Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

A note on Computing SAGBI-Gröbner bases in a Polynomial Ring over a Field

Hans Öfverbeck

Abstract

In the paper [2] Miller has made concrete Sweedler's theory for ideal bases in commutative valuation rings (see [5]) to the case of subalgebras of a polynomial ring over a field, the ideal bases are called SAGBI-Gröbner bases in this case. Miller proves a concrete algorithm to construct and verify a SAGBI-Gröbner basis, given a set of generators for an ideal in the subalgebra. The purpose of this note is to present an observation which justifies substantial shrinking of the so called syzygy family of a pair of polynomials. Fewer elements in the syzygy family means that fewer syzygy-polynomials need to be checked in the SAGBI-Gröbner basis construction/verification algorithm, thus decreasing the time needed for computation.

1 Introduction

SAGBI-Gröbner theory is a generalisation of Gröbner theory to subalgebras of a polynomial ring. Thus we consider a fixed subalgebra A of a polynomial ring $k[X] = k[x_1, \ldots, x_n]$ over a field k, and we want to do Gröbner theory in the subalgebra A.

In Gröbner basis theory a so called *S-polynomial* of a pair (f, g) of polynomials is defined as (see the next section for the exact definitions of the notation):

$$S(f,g) = L_1 f - L_2 g, \tag{1}$$

where

$$L_1 = \frac{\mathrm{lcm}(\mathrm{lp}(f), \mathrm{lp}(g))}{\mathrm{lt}(f)}, \qquad L_2 = \frac{\mathrm{lcm}(\mathrm{lp}(f), \mathrm{lp}(g))}{\mathrm{lt}(g)}.$$

In SAGBI-Gröbner basis theory the analogue of a S-polynomial of a pair is the *syzygy family* of a pair (f, g). As the name indicates, the syzygy family usually consists of more than one element, but all the elements have the form (1) for some $(L_1, L_2) \in A^2$ such that $\mathrm{lt}(L_1 f) = \mathrm{lt}(L_2 g)$. The purpose of this note is to prove that when constructing the syzygy family we need only consider polynomials of the form (1) where $(L_1, L_2) \in A^2$ are such that $\mathrm{lp}(g)$ does not divide $\mathrm{lp}(L_1)$ and $\mathrm{lp}(f)$ does not divide $\mathrm{lp}(L_2)$ in $\mathrm{Lp}(A)$.

This yields a substantially smaller syzygy family than what is indicated in [2].

2 Notation

Since the purpose of this note is to refine a result in the article [2] we try to follow the notation there as closely as possible. Let $k[X] = k[x_1, \ldots, x_n]$ be multivariate polynomial ring over a field k. Suppose we have a term order on $k[X]$, then for a polynomial $p \in k[X]$, $\mathrm{lp}(p)$ denotes the leading X-power product of p, $\mathrm{lc}(p)$ the leading coefficient of p, and $\mathrm{lt}(p) = \mathrm{lc}(p)\mathrm{lp}(p)$ the leading term of p. If $S \subseteq k[X]$, then $\mathrm{Lp}(S)$ denotes $\{\mathrm{lp}(s) | s \in S\}$.

If $w = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$ is an X power product then the *multidegree*, $\mathrm{mdeg}(w)$, of w is defined as $\mathrm{mdeg}(w) = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbb{N}^n$. For a polynomial $f \in k[X]$ we define $\mathrm{mdeg}(f) = \mathrm{mdeg}(\mathrm{lp}(f))$.

For a vector $v = (v_1, \ldots, v_m) \in \mathbb{N}^m$ and a (implicitly ordered) set $S = \{s_1, \ldots, s_m\} \in k[X]$ with m elements we define:

$$S^v = \prod_{j=1}^{m} s_j^{v_j}.$$

Let A be fixed subalgebra of $k[X]$, then $\mathrm{Lp}(A)$ is a multiplicative monoid. For an ideal I in A, $\mathrm{Lp}(I)$ is a monoid-ideal in $\mathrm{Lp}(A)$. A *SAGBI-Gröbner basis* for an ideal I in A is a subset $G \subseteq I$ such that $\mathrm{Lp}(G)$ generates $\mathrm{Lp}(I)$ as a monoid-ideal in $\mathrm{Lp}(A)$.

For an ideal J in $k[X]$ an ordinary Gröbner basis is a subset $G' \subseteq J$ such that $\mathrm{Lp}(G')$ generates J as a monoid ideal in $\mathrm{Lp}(k[X])$. This corresponds to the special case $A = k[X]$ for SAGBI-Gröbner bases, thus we can say that SAGBI-Gröbner bases are a generalisation of Gröbner bases. On the other hand SAGBI-Gröbner bases are a special case of the even more general bases presented in [5] and [4].

Throughout this article we assume that we have a finite SAGBI basis $F = \{f_1, \dots, f_m\}$ for the subalgebra A, i.e. $F \subseteq A$ and $\mathrm{Lp}(F)$ generates $\mathrm{Lp}(A)$ as monoid.

When dealing with ideals in the subalgebra A we need an analogue of ordinary reduction which takes into account the fact that we work inside a subalgebra, the analogue is called SI-reduction.

Definition 1 (SI-reduction) *Let $G \subseteq A$. A polynomial $h \in A$* SI-reduces via *G to $h' \in A$ in one step if there is a nonzero term cX^α of h for which there exists $g \in G$ and $a \in A$ such that $\mathrm{lt}(ag) = cX^\alpha$ and $h' = h - ag$. If there is a chain of one-step reductions from h to h'' via G, then we say that h* SI-reduces *to h'' via G.*

3 Shrinking the syzygy family

Consider the intersection, $\langle \mathrm{lp}(g)\rangle \bigcap \langle \mathrm{lp}(h)\rangle$, of the monoid ideals generated by $\mathrm{lp}(g)$ and $\mathrm{lp}(h)$ in $\mathrm{Lp}(A)$. The intersection is again a monoid ideal in $\mathrm{Lp}(A)$, which plays a central part in the definition of the syzygy family:

Definition 2 (Definition 4.1 in [2]) *Given $g, h \in A$ and a generating set $T_{g,h}$ in $\mathrm{Lp}(A)$ for $\langle \mathrm{lp}(g)\rangle \bigcap \langle \mathrm{lp}(h)\rangle$, a* syzygy family *for g and h is a set that contains, for each $t \in T_{g,h}$ a polynomial of the form $a_t g - b_t h$ with $\mathrm{lt}(a_t g) = \mathrm{lt}(b_t h) = \mathrm{lt}(c_t t)$ for some $c_t \in k$.*

Consider Corollary 4.6 in [2]; there we are told that a syzygy family for g and h can be constructed in the following way:

Let $\mathcal{V}$ be a finite generating set of the monoid of nonnegative integer solutions $v = (v_1, v_2, \dots, v_{2m+2})$ of:

$$v_1 \mathrm{mdeg}(g) + \sum_{j=1}^{m} v_{j+1} \mathrm{mdeg}(f_j) = \sum_{j=1}^{m} v_{m+1+j} \mathrm{mdeg}(f_j) + v_{2m+2} \mathrm{mdeg}(h) \tag{2}$$

where $\{f_1, \ldots, f_m\} = F$ is our SAGBI basis for A.

A minimal generating set of the nonnegative solutions of a diophantine system such as (2) is sometimes called a *Hilbert basis* for the solutions. There exist several algorithms to calculate the Hilbert basis, e.g. those described in [1] and [3], this allows us to effectively compute $\mathcal{V}$.

For an element v of $\mathcal{V}$ we let $v^l = (v_1, \ldots, v_{m+1})$ and $v^r = (v_{m+2}, \ldots, v_{2m+2})$, then v is called the *parent vector* of v^l and v^r. Let

$$\mathcal{V}' = \{v \in \mathcal{V} \,|\, v_1 = v_{2m+2} = 1\}$$

and

$$\mathcal{V}'' = \{u + v \,|\, u \in \mathcal{V}_1, v \in \mathcal{V}_2\}$$

where $\mathcal{V}_1 = \{u \in \mathcal{V} \,|\, u_1 = 1, u_{2m+2} = 0\}$ and $\mathcal{V}_2 = \{v \in \mathcal{V} \,|\, v_1 = 0, v_{2m+2} = 1\}$, and let

$$\mathcal{PV} = \mathcal{V}' \cup \mathcal{V}''.$$

Finally let $G = \{g, f_1, \ldots, f_m\}$ and $H = \{f_1, \ldots, f_m, h\}$, (where $f_1, \ldots, f_m$ are the elements of our SAGBI basis F) then by Corollary 4.6 in [2] a syzygy family for g and h is formed by all polynomials of the form

$$s_v = \operatorname{lc}(H^{v^r}) \cdot G^{v^l} - \operatorname{lc}(G^{v^l}) \cdot H^{v^r}$$

where $v \in \mathcal{PV}$.

The purpose of this note is to prove that in the definition of $\mathcal{PV}$ we can remove the second set from the union and let $\mathcal{PV} = \mathcal{V}'$ and the only price we have to pay for this reduction is to add 0 to the syzygy family.

Theorem 1 (Refinement of Corollary 4.6 in [2])
Let $G = \{g, f_1, \ldots, f_m\}$ and $H = \{f_1, \ldots, f_m, h\}$, let $\mathcal{V}$ be a finite generating set for the monoid of nonnegative solutions of the system of equations (2) and let $\mathcal{PV} = \mathcal{V}'$. Then the set S consisting of 0 *and all polynomials of the form $s_v = \operatorname{lc}(H^{v^r}) \cdot G^{v^l} - \operatorname{lc}(G^{v^l}) \cdot H^{v^r}$, where the parent vector v of v^l and v^r lies in $\mathcal{V}'$, is a syzygy family for g and h.*

Proof. According to Definition 2 a syzygy family for g and h is only required to contain *a* polynomial $a_tg - b_th$ for each $t \in T_{g,h}$, thus if we can replace the polynomial $a_tg - b_th$ with a simpler one: $a'_tg - b'_th$ still having $\mathrm{lt}(a'_tg) = \mathrm{lt}(b'_th) = c_tt$ for some $c_t \in k$, then we still have a syzygy family for g and h. In view of Corollary 4.6 from [2] we need only prove that for each power product t appearing as the leading power product of a polynomial $\mathrm{lc}(H^{v^r}) \cdot G^{v^l}$, where $v \in \mathcal{V}''$, there exist $a_t, b_t \in A$, $c_t \in k\backslash\{0\}$ such that $\mathrm{lt}(a_tg) = \mathrm{lt}(b_th) = c_tt$ and $a_tg - b_th = 0$. Let $v = u + w$ where $u \in \mathcal{V}_1$ and $w \in \mathcal{V}_2$ and let $t = \mathrm{lp}(G^{v^l}) = \mathrm{lp}(H^{v^r})$. Since u and w are solutions of (2) we know that:

$$\begin{aligned} \mathrm{lp}(G^{u^l}) &= \mathrm{lp}(H^{u^r}), \\ \mathrm{lp}(G^{w^l}) &= \mathrm{lp}(H^{w^r}). \end{aligned} \tag{3}$$

Since $u \in \mathcal{V}_1$ and $w \in \mathcal{V}_2$ their left and right halves have the form $u^l = (1, u_2, \ldots, u_{m+1})$ and $w^r = (w_{m+2}, \ldots, w_{2m+1}, 1)$, thus if we let $u' = (u_2, \ldots, u_{m+1})$ and $w' = (w_{m+2}, \ldots, w_{2m+1})$ we get:

$$\begin{aligned} G^{u^l} &= gF^{u'}, \\ H^{w^r} &= F^{w'}h. \end{aligned} \tag{4}$$

