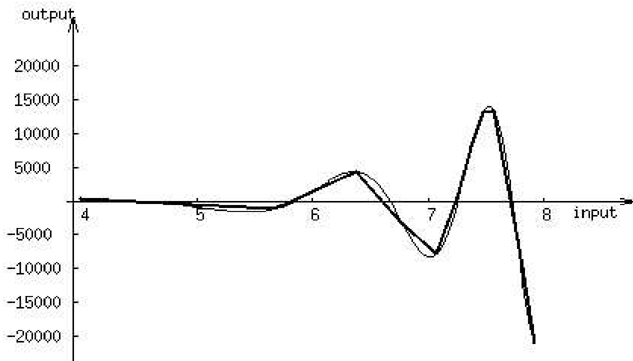
$$P = \{(0.75, 1.87 \cdot 10^{-07}),\ (0.70, 1.04 \cdot 10^{-07}),\ (0.65, 5.44 \cdot 10^{-08}),\ (0.60, 2.61 \cdot 10^{-08}),\ (0.55, 1.17 \cdot 10^{-08}),\ (0.50, 3.40 \cdot 10^{-09})\}$$

collected from Patrick M. Murphy Repository Librarian [5]. This data collection corresponds to the dependence of total energy on the wavelength. The optimal solutions of the problem obtained by DE method corresponds to the proposed by Knut Ångström [1] functional relationship

$$E(\lambda) = 0.016e^{-t}(t^3 + 3t^2 + 6t + 6)/7.85^4 ,$$

where $t = 7.85/\lambda$.

a) Program "**34526**", $s(z) = 0.114650$



b) Program "**3245456**", $s(z) = 14.9 \cdot 10^7$

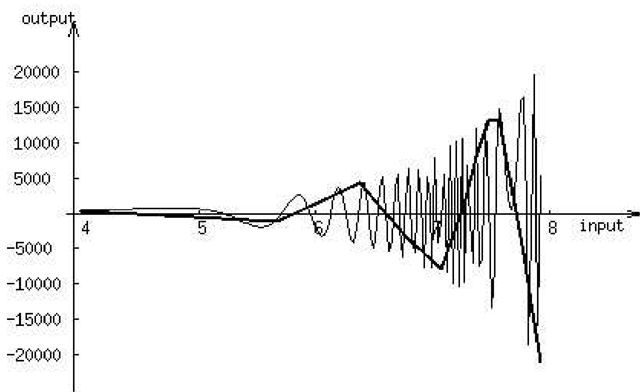


Figure 8. Comparison of two entity programs: a) optimal solution, b) non-optimal solution

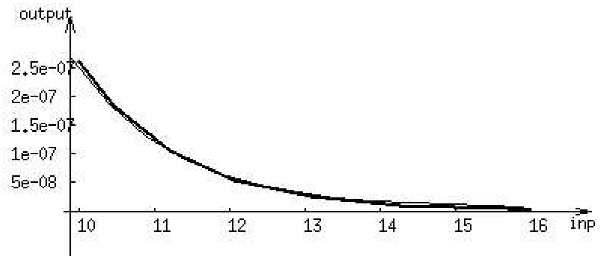


Figure 9. Optimal solution for function $E(\lambda)$ **(**$s(z) = 2.75874 \cdot 10^{-24}$**)**

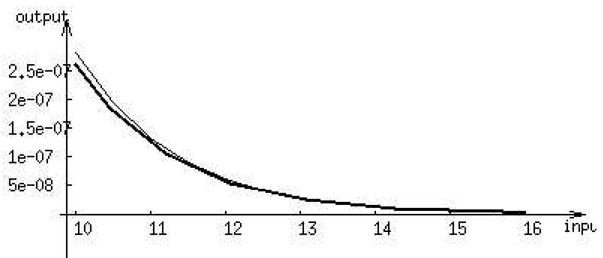


Figure 10. Non-optimal solution for function $E(\lambda)$ **(**$s(z) = 0.360136$**)**

In the experiments with that problem, we leave the DE-algorithm parameters unchanged. The average number of function evaluations required to achieve the "optimal" solutions like 0.376362, 0.383618, 0.39116, 0.40419 or 0.795112 that correspond to the following sequence of relations "**234545456**" is 130. All these programs have a score of $2.75874 \cdot 10^{-24}$. After 12 generations (i.e. 130 function evaluations) we reach the solution in Fig. 9.

Examples of graph outputs for non-optimal solutions 0.360136 and 0.847546 are depicted on Fig. 10 and Fig. 11.

4. Conclusions

First experiments with DE algorithms suggest that stochastic optimization algorithms could be a powerful tool in the automation of program construction. To compare with deductive program synthesis, the specification (FCN) has to be extended by some experimental data that define fitness of different program entities for the candidate of the best solution. Another important difference is that only interpreted FCNS can be used for computing a fitness measure. Therefore, it is impossible here to compose a scheme of a program with free interpretation as a result of the pre-planning process. The approach based on optimization solves the task of program synthesis through the generalization of sample input-output values. So it can be called *inductive program synthesis*.

The inductive method itself is not complete and it cannot achieve the efficiency of deductive program synthesis. At the same time, inductive approach can promote control over the synthesis process and can take appropriate decisions when several solutions are available.