Let $a_t = F^{u'}F^{w'}h$ and $b_t = F^{u'}F^{w'}g$. Then $a_t, b_t \in A$ and:

$$\mathrm{lt}(a_tg) = \mathrm{lt}(b_th) = \mathrm{lt}(F^{u'}F^{w'}gh) = \mathrm{lt}(gF^{u'})\mathrm{lt}(F^{w'}h) = \mathrm{lt}(G^{u^l})\mathrm{lt}(H^{w^r})$$

where the last equality follows from (4). Since $\mathrm{lp}(H^{w^r}) = \mathrm{lp}(G^{w^l})$ according to (3), we can deduce that $\mathrm{lt}(H^{w^r}) = c_t\mathrm{lt}(G^{w^l})$ for some nonzero constant $c_t \in k$. Thus

$$\mathrm{lt}(G^{u^l})\mathrm{lt}(H^{w^r}) = c_t\mathrm{lt}(G^{u^l})\mathrm{lt}(G^{w^l}) = c_t\mathrm{lt}(G^{u^l+w^l}) = c_t\mathrm{lt}(G^{v^l}) = c'_tt$$

where $c'_t \in k \setminus \{0\}$, the next last equality is due to $v = u + w$ and the last equality follows from our definition $t = \mathrm{lp}(G^{v^l})$. Hence $a_tg - b_th$ is an element of the syzygy family corresponding to t. Finally we note that

$$a_tg - b_th = F^{u'}F^{w'}hg - F^{u'}F^{w'}gh = 0.$$

□

The practical use of the syzygy family is to check if a given set is a SAGBI-Gröbner basis, much like S-polynomials are used to check if a set is a Gröbner basis. More precisely a set $G \subseteq A$ is a SAGBI-Gröbner basis if and only if all polynomials in all syzygy families of pairs in G SI-reduce to zero via G, cf. Theorem 5.1 and Algorithm 3 in [2]. A zero SI-reduced remainder indicates that no violation of the SAGBI-Gröbner condition is found for this particular syzygy-polynomial, thus we can remove the extra zero indicated in Corollary 1 from the syzygy family without making the syzygy family less useful. The refinement of Algorithm 2 in [2] becomes:

Algorithm 1

```
*Input:* $g, h \in A$, *a finite SAGBI basis* $F$ *for* $A$
*Output: A syzygy family* $\mathrm{SyzFam}(g,h)$ *for* $g$ *and* $h$
*Initialisation:* $\mathrm{SyzFam}(g,h) := \emptyset$, $\mathcal{PV} := \emptyset$
*Compute a generating set* $\mathcal{V}$ *for the solutions of system (2).*
$\mathcal{PV} := \{v \in \mathcal{V} : c_0 = d_0 = 1\}$
*For Each* $v \in \mathcal{PV}$:
    $s_v := \mathrm{lc}(H^{v^r}) \cdot G^{v^l} - \mathrm{lc}(G^{v^l}) \cdot H^{v^r}$
$\mathrm{SyzFam}(g,h) := \bigcup_{v \in \mathcal{PV}} \{s_v\}$
```

An implementation of this algorithm is included in the author's Maple package for SAGBI and SAGBI-Gröbner computations, see [6]. For calculating the Hilbert bases the Maple package uses Dmitrii V. Pasechnik's implementation of the algorithm described in [3].

As an application of Algorithm 1 we consider example 4.7 and 5.2 in [2].

Example 1 *Let* $A = \mathbb{Q}[x^2, xy] \subseteq \mathbb{Q}[x,y]$ *and use the degree lexicographical order with* $x > y$. *The set* $F = \{x^2, xy\}$ *is a SAGBI basis for* A. *Let* $g = x^3y + x^2$ *and* $h = x^4 + x^2y^2$ *in* A. *A Hilbert basis for the set of solutions of the equation (2) is:*

$$v^{(1)} = (0,0,1,0,1,0), \quad v^{(2)} = (0,1,0,1,0,0), \quad v^{(3)} = (0,2,0,0,0,1),$$
$$v^{(4)} = (1,0,0,1,1,0), \quad v^{(5)} = (1,1,0,0,1,1), \quad v^{(6)} = (2,0,0,0,2,1).$$

Thus $\mathcal{PV} = \{v^{(5)}\}$*, so by Algorithm 1 a syzygy family for* (g, h) *is* $\{G^{(1,1,0)} - H^{(0,1,1)}\} = \{-x^3y^3 + x^4\}$.

In the original version of this example (example 4.7 in [2]) the syzygy family was $\{-x^5y^3 + x^6, -x^3y^3 + x^4\}$ *instead. It should however be noted (as proved in example 5.2, [2]) that the extra syzygy polynomial* $-x^5y^3 + x^6$ *SI-reduces to zero over* $\{g, h\}$*. Thus this extra polynomial does not affect the final result of the SAGBI-Gröbner basis computations. That the extra syzygy polynomial does not effect the further computations is a consequence of Theorem 1.*

References

[1] Evelyne Contejean and Hervé Devie. An efficient incremental algorithm for solving systems of linear Diophantine equations. *Inform. and Comput.*, 113(1):143–172, 1994.

[2] J. Lyn Miller. Effective algorithms for intrinsically computing SAGBI-Gröbner bases in a polynomial ring over a field. In *Gröbner bases and applications (Linz, 1998)*, volume 251 of *London Math. Soc. Lecture Note Ser.*, pages 421–433. Cambridge Univ. Press, Cambridge, 1998.

[3] Dmitrii V. Pasechnik. On computing Hilbert bases via the Elliot-MacMahon algorithm. *Theoret. Comput. Sci.*, 263(1-2):37–46, 2001. Combinatorics and computer science (Palaiseau, 1997).

[4] Lorenzo Robbiano. On the theory of graded structures. *J. Symbolic Comput.*, 2(2):139–170, 1986.

[5] Moss Sweedler. Ideal bases and valuation rings. Manuscript, 1988.

[6] Hans Öfverbeck. *HilbertSagbiSg, Maple packages for Hilbert, SAGBI and SAGBI-Gröbner basis calculations.*, 2005. `http://www.maths.lth.se/matematiklu/personal/hans/maple`.

Hans Öfverbeck,

Received December 9, 2005

Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund
Sweden
E–mail: *hans@maths.lth.se*

Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

A Deterministic and Polynomial Modified Perceptron Algorithm

Olof Barr

Abstract

We construct a modified perceptron algorithm that is deterministic, polynomial and also as fast as previous known algorithms. The algorithm runs in time $O(mn^3 \log n \log(1/\rho))$, where m is the number of examples, n the number of dimensions and ρ is approximately the size of the margin. We also construct a non-deterministic modified perceptron algorithm running in time $O(mn^2 \log n \log(1/\rho))$.

1 A Deterministic and Polynomial Modified Perceptron Algorithm

1.1 Historical and Technological Exposition

The Perceptron Algorithm was introduced by Rosenblatt in [12] and has been well-studied by mathematicians and computer scientists since then. For convenience, we will in this paper discuss the version of the algorithm that, given a set of points (constraints) $A = \cup_i a_i$ from $\mathbb{R}^n$ finds, if any, a normal z to a hyperplane through origo such that $z \cdot a_i > 0$ for every i. (Note that we do not have any zero rows in the matrix A). This is so to say, a hyperplane through origo such that all points are on the very same side of the hyperplane.

The original algorithm can easily be described as follows:

Algorithm 1.1 *The Perceptron Algorithm*

Input: A set of points (constraints) $A = \cup_i a_i$ *from* $\mathbb{R}^n$.

Output: A normal z *to a hyperplane such that* $z \cdot a_i > 0$ *for every* i*, if there is such a solution.*

1. *Let z=0.*

2. *If there is a point $a_i \in A$ such that $z \cdot a_i \leq 0$, then $z \leftarrow z + a_i$*

3. *Repeat step 2 until no such point is found and output z.*

Obviously this algorithm will never halt if there is no solution to the problem. On the other hand Novikoff proved that if there is a solution to the problem, the algorithm will halt in a finite number of steps, even if the number of constraints is infinite.

Theorem 1.1 *The number of mistakes made by the on-line perceptron algorithm, on a set A that has a solution to the problem, is at most $(2R/\gamma)^2$, where $R = \max \|a_i\|$ and γ is the size of the margin.*

Now, since the margin can be any positive number close to zero, this upper bound of the performance does not say very much. And even though there are many results showing that the algorithm runs much faster in the "usual" case, there are constructions of constraint sets such that the behaviour of the algorithm is exponential in terms of the number of constraints [2].

1.1.1 Linear Programs

The problem solved by this Perceptron Algorithm is the homogenized form of a feasible standard form of a linear program: Find x such that $Ax \geq 0$ and $x \neq 0$. Here, A is the matrix with a_i as row i and $x = z^T$, where z is the normal of the hyperplane described above. This problem is of great importance, since it solves $\max c^T x$, $Ax \leq b$, $x \geq 0$ by iterative solving the homogenized version and performing a binary search.

Due to the importance of the problem, many different methods have been evolved to find solutions to it. To mention here, those are the simplex algorithm, the interior point method, ellipsoid methods, the perceptron algorithm and the modified perceptron algorithm.

1.1.2 The Modified Perceptron Algorithm

The first polynomial algorithm using the perceptron algorithm was created by Dunagan and Vempala in [7]. Even though the algorithm they constructed was not as fast as other algorithms mentioned above, it must be said it was a break through for the Perceptron Algorithm. After this result, the algorithm could not be counted out, it could be competitive.

Algorithm 1.2 *The Modified Perceptron Algorithm (Dunagan & Vempala)*

Input: An $m \times n$ matrix A.
Output: A point x such that $Ax \geq 0$ and $x \neq 0$.

```
1. Let $B = I$, $\sigma = 1/(32n)$.
2. (Perceptron)
    (a) Let $x$ be the origin in $\mathbb{R}^n$.
    (b) Repeat at most $16n^2$ times: If there exists a row $a$ such that
        $a \cdot x \leq 0$, set $x = x + \bar{a}$.
    Here $\bar{a}$ denotes the normalized vector $a/\|a\|$.
3. If $Ax \geq 0$, then output $Bx$ as a feasible solution and stop.
4. (Perceptron Improvement)
    (a) Let $x$ be a random unit vector in $\mathbb{R}^n$.
    (b) Repeat at most $(\ln n)/\sigma^2$ times: If there exists a row $a$ such
        that $\bar{a} \cdot \bar{x} < -\sigma$, set $x \leftarrow x - (\bar{a} \cdot x)\bar{a}$. If $x = 0$, go back to step
        (a). (This is to assure us of not having the vector $x$ set to zero)
    (c) If there still exists a row $a$ such that $\bar{a} \cdot \bar{x} < -\sigma$, restart at
        step (a). (This takes care of the situation of a bad choice of the
        randomly chosen vector $x$)
5. If $Ax \geq 0$, then output $Bx$ as a feasible solution and stop.
```

6. *(Rescaling)*

   Set $A \leftarrow A(I + \bar{x}\bar{x}^T)$ *and* $B \leftarrow B(I + \bar{x}\bar{x}^T)$.

7. *Go back to step 2.*

It is not evident that this algorithm will terminate. But Dunagan and Vempala prove that the margin of the Linear Program will increase in mean, when many rescalings are made inside the algorithm. This will in turn make the margin so large such that the Perceptron part of the algorithm will return a solution to the problem of the Linear Program.

Beside the size of the running time ($O(mn^4 \log n \log(1/\rho)$, where ρ is approximately the size of margin γ), the algorithm they presented was not deterministic. This meant that the result was given in the mentioned time with very high probability. This is bad, since exceeding the specified time will not always imply that no solution exists.

Dunagan and Vempala put an open question whether or not there was a deterministic version of their algorithm.

1.2 A Deterministic and Polynomial Modified Perceptron Algorithm

In this paper, we answer the above stated question in the affirmative. Also, the constructed algorithm presented has the a running time a factor $O(n)$faster than the one by Dunagan and Vempala. As a consequence of the construction, we also get a non-deterministic algorithm running a factor $O(n^2)$ faster than the algorithm constructed by Dunagan and Vempala.

1.2.1 How to Make it Deterministic

First of all we can conclude that it is due to the random choice of a unit vector inside the algorithm that makes the algorithm of Dunagan and Vempala a non-deterministic one. The question to put is if there is a way of choosing appropriate vectors so that we can keep control of the number of iterations being made inside the algorithm.

What the algorithm wants to choose is a unit vector that has an inner product of at least $1/\sqrt{n}$ with a feasible solution z to the posed problem. But since we do not know a solution to the problem, we can ask us if we can have a set V of vectors, where at least one vector $v \in V$ has the mentioned property. The answer to this is positive. $V = \cup_{i=1}^{n}\{e_i, -e_i\}$, where $\cup_{i=1}^{n}\{e_i\}$ constitutes an ON-basis for $\mathbb{R}^n$, will do according to the following proposition.

Proposition 1.1 *Let $V = \cup_{i=1}^{n}\{e_i, -e_i\}$, where $\cup_{i=1}^{n}\{e_i\}$ constitutes an ON-basis for $\mathbb{R}^n$. Now, for every unit vector $w \in \mathbb{R}^n$,*

$$\max_{v \in V} w \cdot v \geq \frac{1}{\sqrt{n}}.$$

Proof: First assume that e_i is the vector with a 1 in the ith coordinate and zero elsewhere. Now let $w = (w_1, \ldots, w_n)$. At least one w_i must have an absolute value of at least $1/\sqrt{n}$, otherwise $\|w\| < 1$. This yields that there exists at least one vector $v \in V$ such that $w \cdot v \geq 1/\sqrt{n}$ as stated above. To generalize this statement for any ON-basis, we only have to consider the rotation symmetry of $\mathbb{R}^n$. $\square$

To make the algorithm deterministic, we will now run the algorithm in $2n$ parallel tracks. And instead of using the origin in step 2(a) and a random unit vector in step 4(a), we will use vectors from our set V and update them for each iteration in the algorithm. Since one of parallel tracks will come closer and closer to a solution for each round, this specific track will terminate in the time mentioned above. But, we are running $2n$ tracks, and the total running time will be a factor $2n$ greater than before.

In practice, this is an advantage since the algorithm will tell us how to make use of paralell processors in a practical situation, making the algorithm fast when implemented.

1.2.2 How to Speed Up the Algorithm

In order to speed up the algorithm, a deep analysis of all estimates done by Dunagan and Venpala has been done. As a result of this σ can be enlarged to $1/(32\sqrt{n})$ and the $16n^2$ in step 2(b) can be reduced to $4n$. These alterations will speed up the process with a factor of order $O(n)$. New estimates in the margin growth causes another factor of order $O(n)$.

1.3 The Deterministic Modified Perceptron Algorithm

Now, we are ready to go into details with the topic of this paper.

Below we can study the general structure of the algorithm, breaking it up into smaller parts that will be presented further on. Important is though that we are running the algorithm in $2n$ parallel tracks. This does not imply that we have to run the algorithm on parallel processors, only that we do each step for every single track before going to the next step.

Algorithm 1.3 *The Modified Perceptron Algorithm*
Input: An $m \times n$ matrix A.
Output: A vector x such that $Ax \geq 0$ and $x \neq 0$ or "No solution exists".

1. *Initials*

   Choose $R = n$ (the dimension of the Linear Program) and put $\sigma = \frac{1}{32\sqrt{n}}$.

   Let $B = I$ and $V = \cup_{i=1}^{n}\{e_i, -e_i\}$, where $\cup_{i=1}^{n}\{e_i\}$ constitutes an ON-basis for $\mathbb{R}^n$.

2. *Wiggle Phase*

   Let $U = \emptyset$. For each vector $v \in V$, run the Wiggle Algorithm, collecting all returned corresponding vectors u in U.

   $V \leftarrow U$.

3. *Check for no solution*

   If $V = \emptyset$, then output "No solution exists" and stop.

4. *Rescaling Phase Normalize every vector in V. That is, for every $v \in V$, let $v \leftarrow \frac{v}{\|v\|}$*

   For each vector v in V (at most $2n$), together with its corresponding matrices A and B, run the Rescaling Algorithm.

5. *Perceptron Phase*

   For each vector $v \in V$, run the Perceptron Algorithm for at most R rounds with v as the initial vector.

6. *If no feasible solution was obtained in the last iteration of the algorithm, scale every vector $v \in V$ such that $\|v\| = 1$ and go back to step 2.*

Note that there are more things to show than only to describe the smaller algorithms used inside the larger structure. We have to show that they work, to calculate the complexity and to prove that the main algorithm always will return a correct answer in the time mentioned above.

First we describe the Wiggle Algorithm:

Algorithm 1.4 *Wiggle Algorithm*

Input: An $m \times n$ matrix A, a vector $v \in \mathbb{R}^n$ and a set of vectors U.

Output: One of the following three: A solution x to $Ax \geq 0$, an updated vector v that will be put in U or the empty set $\emptyset$ (also to be put in U).

1. *If $\frac{a \cdot v}{\|a\|\|v\|} < -\sigma$ for some row $a \in A$,*

   then $v \leftarrow v - \left(\frac{a}{\|a\|} \cdot v\right) \frac{a}{\|a\|}$.

2. *Repeat step 1 at most $(\log n)/\sigma^2$ times.*

3. *If $\frac{a \cdot v}{\|a\|\|v\|} \geq -\sigma$ for every row $a \in A$, then $U \leftarrow U \cup v$*

4. *If $Av \geq 0$, then output Bv as a feasible solution and stop.*

This algorithm is shown in [5], to output a vector v in at most $(\log n)/\sigma^2$ steps, if the input vector v satisfies $v \cdot z \geq 1/\sqrt{n}$, where z is a unit vector that solves $Ax \geq 0$. Thus, we have to insure us that at least one of our starting vectors in V does have this property. But this was shown in the earlier proposition presented above.

Also we must calculate the complexity of running through the Wiggle Algorithm once:

Proposition 1.2 *The number of iterations inside the Wiggle Algorithm is of order $O(mn^2 \log n)$.*

Proof: The inner loop of the algorithm requires at most one matrix-vector multiplication, time $O(mn)$, and a constant number of vector manipulations, time $O(n)$. This is repeated at most $(\log n)/\sigma^2 = 32^2 n \log n$ times. So, the overall time bound is $O(mn^2 \log n)$. □

The following rescaling procedure is a simple matrix-matrix multiplication, being of order $O(n^2)$.

Algorithm 1.5 *Rescaling Phase*

```
Input: a vector v and its
corresponding matrices A and B.
Output: rescaled matrices A and B.

1. $A \leftarrow A(I + \frac{v}{\|v\|}\frac{v}{\|v\|}^T)$
2. $B \leftarrow B(I + \frac{v}{\|v\|}\frac{v}{\|v\|}^T)$
```

The important thing to prove for this part, is that for at least one of our $2n$ parallel processes, the matrices will be stretched in a direction such that the margin increases in the new problem $Ax \geq 0$. The proof of this follows substantially the proof in [7], but using some other indata. The reason for letting $\rho \leq 1/(2\sqrt{n})$ in the following theorem is that if ρ would be larger, the algorithm will halt later on in the Perceptron phase.

Theorem 1.2 *Suppose $\rho \leq 1/(2\sqrt{n})$ and $\sigma = 1/(32\sqrt{n})$. Let A' be obtained from A by one iteration of the algorithm (where the problem is not solved). Let ρ' and ρ be the margins of the problems $A'x \geq 0$ and $Ax \geq 0$ respectively. Also assume that we are studying one of those processes running parallel, where $v \cdot z \geq 1/\sqrt{n}$ and z is a feasible solution, of length one, to $Ax \geq 0$. Then $\rho' \geq (1+\frac{1}{6})\rho$.*

Proof: Let a_i, $i = 1, \dots, m$ be the rows of A at the beginning of some iteration, for one of the parallel processes having $v \cdot z \geq 1/\sqrt{n}$. (Below we drop the index i, and usually denote a row a_i only with a). Let z be the unit vector satisfying $\rho = \min_i \frac{a}{\|a\|} \cdot z$, and let $\sigma_i = \frac{a}{\|a\|} \cdot v$. After the wiggle phase, we get a vector v such that $\frac{a}{\|a\|} \cdot v = \sigma_i \geq -\sigma$ for every i.

As described in the algorithm, let A' be the matrix obtained after the rescaling step, i.e. $a'_i = a_i + k(a_i \cdot v)v$. Finally define $z' = z + \alpha(z \cdot v)v$, where

$$2\alpha + 1 = \rho\sqrt{n}$$

or to put it another way

$$\alpha = (\rho\sqrt{n} - 1)/2.$$

Even though z' might not be an optimal choice, it is enough to consider this one element to lower bound ρ'. We have $\rho' \geq \min_j \frac{a'}{\|a'\|} \cdot \frac{z'}{\|z'\|}$.

We will first prove that $\frac{a'}{\|a'\|} \cdot z'$ cannot be too small.

$$\frac{a'}{\|a'\|} \cdot z' = \frac{\frac{a}{\|a\|} + (\frac{a}{\|a\|} \cdot v)v}{\|\frac{a}{\|a\|} + (\frac{a}{\|a\|} \cdot v)v\|} \cdot z' = \frac{[\frac{a}{\|a\|} + (\frac{a}{\|a\|} \cdot v)v][z + \alpha(z \cdot v)v]}{\sqrt{1 + 3(\frac{a}{\|a\|} \cdot v)^2}},$$

since the vector v is normalized before the rescaling is done. Now, in the case of a positive σ_i,

$$= \frac{\rho + \sigma_i(z \cdot v)(1 + 2\alpha)}{\sqrt{1 + 3\sigma_i^2}} \geq \rho \frac{1 + \sigma_i(z \cdot v)\sqrt{n}}{\sqrt{1 + 3\sigma_i^2}} \geq \rho \frac{1 - \sigma}{\sqrt{1 + 3\sigma^2}},$$

where the last inequality follows from that $(z \cdot v) \geq 1/\sqrt{n}$ and $\sigma_i \in [-\sigma, 1]$, with some hard work using the method of Lagrange. On the other hand, if σ_i is negative we get that the expression is at least

$$\frac{31}{32\sqrt{1+3\sigma^2}}.$$

Now we are about to bound $\|z'\|$ from above, and for convenience we study the square of it, $\|z'\|^2$. We know that

$$\|z'\|^2 = \|z + \alpha(z \cdot v)v\|^2 = 1 + (\alpha^2 + 2\alpha)(v \cdot z)^2$$

Inserting our known $\alpha = (\rho\sqrt{n} - 1)/2$ we get that

$$\|z'\|^2 = 1 + ((\alpha+1)^2 - 1)(v \cdot z)^2 \leq 1 - \frac{7}{16} = \frac{9}{16}$$

since $\alpha + 1 \leq 3/4$.

Using the identity

$$\frac{1}{\sqrt{1+\beta}} \geq 1 - \frac{\beta}{2}$$

for $\beta \in (-1, 1)$, we find that the total estimate for the new margin is

$$\rho' \geq \rho\left(1 - \frac{1}{2^5\sqrt{n}}\right)\left(1 - \frac{3}{2^{11}n}\right)\left(\frac{4}{3}\right) \geq \rho\left(\frac{7}{6}\right)$$

when σ_i is positive. Otherwise, when σ_i is negative we get that

$$\rho' \geq \rho\left(\frac{31}{32}\right)\left(1 - \frac{3}{2^{11}n}\right)\left(\frac{4}{3}\right) \geq \rho\left(\frac{7}{6}\right).$$

This estimate will be enough to fulfill the demand we have on the margin to increase with a certain proportion, such that we also guarantee a convergence in the case of the non-deterministic algorithm. For further details, see [3] and [7]. □

Now we describe the classical Perceptron Algorithm, but reduced to at most n steps, being an important component of the modified algorithm.

Algorithm 1.6 *Perceptron Algorithm*

Input: A starting vector v from V.

Output: A feasible solution Bv or a new updated vector v.

```
1. If there is a row $a$ in $A$ such that $v \cdot a \leq 0$, then $v \leftarrow v + a/\|a\|$.
2. If $Av \geq 0$, then output $Bv$ as a feasible solution and stop.
3. Repeat the two steps above at most $R = 4n$ times.
```

The behaviour of this algorithm is well-studied, and we know from, for example, [6] that it produces a feasible solution if the problem has a margin ρ of size at least $1/\sqrt{R} = 1/(2\sqrt{n})$. Also we can conclude that:

Proposition 1.3 *The number of iterations inside the Perceptron Algorithm is of order $O(mn^2)$.*

Proof: The algorithm will perform at most one matrix-vector multiplication in its inner loop (made at time $O(mn)$) and a constant number of vector manipulations (in time $O(n)$). This is done at most $2n$ times. So we get $O(mn^2)$. $\square$

Lemma 1.1 *The number of times we repeat step 2-5 in Algorithm 1.1 is at most of order $O(n \log(1/\rho))$.*

Proof: As we have seen above, at least one of our $2n$ parallel processes will start with a vector v satisfying $v \cdot z \geq 1/\sqrt{n}$ where z is a feasible unit vector. So, after the wiggling phase, the resulting vector will be a proper direction for rescaling, this enlarges the radius ρ with a factor of size at least $(1 + 1/6)$. But since the perceptron stage will terminate, yielding a feasible solution if $\rho \geq 1/(2\sqrt{n})$, we know that our algorithm will terminate after k proper rescalings when

$$\rho \left(1 + \frac{1}{6}\right)^k \geq \frac{1}{2\sqrt{n}}.$$

This yields that $k = O(\log(1/\rho))$ $\square$

Summing up the information we have got, we get the complexity of the algorithm in total.