Inductive programs synthesis could be treated as a genetic programming technique. As a case study for finding the relationship between parameters in response to a given set of inputs, a non-linear system identification problem was investigated by the authors [6]. We have used *Evolutionary Algorithm* **e** workbench [2] for performing these experiments.

During the experiments in most cases unbiased estimates were obtained and the EA was able to find a model of the system (i.e. to solve a symbolic regression problem). So we can say that the linear genetic programming technique implemented in **e** produced satisfactory results for the current symbolic regression problem. However, based on a simple application, it is insufficient to make generic comments on the performance of the linear GP technique. Though, with a certain degree of confidence, we can say that search methods like GP do not allow obtaining a model with an assumed structure because of a predefined set of instructions of the underlying abstract machine that is used as genetic material

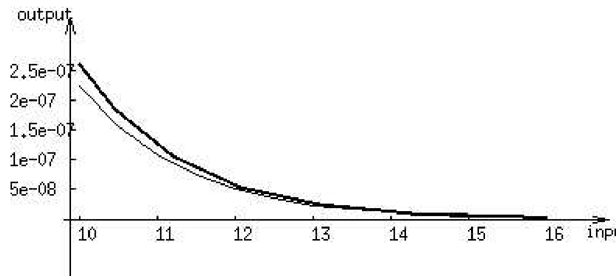


Figure 11. Non-optimal solution for function $E(\lambda)$ ($s(z) = 0.847546$)

in the evolution process. Exploiting user defined constraints from the problem/system specification would improve this aspect and provide control over the structure and properties of the generated program.

Though our approach provides an abstract framework for program construction by means of evolutionary strategies for optimization, many experimental and theoretical studies are to be continued. There is a number of heuristics to be refined to achieve the most efficient solution. A new method is to be developed towards using of high-order FCN and its smooth integration with the deductive method. Definitely, engineering methods and skills have to be developed to acquire powerful specifications of tasks via computational models, training data, etc.

5. Acknowledgements

The authors are grateful to Prof. Enn Tyugu for his comments and valuable suggestions. This work was partially financed by Estonian Science Foundation (grant #5567).

6. References

[1] K. Ångström. Energy in the visible spectrum of the hefner standard. *Physical Review*, XVII:312, 1903.

[2] W. Dwinnell. *e*, an evolutionary learning tool. *PC AI Magazine*, 11(5):38–40, 1997.

[3] M. Matskin and E. Tyugu. Strategies of structural synthesis of programs and its extensions. *Computing and Informatics*, 20(1):1–25, 2001.

[4] G. Mints and E. Tyugu. Justification of the structural synthesis of programs. *Science of Computer Programming*, 2(3):215–240, 1982.

[5] P. M. Murphy and D. W. Aha. Uci repository of machine learning databases, URL: http://www.ics.uci.edu/∼mlearn/mlrepository.html, 1994.

[6] J. Sanko and J. Penjam. Program construction in the context of evolutionary programming. In *Proc. of Andrei Ershov Fifth International Conference "Perspectives of System Informatics", PSI'03, Novosibirsk, Russia, June 2003*, volume 2890 of *Lecture Notes in Computer Science*, pages 50–58. Springer-Verlag, Berlin, 2003.

[7] R. Storn and K. Price. Web site of price and storn as on July, 2000. URL: http://www.icsi.berkeley.edu/∼storn/code.html.

[8] R. Storn and K. Price. Differential evolution - a simple and effective adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, International Computer Science Institute (ICSI), Berkeley, CA, 1995.

[9] R. Storn and K. Price. Minimizing the real functions of the icec 96 contest by differential evolution. In *Proc. of IEEE International Conference on Evolutionary Computation*, pages 842–844, Nagoya, 1996.

[10] R. Storn and K. Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[11] E. Tyugu. *Knowledge-Based Programming*. Turing Inst. Press Knowledge Engineering Tutorial Series. Addison-Wesley Publ. Co., Wokingham, 1988.

[12] E. Tyugu, M. Matskin, and J. Penjam. Applications of structural synthesis of programs. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. of World Congress on Formal Methods in the Development of Computing Systems, FM'99, Toulouse, France, 20–24 Sept. 1999, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 551–569. Springer-Verlag, Berlin, 1999.

[13] E. Tyugu and T. Uustalu. Higher-order functional constraint networks. In B. Mayoh, E. Tyugu, and J. Penjam, editors, *Proc. of NATO ASI on Constraint Programming, Pärnu, Estonia, 13–24 Aug 1993*, volume 131 of *NATO ASI Series F*, pages 116–139. Springer-Verlag, Berlin, 1994.

Jaan Penjam (b. 1955) graduated from Tartu University in 1979 as mathematician. Currently, he is a professor of theoretical computer science at Tallinn Technical University and director of the Institute of Cybernetics. His scientific interests vary from semantics of programs and computational logic to artificial intelligence, constraint programming and high performance computing on networks.

Jelena Sanko obtained her Diploma and M.S. degree in computer and system engineering from the Tallinn University of Technology in 1998 and 2000, respectively. Currently, she is a PhD student in the Institute of Cybernetics at the same university. Her area of research involves program synthesis and evolutionary computations.