Theorem 1.3 *The modified Perceptron Algorithm returns an answer in time $O(mn^3 \log n \log(1/\rho))$.*

Proof: The algorithm runs in $2n$ parallel processes. The wiggle algorithm runs for at most $2^{10} n \log n$ times, each round taking time $O(mn)$. The rescale process takes $O(n^2)$ and the Perceptron algorithm runs for at most n times, each round taking time $O(mn)$. All these three stages are repeated at most $O(\log(1/\rho))$ times. So we get that the total time is

$$O\left(2nO\left(\log\left(\frac{1}{\rho}\right)\right)(2^{10}mn^2\log n + O(n^2) + mn^2)\right) =$$
$$= O\left(mn^3 \log n \log\left(\frac{1}{\rho}\right)\right)$$

$\square$

1.3.1 A Fast Non-Deterministic Polynomial Modified Perceptron

The results in the previous section can be used to strengthen the results made by Dunagan and Vempala in [7]. In the non-deterministic case we are not longer in need for our $2n$ parallel processes anymore. Now, taking away the condition about having a deterministic process, we can speed it up a factor $2n$.

In general, one could follow the proof made by Dunagan and Vempala in [7] to prove the behaviour of the non-deterministic algorithm. The only changes made are the size of R, σ and k inside the algorithm.

But we do want to point out a statement made in the article: a statement that says that the probability of two random unit vectors have inner product at least $1/\sqrt{n}$ is at least $1/8$ can be shown by a standard computation. The statement is true, but we have not found a standard argument proving this statement. A detailed analysis can be found in [3] showing that the probability is at least $(1 - \mathrm{erf}(1/\sqrt{2})/2 \geq 1/8$ where $\mathrm{erf}(x)$ is the errorfunction

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{dt}.$$

Anyhow, we get the following:

Corollary 1.1 *There is a polynomial non-deterministic modified perceptron algorithm that terminates in time $O(mn^2 \log n \log(1/\rho))$.*

References

[1] N. Alon and A. Naor, Approximating the Cut-Norm via Grothendieck's Inequality, *submitted.* Available at http://www.math.tau.ac.il/˜nogaa/PDFS/publications.html

[2] M. Anthony and J. Shawe-Taylor, Using the Perceptron Algorithm to Find Consistent Hypotheses *Combinatorics, Probability and Computing* (1993) 2:pp. 385-387.

[3] O. Barr and O. Wigelius, *New Estimates Correcting an Earlier Proof of the Perceptron Algorithm to be Polynomial*, ISSN 1403-9338, LUTFMA-5041-2004.

[4] A. Blum and J. Dunagan, Smoothed Analysis of the Perceptron Algorithm for Linear Programming, in *SODA '02*, 2002; 905–914.

[5] A. Blum, A. Frieze, R. Kannan and S. Vempala, A Polynomial-time Algorithm for Learning Noisy Linear Threshold Functions *Algorithmica,* **22**(1/2):35–52, 1997.

[6] N. Cristianini and J. Shawe-Taylor, *Support Vector Machines,* Cambridge, 2000.

[7] J. Dunagan and S. Vempala, *A Polynomial-time Rescaling Algorithm for Solving Linear Programs,* Microsoft Research, Redmond.

[8] M. Grötschel, L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization,* Springer Verlag, Berlin Heidelberg, 1988.

[9] D. G. Luenberger, *Linear and Nonlinear Programming,* Addison-Wesley, Reading, Massachusetts, 1984.

[10] M. E. Muller, A Note on a Method for Generating Points Uniformly on N-Dimensional Spheres *Comm. Assoc. Comput. Mach.***2**, 19-20, 1959.

[11] Y. Nesterov and A. Nemirovskii, *Interior-Point Polynomial Algorithms in Convex Programming* Studies in Applied Mathematics, Vol. 13, Philadelphia, 1994.

[12] F. Rosenblatt, *Principles of Neurodynamics.* Spartan Books, 1962.

[13] N. Z. Shor, *Minimization Methods for Non-Differentiable Functions,* Springer-Verlag, Berlin, Heidelberg, 1985.

[14] D. Spielman and S. Teng, Smoothed Analysis of Termination of Linear Programming Algorithms, in *Mathematical Programming, Series B,* Vol. **97**, 2003

[15] D. Spielman and S. Teng, Smoothed Analysis: Why The Simplex Algorithm Usually Takes Polynomial Time, in *Proc. of the 33rd ACM Symposium on the Theory of Computing,*, 296–305, 2001.

Olof Barr,

Centre for Mathematical Sciences
Lund University
Sweden
E–mail: *barr@maths.lth.se*

Received December 9, 2005

Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

DDP-Based Ciphers: Differential Analysis of SPECTR-H64

A.V. Bodrov, A.A. Moldovyan, P.A. Moldovyanu

Abstract

Use of data-dependent (DD) permutations (DDP) appears to be very efficient while designing fast ciphers suitable for cheap hardware implementation, few papers devoted to security analysis of the DDP-based cryptosystems have been published though. This paper presents results of differential cryptanalysis (DCA) of the twelve-round cipher SPECTR-H64 which is one of the first examples of the fast block cryptosystems using DDP as cryptographic primitive. It has been shown that structure of SPECTR-H64 suits well for consideration of the differential characteristics. Experiments have confirmed the theoretic estimations. Performed investigation has shown that SPECTR-H64 is secure against DCA, some elements of this cipher can be improved though. In order to make the hardware implementation faster and cheaper a modified version of this cipher with eight rounds is proposed.

Key words: Fast ciphers, hardware encryption, controlled operations, data-dependent permutations, differential analysis

1 Introduction

Data encryption is widely used to solve different problems of the information security. This defines importance of the design of the ciphers suitable for cheap hardware implementation. Recently the controlled operations (CO) has been proposed as an attractive cryptographic primitive suitable for the design of such ciphers [1, 2]. One of early applications of the CO relates to [1], where the key-dependent substitution boxes are used while designing a block cipher. A class of CO suitable for cryptographic applications is proposed in [2]. Controlled permutations (CP) attract much attention of cryptographers, since they can be implemented in fast and cheap hardware using permutation networks (PN) developed and investigated previously [3-5]. The PN are well suited for cryptographic applications, since they allow one to specify and perform permutations at the same time. A variant of the symmetric cryptosystem based on PN and Boolean functions is presented in [6]. Another cryptographic application of PN is presented by the cipher ICE [7] in which a very simple PN is used to specify a key-dependent permutation. In such applications of CP the permutation on data bit strings is a linear operation. Such use of CP has been shown [8] to be not very effective against differential cryptanalysis (DCA), very large number of different bit permutations can be specified though.

Efficiency of CO as cryptographic primitive crucially increases while using CO as data-dependent (DD) operations (DDO). A particular kind of CP represented by DD rotations (DDR) are successfully used in cryptosystems RC5 [9], RC6 [10], and MARS [11]. In spite of the fact that DDR contain few different realizable modifications they thwart well DCA and linear cryptanalysis (LCA). Use of CP in the form of DD permutations (DDP) appears to be very suitable to design fast ciphers oriented to cheap hardware implementation [12]. The iterative 64-bit block cipher SPECTR-H64 represents an example of DDP-based ciphers [13]. It is interesting that the round transformation of this cipher is not involution, the same algorithm performs encryption and decryption though. Since the DDP-based design is oriented to drastic decrease of the hardware implementation cost, the security estimation of the DDP-based ciphers is of the great importance. If the detailed cryptanalysis show the DDP-based ciphers are secure, then we will have actually a very efficient approach to embed fast encryption algorithm in cheap hardware.

The present paper is one of the first ones devoted to the security analysis of the DDP-based ciphers.

The paper is organized in the following way: In the second section we describe briefly the algorithm SPECTR-H64 paying attention to the design of the operational boxes performing DDP. Section 3 considers differential characteristics of the primitives used in SPECTR-H64 and presents security analysis of this cipher and experimental results confirming our estimations. In section 4 we propose some improvements allowing one to reduce the number of rounds from 12 to 8 that results in the performance increase and hardware cost decrease.

Notation. Let $\{0,1\}^n$ be the set of all binary vectors $U = (u_1, ..., u_n)$, where $\forall i \in \{1, ..., n\}$ $u_i \in \{0,1\}$. Let us denote $U_{\text{lo}} = (x_1, ..., x_{n/2})$ and $X_{\text{hi}} = (x_{n/2+1}, ..., x_n)$, i.e. $X = (X_{\text{lo}}, X_{\text{hi}})$ or $X = (X_{\text{lo}}|X_{\text{hi}})$, where "$|$" denotes the concatenation operation. Let $e \in \{0,1\}$ denote encryption ($e = 0$) or decryption ($e = 1$).

Let $Y = X^{\ggg k}$ denote cyclic rotation of the word X by k bits, where $Y = (y_1, ..., y_n)$ is the output vector and $\forall i \in \{1, ..., n-k\}$ we have $y_i = x_{i+k}$ and $\forall i \in \{n-k+1, ..., n\}$ we have $y_i = x_{i+k-n}$.

Let XY denote bit-wise AND operation of the two vectors X and Y: $X, Y \in \{0,1\}^n$. Let $\oplus$ denote the XOR operation.

2 Design of the block cipher SPECTR-H64

2.1 General encryption scheme

SPECTR-H64 is a new 12-round block cipher with 64-bit input. The general encryption scheme (Fig. 1) is defined by the following formulas: $C = \mathbf{Encr}(M, K)$ and $M = \mathbf{Decr}(C, K)$, where M is the plaintext, C is the ciphertext ($M, C \in \{0,1\}^{64}$), K is the secret key ($K \in \{0,1\}^{256}$), **Encr** is the encryption function, and **Decr** is the decryption function. In the block cipher SPECTR-H64 the encryption and decryption functions are described by formula

$$Y = \mathbf{F}(X, Q^{(e)}),$$

where $Q^{(e)} = \mathbf{H}(K, e)$ is the extended key, the last being a function of the secret key $K = (K_1, K_2, ..., K_8)$, $K_i \in \{0,1\}^{32}$ for $i = 1, 2, ..., 8$, and of the transformation mode parameter e ($e = 0$ defines encryption, $e = 1$

defines decryption). We have $X = M$, for $e = 0$ and $X = C$ for $e = 1$. Extended key is represented as follows:

$$Q^{(e)} = (Q_{\mathrm{IT}}^{(e)}, Q_1^{(e)}, ..., Q_{12}^{(e)}, Q_{\mathrm{FT}}^{(e)})$$

where $Q_{\mathrm{IT}}^{(e)}, Q_{\mathrm{FT}}^{(e)} \in \{0,1\}^{32}$ and $\forall j = 1, ..., 12 \quad Q_j^{(e)} \in \{0,1\}^{192}$. Each round key $Q_j^{(e)} = (Q_j^{(1,e)}, ..., Q_j^{(6,e)})$, where $\forall h = 1, ..., 6 \; Q_j^{(h,e)} \in \{0,1\}^{32}$, is represented as concatenation of six subkeys which are selected from the set $\{K_1, K_2, ..., K_8\}$ depending on the number of the current round and the value e. Output value Y is the ciphertext C in the encryption mode or the plaintext M in the decryption mode. The algorithm (function **F**) is designed as sequence of the following procedures: 1) *initial transformation* **IT**, 2) 12 rounds with procedure **Crypt**, and 3) *final transformation* **FT**. For detailed description of the key scheduling and **IT** and **FT** one can see [13]. In our analysis we consider the round keys to be uniformly distributed random values. This makes our security estimate to be valid in the case of the more strong key scheduling. We assume also that **IT** and **FT** do not contribute significantly to security.

Let consider the jth encryption round and denote $Q_j^{(e)} = (A, B, A', B', A'', B'')$. The round transformation of SPECTR-H64 is denoted as procedure **Crypt** shown in Fig. 2. This procedure has the form: $R = \mathbf{Crypt}(R, L, A, B, A', B', A'', B'')$, where $L, R, A, B, A', B', A'', B'' \in \{0,1\}^{32}$. The procedure **Crypt** uses the following operations: cyclic rotation ">>>" by fixed number of bits, XOR operation "$\oplus$", non-linear operation **G**, DDP operations $\mathbf{P}_{32/80}$ and $\mathbf{P}_{32/80}^{-1}$, and extension operation **E**.

2.2 Non-linear operation G

Realization of the operation $Y = \mathbf{G}(X, A, B)$ is defined in the vector form by the following expression:

$$Y = W_0 \oplus W_1 \oplus W_2 A \oplus W_2 W_5 B \oplus W_3 W_5 \oplus W_4 B,$$

where binary vectors W_j for $j = 0, 1, ..., 5$ are expressed as follows: $W_0 = X = (x_1, x_2, ..., x_{32})$, $W_1 = (1, x_1, x_2, ..., x_{31}), ...,$ $W_j =$

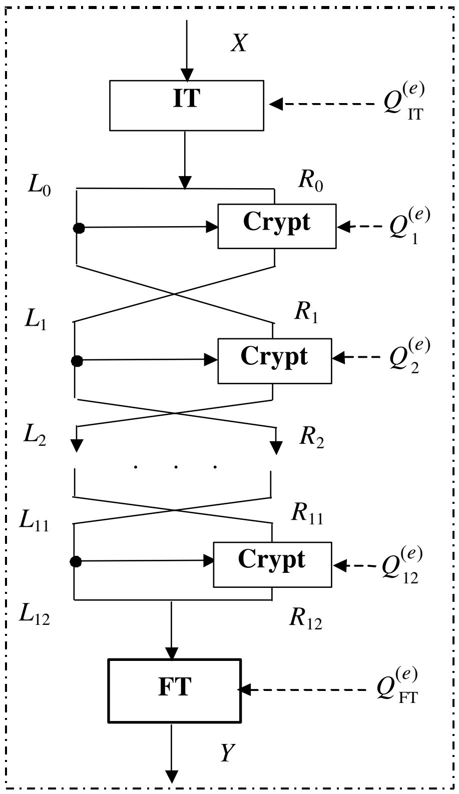


Figure 1. General structure of SPECTR-H64

$(1, ..., 1, x_1, x_2, ..., x_{32-j})$. The ith output bit of the operation $\mathbf{G}$ is the following Boolean function:

$$y_i = x_i \oplus x_{i-1} \oplus x_{i-2}a_i \oplus x_{i-2}x_{i-5}b_i \oplus x_{i-3}x_{i-5} \oplus x_{i-4}b_i,$$

where $x_{-4} = x_{-3} = x_{-2} = x_{-1} = x_0 = 1$.

2.3 CP-boxes $\mathbf{P}_{32/80}$ and $\mathbf{P}^{-1}_{32/80}$

The CP boxes $\mathbf{P}_{32/80}$ and $\mathbf{P}^{-1}_{32/80}$ are built up from switching elements $\mathbf{P}_{2/1}$ in accordance with the layered structure shown in Fig. 3. Each

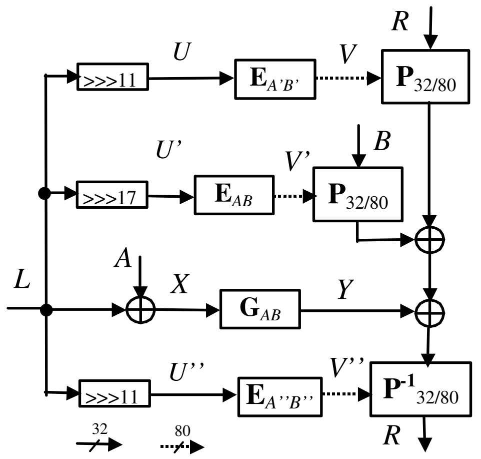


Figure 2. Structure of the procedure **Crypt**

box $\mathbf{P}_{2/1}$ is controlled by one bit v: $y_1 = x_{1+v}$ and $y_2 = x_{2-v}$, where (x_1, x_2) is input and (y_1, y_2) is output. In all figures in this paper the solid lines indicate data movement, while dotted lines indicate the controlling bits. Layered CP boxes we shall denote as $\mathbf{P}_{n/m}$, where n corresponds to the input/output size in bits and m indicates the size of the controlling input that is equal to the number of the used $\mathbf{P}_{2/1}$-boxes. Performing a CP operation can be denoted as $Y = \mathbf{P}_{n/m(V)}(X)$, where X is input vector, Y is output vector, and V is controlling vector. For fixed value V the CP box performs fixed permutation that is called CP modification. Let indexing the elementary $\mathbf{P}_{2/1}$-boxes in a CP box $\mathbf{P}_{n/m}$ from left to right and from top to bottom. The CP-box $\mathbf{P}^{-1}_{n/m}$ is called inverse of $\mathbf{P}_{n/m}$-box, if for all V the corresponding CP modifications $\mathbf{P}_V$ and $\mathbf{P}^{-1}_V$ are mutually inverse [12].

Let given a CP box $\mathbf{P}_{n/m}$. Then it is easy to construct $\mathbf{P}^{-1}_{n/m}$ by changing the direction of the bits moving. We shall enumerate the $\mathbf{P}_{2/1}$-boxes of some $\mathbf{P}^{-1}_{n/m}$-box from left to right and from bottom to top. Thus, in both $\mathbf{P}_{n/m}$ and $\mathbf{P}^{-1}_{n/m}$ boxes the controlling bit v_j controls the jth $\mathbf{P}_{2/1}$-box.

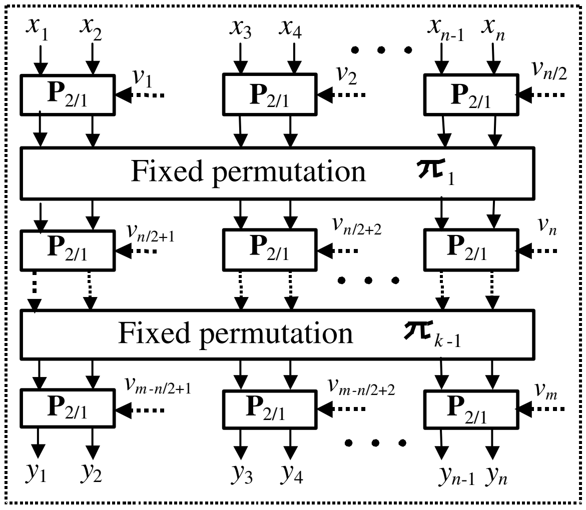

x_1 x_2 x_3 x_4 x_{n-1} x_n
$\mathbf{P}_{2/1}$
v_1 v_2 $v_{n/2}$
Fixed permutation π_1
$v_{n/2+1}$ $v_{n/2+2}$ v_n
Fixed permutation π_{k-1}
$v_{m-n/2+1}$ $v_{m-n/2+2}$ v_m
y_1 y_2 y_3 y_4 y_{n-1} y_n

Figure 3. A CP box with layered structure

The structure of the boxes $\mathbf{P}_{32/80}$ and $\mathbf{P}^{-1}_{32/80}$ is explained in Fig. 4-6. The fixed permutational involution between the third and fourth layers of the elementary boxes $\mathbf{P}_{2/1}$ in the $\mathbf{P}_{32/80}$-box is described as follows:

$$\begin{array}{c}(1)(2,9)(3,17)(4,25)(5)(6,13)(7,21)(8,29)(10)(11,18)\\(12,26)(14)(15,22)(16,30)(19)(20,27)(23)(24,31)(28)(32)\end{array}.$$

2.4 Extension box E

The extension box $\mathbf{E}$ is used to form a 80-bit controlling vector, given the 32-bit input vector. The formal representation of the extension transformation is: $V = (V_1|V_2|V_3|V_4|V_5) = \mathbf{E}(U, A', B') = \mathbf{E}_{A',B'}(U)$, where $V \in \{0,1\}^{80}$; $V_1, V_2, V_3, V_4, V_5 \in \{0,1\}^{16}$; $U, A', B' \in \{0,1\}^{32}$. Actually the vectors V_1, V_2, V_3, V_4, V_5 are determined according to the

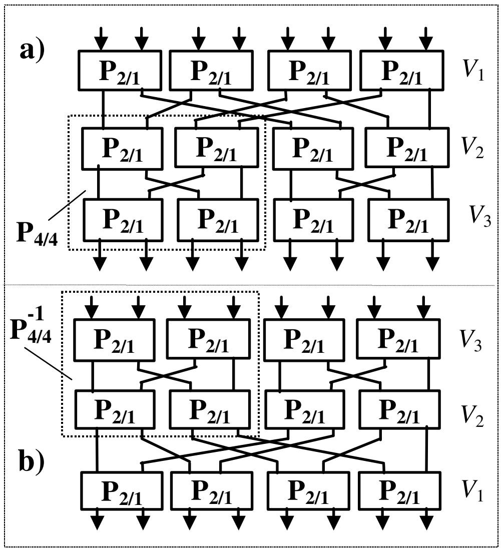


Figure 4. Structure of the boxes $\mathbf{P}_{8/12}$ (a) and $\mathbf{P}_{8/12}^{-1}$ (b)

formulas:

$$V_1 = U_{\text{hi}};\ V_2 = \pi((U \oplus A)_{\text{hi}}); \qquad V_3 = \pi'((U \oplus B')_{\text{hi}});$$

$$V_4 = \pi'((U \oplus B')_{\text{lo}}); \qquad V_5 = \pi((U \oplus A)_{\text{lo}}),$$

where fixed permutations π and π' are the following: $\pi(Z) = Z_{\text{hi}}^{\ggg 1}|Z_{\text{lo}}^{\ggg 1}$ and $\pi'(Z) = Z_{\text{hi}}^{\ggg 5}|Z_{\text{lo}}^{\ggg 5}$. Table 1 specifies which bit $u_i \in U$ controls which $\mathbf{P}_{2/1}$-box in the PN is presenting the $\mathbf{P}_{32/80}$-box. Number i is indicated in the position of the correspondent $\mathbf{P}_{2/1}$-box. In other words this distribution table shows correspondence between bits of the vector U and bits of the vector V. For example, $v_1 = u_{17}$, $v_2 = u_{18}$,..., $v_{16} = u_{32}$, $v_{17} = k'_1 \oplus u_{26}$,..., $v_{32} = k'_{16} \oplus u_{26}$, where k'_1,...,k'_{16} are fixed bits (actually they are some bits of the key). Designed distribution provides that each input bit is affected by five different bits of U for all possible values U, A', and B'.

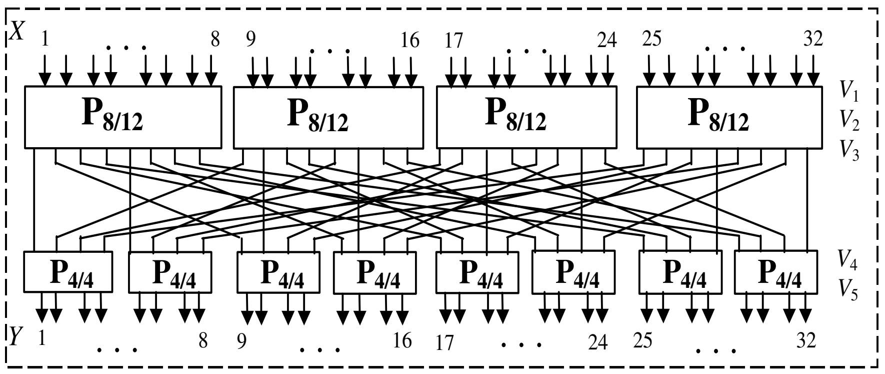


Figure 5. Structure of the box $\mathbf{P}_{32/80}$

Table 1. Distribution of bits of the vector U

V_1	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
V_2	26	27	28	29	30	31	32	25	18	19	20	21	22	23	24	17
V_3	30	31	32	25	26	27	28	29	22	23	24	17	18	19	20	21
V_4	14	15	16	9	10	11	12	13	6	7	8	1	2	3	4	5
V_5	10	11	12	13	14	15	16	9	2	3	4	5	6	7	8	1

3 Differential Analysis of SPECTR-H64

3.1 Some properties of the controlled operations

Let Δ_q^W be the difference with arbitrary q active (non-zero) bits corresponding to the vector W. Let $\Delta_{q|i_1,...,i_q}$ be the difference with q active bits and $i_1, ..., i_q$ be the numbers of digits corresponding to active bits. Note that Δ_1 corresponds to one of the differences $\Delta_{1|1}, \Delta_{1|2}, ..., \Delta_{1|32}$. We shall also denote the difference Δ_q at the input or output of the operation $\mathbf{F}$ as $\Delta_q^{\mathbf{F}\downarrow}$ or $\Delta_q^{\mathbf{F}}$, respectively.

Differential properties of the CP boxes with the given structure are defined by properties of the elementary switching element. Using the

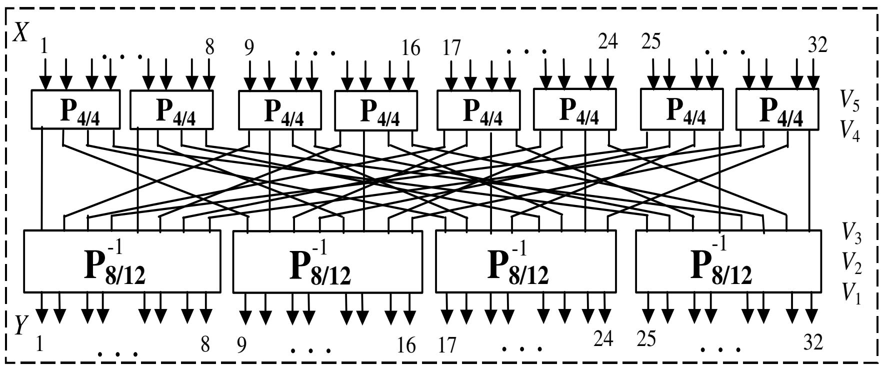


Figure 6. Structure of the box $\mathbf{P}^{-1}_{32/80}$

main properties of the last (see Fig. 7) it is easy to find characteristics of the $\mathbf{P}_{32/80}$-box.

Figure 8 illustrates the case when some difference with one active bit Δ^L_q passes the left branch of the cryptoscheme. The difference Δ^L_q can cause generation or annihilation of $w = 1, 2, 3$ pairs of active bits in the CP box. Let consider the $\mathbf{P}_{32/80}$-box in right branch in the case $q = 1$. The difference $\Delta^L_{1|i}$ is transformed by the extension box into Δ^V_2 or Δ^V_3 (depending on i) at the controlling input of $\mathbf{P}_{32/80}$, i.e. one active bit in the left subblock influences two or three switching elements $\mathbf{P}_{2/1}$ permuting four or six different bits of the right data subblock. Depending on value of the permuted bits and input difference Δ^R_q of the $\mathbf{P}_{32/80}$-box the output differences Δ'^R_g with different number of active bits can be formed by this CP box.

Avalanche effect corresponding to the operations $\mathbf{G}$ is defined by its structure that provides each input bit influences several output bits (except the 32nd input bit influences only the 32nd output bit). Table 3 presents the formulas describing avalanche caused by inverting the bit l_i. One can see that alteration of the input bit x_i, where $3 \leq i \leq 27$, causes deterministic alteration of two output bits y_i and y_{i+1} and probabilistic alteration of the output bits y_{i+2}, y_{i+3}, y_{i+4}, y_{i+5} which change with probability $p = 0.5$. Note that for $i = 1, 2$ alteration

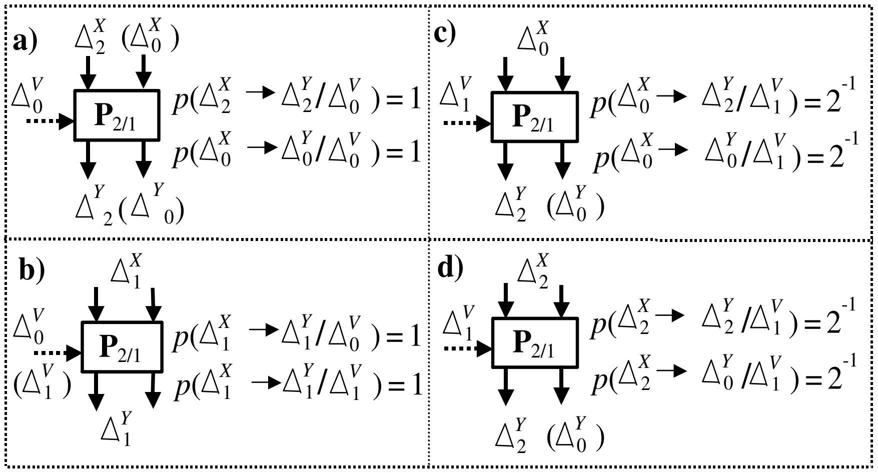
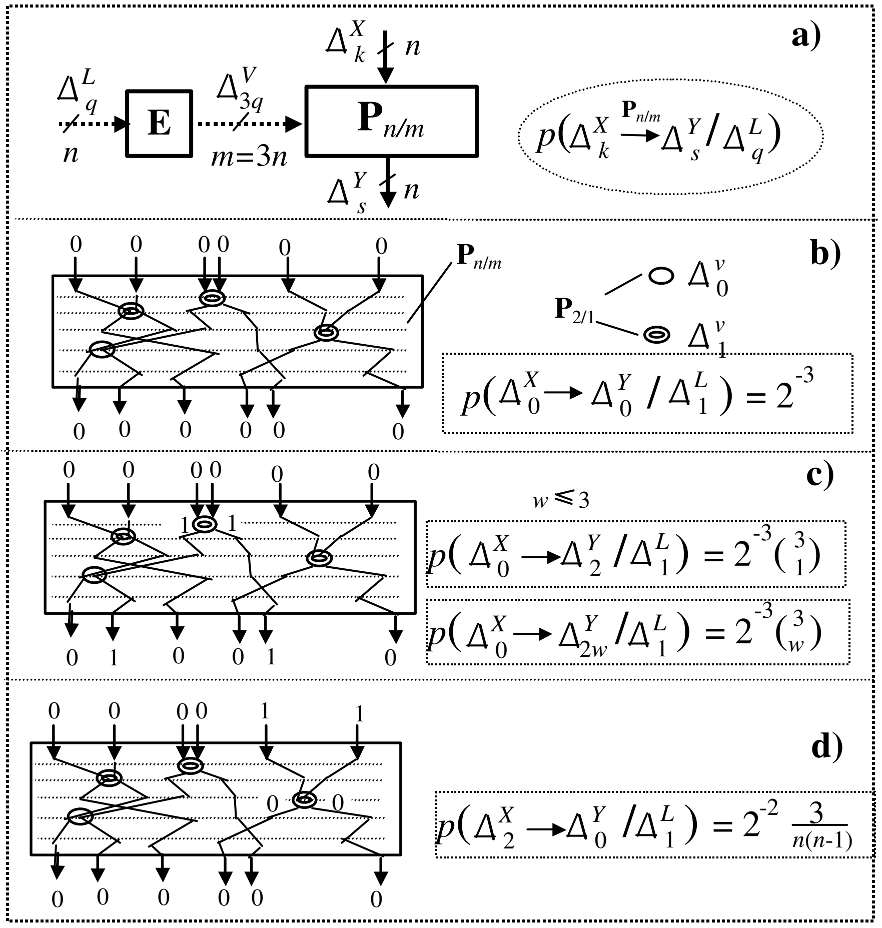

a)
Δ_2^X (Δ_0^X)
Δ_0^V
$P_{2/1}$
$p(\Delta_2^X \to \Delta_2^Y/\Delta_0^V) = 1$
$p(\Delta_0^X \to \Delta_0^Y/\Delta_0^V) = 1$
$\Delta_2^Y(\Delta_0^Y)$
b)
Δ_1^X
Δ_0^V
(Δ_1^V)
$P_{2/1}$
$p(\Delta_1^X \to \Delta_1^Y/\Delta_0^V) = 1$
$p(\Delta_1^X \to \Delta_1^Y/\Delta_1^V) = 1$
Δ_1^Y
c)
Δ_0^X
Δ_1^V
$P_{2/1}$
$p(\Delta_0^X \to \Delta_2^Y/\Delta_1^V) = 2^{-1}$
$p(\Delta_0^X \to \Delta_0^Y/\Delta_1^V) = 2^{-1}$
Δ_2^Y (Δ_0^Y)
d)
Δ_2^X
Δ_1^V
$P_{2/1}$
$p(\Delta_2^X \to \Delta_2^Y/\Delta_1^V) = 2^{-1}$
$p(\Delta_2^X \to \Delta_0^Y/\Delta_1^V) = 2^{-1}$
Δ_2^Y (Δ_0^Y)

Figure 7. Properties of the elementary box $\mathbf{P}_{2/1}$

of x_i causes deterministic alteration of three output bits y_i, y_{i+1}, and y_{i+3}. When passing through the operation $\mathbf{G}$ the difference $\Delta_{1|i}^L$ can be transformed with certain probability in the output differences $\Delta_2^{\mathbf{G}}$, $\Delta_3^{\mathbf{G}}$, ..., $\Delta_6^{\mathbf{G}}$.

3.2 Security of SPECTR-H64

Trying different attacks against SPECTR-H64 we have found that the differential analysis is the most efficient. Our best variant of the DCA corresponds to two-round characteristic with difference (Δ_0^L, Δ_1^R). The difference passes the first round with probability 1 and after swapping subblocks it transforms in (Δ_1^L, Δ_0^R) (see Fig. 9). In the second round the active bit passing through the left branch of cryptoscheme can form at the output of the operation $\mathbf{G}$ the difference $\Delta_g^{\mathbf{G}}$, where $g \in \{1, 2, 3, 4, 5, 6\}$. Only differences with even number of active bits contribute to the probability of the two-round iterative characteristic. The most contributing are the differences $\Delta_{2|i,i+1}^{\mathbf{G}}$. The most contributing mechanisms of the formation of the two-round characteristic belong



a)
Δ_k^X
n
Δ_q^L
E
Δ_{3q}^V
$\mathbf{P}_{n/m}$
$m=3n$
Δ_s^Y
$p(\Delta_k^X \xrightarrow{\mathbf{P}_{n/m}} \Delta_s^Y / \Delta_q^L)$
b)
$\mathbf{P}_{n/m}$
$\mathbf{P}_{2/1}$
Δ_0^v
Δ_1^v
$p(\Delta_0^X \rightarrow \Delta_0^Y / \Delta_1^L) = 2^{-3}$
c)
$w \leq 3$
$p(\Delta_0^X \rightarrow \Delta_2^Y / \Delta_1^L) = 2^{-3}\binom{3}{1}$
$p(\Delta_0^X \rightarrow \Delta_{2w}^Y / \Delta_1^L) = 2^{-3}\binom{3}{w}$
d)
$p(\Delta_2^X \rightarrow \Delta_0^Y / \Delta_1^L) = 2^{-2}\frac{3}{n(n-1)}$

Figure 8. Some properties of the CP box: a - notation of the general case; b - zero difference passes the CP box; c - formation of two active bits; d - annihilation of two active bits.

Table 2. Change of output bits caused by single bit alteration ($\Delta x_i = 1$) at input of the operation **G**

Expression	Probability
$\Delta y_i = \Delta x_i$	$p_{\Delta y_i=1} = 1$
$\Delta y_{i+1} = \Delta x_i$	$p_{\Delta y_{i+1}=1} = 1$
$\Delta y_{i+2} = \Delta x_i(a_{i+2})$	$p_{\Delta y_{i+2}=1} = 1/2$
$\Delta y_{i+3} = \Delta x_i(x_{i-2})$	$p_{\Delta y_{i+3}=1} = 1/2$
$\Delta y_{i+4} = \Delta x_i(b_{i+4})$	$p_{\Delta y_{i+4}=1} = 1/2$
$\Delta y_{i+5} = \Delta x_i(x_{i+2} \oplus x_{i+3} \oplus b_{i+5})$	$p_{\Delta y_{i+5}=1} = 1/2$

to Cases 1a, 1b, 1c, 2a, 2b, 3a, 3b, 4a, and 4b, where $i \in \{1, ..., 32\}$, described below.

Case 1a includes the following elementary events:

1) The difference $\Delta^{\mathbf{G}}_{2|i,i+1}$ is formed at the output of the operation **G** with probability $p_2^{(i,i+1)} = \Pr\left(\Delta^{\mathbf{G}}_{2|i,i+1} \big/ \Delta^{\mathbf{G}\downarrow}_{1|i}\right)$.

2) The difference $\Delta^{\mathbf{P}'}_{2|i,i+1}$ is formed at the output of the CP box $\mathbf{P}'$ with probability $p_3^{(i,i+1)} = \Pr\left(\Delta^{\mathbf{P}'}_{2|i,i+1} \big/ \Delta^{\mathbf{P}'_\downarrow}_{0}\right)$.

3) The difference $\Delta^{\mathbf{P}''}_0$ is formed at the output of the CP box $\mathbf{P}''$ with probability $p_1 = 2^{-z} = \Pr\left(\Delta^{\mathbf{P}''}_0 \big/ \Delta^{\mathbf{P}''_\downarrow}_0\right)$, where $z = 2, 3$, depending on i.

4) After XORing differences $\Delta^{\mathbf{G}}_{2|i,i+1}$, $\Delta^{\mathbf{P}'}_{2|i,i+1}$, and $\Delta^{\mathbf{P}''}_0$ we have zero difference $\Delta^{\mathbf{P}^*}_0$ at the input of the $\mathbf{P}^*$-box. It passes this box with probability $p_4 = \Pr\left(\Delta^{\mathbf{P}^*}_0 \big/ \Delta^{\mathbf{P}^*_\downarrow}_0\right) = 2^{-z}$.

One can denote Case 1a as set of the following events:

$$\left(\Delta^{\mathbf{G}}_{2|i,i+1} \big/ \Delta^{\mathbf{G}\downarrow}_{1|i}\right) \bigcap \left(\Delta^{\mathbf{P}'}_{2|i,i+1} \big/ \Delta^{\mathbf{P}'_\downarrow}_{0}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}''}_0 \big/ \Delta^{\mathbf{P}''_\downarrow}_0\right) \bigcap \left(\Delta^{\mathbf{P}^*}_0 \big/ \Delta^{\mathbf{P}^*_\downarrow}_0\right).$$

Using this form of the represention one can describe other cases as follows ($\forall i, t: \quad t \in \{1, 2, ..., 32\}$, $t \neq i$, and $t \neq i+1$):

$$\text{Case 1b:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}''}_{2|i,i+1} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}'}_{0} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{0}\right).$$

$$\text{Case 1c:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}'}_{0} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}''}_{0} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{2|i,i+1}\right).$$

$$\text{Case 2a:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}'}_{2|i+1,t} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}''}_{2|i,t} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{0}\right).$$

$$\text{Case 2b:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}'}_{2|i,t} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}''}_{2|i+1,t} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{0}\right).$$

$$\text{Case 3a:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}'}_{0} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}''}_{2|i+1,t} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{2|i,t}\right).$$

$$\text{Case 3b:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}'}_{0} / \Delta^{\mathbf{P}'_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}''}_{2|i,t} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{2|i+1,t}\right).$$

$$\text{Case 4a:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

$$\bigcap \left(\Delta^{\mathbf{P}''}_{0} / \Delta^{\mathbf{P}''_{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}'}_{2|i+1,t} / \Delta^{\mathbf{P}'{\downarrow}}_{0}\right) \bigcap \left(\Delta^{\mathbf{P}^*}_{0} / \Delta^{\mathbf{P}^*_{\downarrow}}_{2|i,t}\right).$$

$$\text{Case 4b:} \quad \left(\Delta^{\mathbf{G}}_{2|i,i+1} / \Delta^{\mathbf{G}_{\downarrow}}_{1|i}\right) \bigcap$$

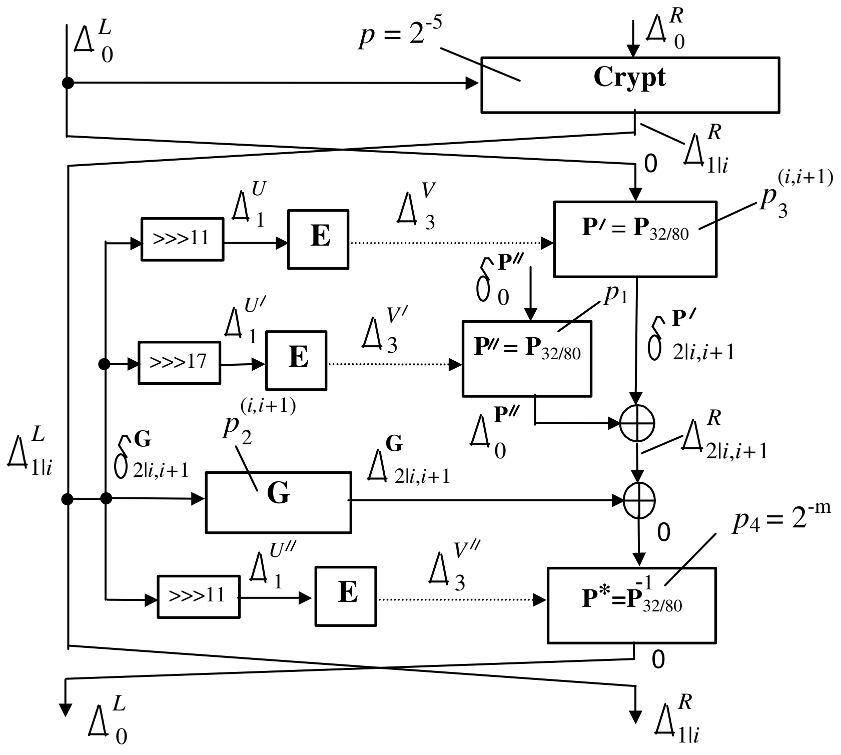


Figure 9. Formation of the two-round characteristic

$$\left(\Delta_0^{\mathbf{P}''}/\Delta_0^{\mathbf{P}''_{\downarrow}}\right)\bigcap\left(\Delta_{2|i,t}^{\mathbf{P}'}/\Delta_0^{\mathbf{P}'_{\downarrow}}\right)\bigcap\left(\Delta_0^{\mathbf{P}^*}/\Delta_{2|i+1,t}^{\mathbf{P}^*_{\downarrow}}\right).$$

There are possible some other mechanisms contributing to the probability of the two-round characteristic and corresponding to generation the differences $\Delta_4^{\mathbf{G}}$ at the output of the operation $\mathbf{G}$. Variants of the formation of the two-round characteristics connected with these mechanisms we shall attribute to the Case 5. Besides, due to the use of the mutually inverse CP boxes $\mathbf{P}_{32/80}$ and $\mathbf{P}_{32/80}^{-1}$ there are possible significantly contributing cases when the box $\mathbf{P}_{32/80}$ generates an additional pair of active bits and the box $\mathbf{P}_{32/80}^{-1}$ annihilates this pair of active bits. Let attribute variants connected with this mechanism to Case 6.

Values $p_1^{(j,t)}$, $p_3^{(j,t)}$, and $p_4^{(j,t)}$, where $j \in \{1, 2, ..., 32\}$, can be easy calculated using the structure of the box $\mathbf{P}_{32/80}$ and distribution of the controlling bits over elementary switching boxes $\mathbf{P}_{2/1}$ (this distribution is defined by Table 1 and the respective bit-rotation operation ("$\ggg$ 11" or "$\ggg$ 17").

For each value $i \in \{1, 2, ..., 32\}$ we have performed the statistic test "1,000 keys and 100,000 pairs of plaintexts" including 10^8 experiments in order to determine the experimental probability $p^{(i)}$ that Δ_0^R passes the right branch of the procedure **Crypt** in the case when in the left data subblock we have the difference $\Delta_{1|i}^L$. Let $s^{(i)}$ be the number of such events. Then we have $\hat{p}^{(i)} = 10^{-8}s^{(i)}$. We have also calculated the probabilities $p^{(i)}$ taking into account the mechanisms of the formation of the two-round characteristic described above. For all i the theoretic values $p^{(i)}$ match sufficiently well the experimental ones $\hat{p}^{(i)}$ (see Table 4) demonstrating that the most important mechanisms of the formation of the two-round differential characteristic correspond to Cases 1-6. The values $p^{(i)*}$ correspond to the modified version of SPECTR-H64+ (see section 3.4).

Probability $P(2)$ of the two-round characteristic can be calculated using the following formula:

$$P(2) = \sum_i p^{(i)}p(i) = 1.15 \cdot 2^{-13},$$

where $p(i) = 2^{-5}$ is the probability that after the first round the active

Table 3. Comparison of the theoretic calculation with experiment

i	1	2	3	4	5	6
#	392	26	117	255	59	50
$\hat{p}^{(i)}$	$1.03{\cdot}2^{-18}$	$1.09{\cdot}2^{-22}$	$1.23{\cdot}2^{-20}$	$1.34{\cdot}2^{-19}$	$1.24{\cdot}2^{-21}$	$1.05{\cdot}2^{-21}$
$p^{(i)}$	$1.08{\cdot}2^{-19}$	$1.43{\cdot}2^{-23}$	$1.35{\cdot}2^{-21}$	$1.9 \cdot 2^{-20}$	$1.13{\cdot}2^{-21}$	$1.39{\cdot}2^{-22}$
$p^{(i)*}$	$1.25{\cdot}2^{-25}$	$1.5 \cdot 2^{-30}$	$0.94{\cdot}2^{-20}$	0	$1.25{\cdot}2^{-21}$	$1.25{\cdot}2^{-21}$

i	7	8	9	10	11	12
#	399	205	679	99	117	388
$\hat{p}^{(i)}$	$1.05{\cdot}2^{-18}$	$1.07{\cdot}2^{-19}$	$1.78{\cdot}2^{-18}$	$1.04{\cdot}2^{-20}$	$1.23{\cdot}2^{-20}$	$1.02{\cdot}2^{-18}$
$p^{(i)}$	$1.13{\cdot}2^{-18}$	$1.6 \cdot 2^{-20}$	$1.13{\cdot}2^{-18}$	$1.31{\cdot}2^{-21}$	$1.36{\cdot}2^{-21}$	$1.56{\cdot}2^{-19}$
$p^{(i)*}$	$0.94{\cdot}2^{-20}$	2^{-21}	$1.3 \cdot 2^{-20}$	$1.5 \cdot 2^{-20}$	$1.5 \cdot 2^{-22}$	2^{-21}

i	13, 14, 15	16	17	18	19,20	21
#	0	467	54054	158196	0	238520
$\hat{p}^{(i)}$	0	$1.22{\cdot}2^{-18}$	$1.11{\cdot}2^{-11}$	$1.62{\cdot}2^{-10}$	0	$1.22 \cdot 2^{-9}$
$p^{(i)}$	0	$1.09{\cdot}2^{-18}$	2^{-11}	$1.5 \cdot 2^{-10}$	0	$1.25 \cdot 2^{-9}$
$p^{(i)*}$	0	0	0	0	0	0

i	22,...,27	28	29	30	31	32
#	0	820	33	141	1901	0
$\hat{p}^{(i)}$	0	$1.07 \cdot 2^{-17}$	$1.38 \cdot 2^{-22}$	$1.48 \cdot 2^{-20}$	$1.25 \cdot 2^{-16}$	0
$p^{(i)}$	0	$1.56 \cdot 2^{-18}$	$1.25 \cdot 2^{-21}$	$1.25 \cdot 2^{-20}$	$1.19 \cdot 2^{-16}$	0
$p^{(i)*}$	0	0	0	0	0	0

bit moves to ith digit. Thus, the performed analysis has shown that the 21st and the 18th digits contribute to $P(2)$ about 88% and the 17th digit contributes to $P(2)$ about 12%. Contribution of other digits is very small. Such strongly non-uniform dependence of $p^{(i)}$ on i is caused by several lacks in the distribution Table 1, nevertheless the ten-round and twelve-round variants of SPECTR-H64 are secure against DCA. Indeed, the difference (Δ_0^L, Δ_1^R) passes ten and twelve rounds with probability $P(10) \approx 2^{-64}$ and $P(12) \approx 1.2 \cdot 2^{-77}$ (for random cipher we have $P = 2^5 \cdot 2^{-64} = 2^{-59} > 2^{-64} > 2^{-77}$).

3.3 Modified version SPECTR-H64+

Differential analysis has shown that the structure of the extension box (i.e. the table describing distribution of the bits of the left data subblock over elementary switching elements of the CP boxes) is a critical part in the design of SPECTR-H64. It is easy to see that small changes in the extension box can cause significant decrease or increase of the probability of two-round characteristic. Taking into account the results of DCA one can easy change positions of the 17th, 18th, and 21st bits and obtain value $P(2) < 2^{-18}$. More accurate modification of the extension box shown in Table 4 gives $P(2) \approx 0.92 \cdot 2^{-22}$. We shall denote SPECTR-like cryptosystem with extension box described in Table 4 as SPECTR-H64+.

For SPECTR-H64+ the most efficient differential characteristic is the three-round one. This characteristic corresponds to the difference $(\Delta_0^L, \Delta_{1|32}^R)$. The peculiarity of the three-round characteristic consists in that the active bit spreads in the second and third rounds through the 32nd digit in the left data subblock (in this case the active bit in the left data subblock generates the single active bit at the output of the operation $\mathbf{G}$ with probability 1). The formation of this characteristic is shown in Fig. 10. This characteristic does not depend on small modifications of the distribution table. The probability of the tree-round characteristic is $P(3) = P_1 P_2 P_3$.

Probability P_1 corresponds to the event that after the first round and swapping the data subblocks we have the difference $(\Delta_{1|32}^L, \Delta_0^R)$.

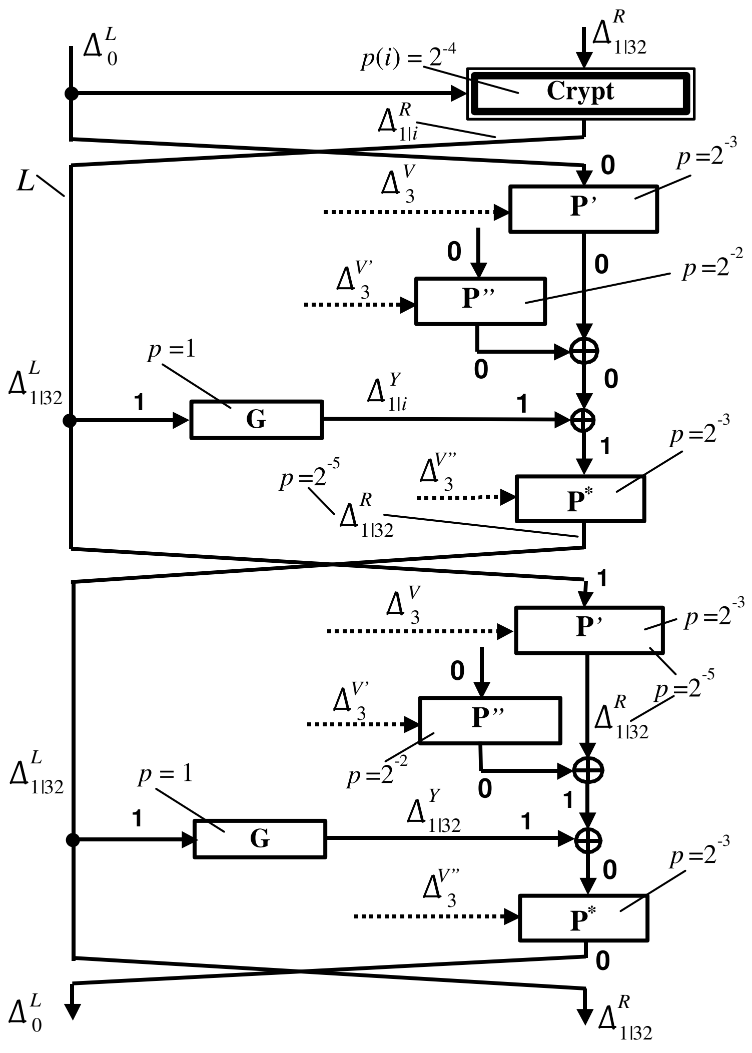


Figure 10. Formation of the three-round characteristic

Table 4. Modified structure of the extension box

V_1	21	26	27	24	28	27	23	24	30	26	32	22	24	30	28	22
V_2	18	19	17	29	25	20	22	25	18	19	31	23	31	21	32	17
V_3	28	20	32	25	26	29	30	29	27	20	21	17	18	19	31	23
V_4	6	7	16	9	10	11	12	13	14	15	8	1	2	3	4	5
V_5	10	11	12	13	14	15	4	9	2	3	16	5	6	7	8	1

Since we consider the case when the difference $\Delta^R_{1|32}$ passes the right branch of the first round, it should be taken into account that the first active layer in $\mathbf{P}_{32/80}$-box and the fifth active layer in $\mathbf{P}^{-1}_{32/80}$-box are controlled with the same controlling vector. Because of the last fact and symmetry of these CP boxes for odd (even) i the difference $\Delta^R_{1|i}$ transforms into $\Delta^R_{1|j}$, where $j \neq i$ and $j \neq i+1$ ($j \neq i-1$), with probability 2^{-5}. The difference $\Delta^R_{1|i}$ transforms into $\Delta^R_{1|i}$ with probability 2^{-4}. It never transforms into $\Delta^R_{1|i+1}$ for odd i and $\Delta^R_{1|i-1}$ for even i. Thus we have $P_1 = 2^{-4}$.

The probability P_2 corresponds to the event that at the output of the second round we have difference $(\Delta^L_{1|32}, \Delta^R_{1|32})$. One can calculate P_2 as $P_2 = 2^{-5}P'P''P^*$, where P', P'', and P^* are the probabilities of the events that CP boxes $\mathbf{P}'$, $\mathbf{P}''$, and $\mathbf{P}^*$, respectively, do not generate active bits. Coefficient 2^{-5} corresponds to the probability that the box $\mathbf{P}^*$ moves the active bit in the 32nd digit. It is easy to see that $P' = P^* = 2^{-3}$, $P'' = 2^{-2}$, and $P_2 = 2^{-13}$.

The probability P_3 corresponds to the event that after the third round and swapping the data subblocks we have the difference (Δ^L_0, $\Delta^R_{1|32}$). One can calculate P_3 as $P_3 = 2^{-5}P'P''P^*$, where 2^{-5} corresponds to probability that at the output of the CP box $\mathbf{P}'$ we have the difference $\Delta^{\mathbf{P}'}_{1|32}$ which annihilates after XOR-ing with output difference of the operation $\mathbf{G}$: $\Delta^{\mathbf{P}'}_{1|32} \oplus \Delta^{\mathbf{G}}_{1|32} = \Delta^R_0$. Using value $P_3 = P_2 = 2^{-13}$ one can calculate $P(3) = 2^{-30}$.

Taking into account the probability of the three-round characteristic one can calculate that six-round SPECTR-H64+ is undistinguishable from a random cipher and conservatively estimate that eight round SPECTR-H64+ is secure against DCA. Thus, due to optimization of the **E**-box structure we have reduced the number of rounds from 12 to 8. This significantly reduces the hardware implementation cost and increases performance.

3.4 Comments on other attacks

Our preliminary study of the security of SPECTR-H64 against LCA has shown that structure of this cipher is also suitable for calculation of the biases of the linear characteristics in the case of few active bits, such characteristics having the largest values of the bias. Our best linear characteristic corresponding to one round has the bias $b(1) \leq 2^{-11}$. A rough estimation of SPECTR-H64 and SPECTR-H64+ for six rounds gives $b(6) \leq 2^5 b^6(1) \approx 2^{-61}$. Thus, these ciphers with six and more rounds are undistinguishable from a random cipher with LCA.

High degree of the algebraic normal form and the complexity of the Boolean functions describing round transformation of SPECTR-H64 prevent the interpolation and high order differential attacks. In spite of the use of very simple key scheduling the described ciphers are secure against slide attack due to non-periodic use of the round subkeys and data-dependent subkey transformation. Truncated differentials attack is prevented, since (1) the data block is transformed as a single unit and (2) each bit of the controlling data subblock influences the selection of the current permutation operation.

3.5 Comments on key scheduling

In the case when encryption and decryption are performed with the same algorithms the direct use of subkeys of the secret key produces the problem of weak and semi-weak keys, the portion of such keys is very low though. It is evident that for SPECTR-H64 the key $K' = (X, X, X, X, X, X, X, X)$ is a weak one (probability to select at random one of such keys is 2^{-192}). One can easy avoid problem of the weak and semi-weak keys introducing minor modification of the initial and final transformations in SPECTR-H64. For example, to implement this one can perform two XOR operations: 1) between subkey A' used in the initial transformation and the parameter $E' \in \{0,1\}^{64}$, where $\forall i \in \{1, 2, ..., 64\}: \; e'_i = e$, and 2) between subkey A'' used in the final transformation and the parameter $E'' \in \{0,1\}^{64}$, where $\forall i \in \{1, 2, ..., 64\}: \; e''_i = e \oplus 1$. After this modification we have e-dependent initial and final transformations: $Y = \mathbf{IT}(X, A' \oplus E')$ and $Y = \mathbf{FT}(X, A'' \oplus E'')$. Another way to prevent weak keys is the use of the rotation operation by j bits, where j is the number of the current round, performed, for example, on the output of the operation $\mathbf{G}$. For majority of the fast implementations this requires no additional hardware resources. Yet another way is the use of some simple key processing, however this requires the use of some additional NAND gates. The use of the simple e-dependent and r-dependent procedures or operations in the ciphers with simple key scheduling appears to be a preferable way to avoid weak and semi-weak keys. The CP boxes suite well to the design of different kinds of the e-dependent permutations.

4 Conclusion

Investigating security of SPECTR-H64 we have shown that security of the DDP-based ciphers depends significantly on the structure of the extension box. The performed analysis of SPECTR-H64 and SPECTR-H64+ has shown that variable bit permutations suite well for calculation of the differential and linear characteristics. Comparative analysis of different attacks against SPECTR-H64 shows that DCA is the most efficient one. Differential analysis presented in this paper allows one to conclude that twelve-round SPECTR-H64 is secure, some optimization of its structure is possible though. Optimized eight-round version SPECTR-H64+ has been proposed. Thus, due to performed security analysis we have found significant reserves in the encryption mechanism of SPECTR-H64 which can be easy used to design new SPECTR-like cryptosystems free of weak keys that will be faster and cheaper in hardware.

References

[1] J.B. Kam and G.I. Davida. *Structured design of substitution-permutation encryption networks*, IEEE Transactions on computers, vol. C-28, no 10 (1979), pp. 747–753.

[2] B.V. Izotov, A.A. Moldovyan, and N.A. Moldovyan, *Controlled operations as a cryptographic primitive*, Proceedings of the International workshop, Methods, Models, and Architectures for Network Security. Lect. Notes Comput. Sci. Berlin: Springer-Verlag, vol. 2052 (2001), pp. 230–241.

[3] V.E. Benes, *Mathematical theory of connecting networks and telephone trafic*, Academic Press, New York, 1965.

[4] A.A.Waksman. *Permutation Network*, Journal of the ACM, vol. 15, no 1 (1968), pp. 159–163.

[5] D.S. Parker. *Notes on shuffle/exchange-type switching networks*, IEEE Transactions on computers, vol. C-29, no 3 (1980), pp. 213–223.

[6] M. Portz, *A generallized description of DES-based and Benes-based permutation generators*, Lect. Notes Comput. Sci. Berlin: Springer-Verlag, vol. 718 (1992), pp. 397–409.

[7] M. Kwan, *The design of the ICE encryption algorithm*, Proceedings of the 4th International Workshop, Fast Software Encryption - FSE '97, Lect. Notes Comput. Sci. Berlin: Springer-Verlag, vol. 1267 (1997), pp. 69–82.

[8] B. Van Rompay, L.R. Knudsen, and V. Rijmen, *Differential cryptanalysis of the ICE encryption algorithm*, Proceedings of the 6th International Workshop, Fast Software Encryption - FSE'98, Lect. Notes Comput. Sci. Berlin: Springer-Verlag, vol. 1372 (1998), pp. 270–283.

[9] R.L. Rivest, *The RC5 Encryption Algorithm*, Proceedings of the 2nd International Workshop, Fast Software Encryption - FSE'94,

Lect. Notes Comput. Sci. Berlin: Springer-Verlag, vol. 1008 (1995), pp. 86–96.

[10] R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin, *The RC6 Block Cipher*, 1st Advanced Encryption Standard Candidate Conference Proceedings, Venture, California, Aug. 20-22, 1998.

[11] C.Burwick, D.Coppersmith, E.D'Avingnon et al. *MARS - a Candidate Cipher for AES*, 1st AES Candidate Conference Proc., Venture, California, Aug. 20-22, 1998.

[12] A.A. Moldovyan, N.A. Moldovyan, *A cipher based on data-dependent permutations.* Journal of Cryptology. 2002, vol. 15, no. 1, pp.61–72.

[13] N.D. Goots, A.A. Moldovyan, N.A. Moldovyan, *Fast encryption algorithm SPECTR-H64*, Proceedings of the International workshop "Methods, Models, and Architectures for Network Security". LNCS, Springer-Verlag, vol. 2052 (2001) pp. 275–286.

A.V. Bodrov, A.A. Moldovyan, P.A. Moldovyanu,

Received August 1, 2005

Specialized Center of Program Systems "SPECTR",
Kantemirovskaya, 10, St.Petersburg 197342, Russia
ph./fax.7-812-2453743.
E–mail: *mold@cobra.ru*

Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

Concept as a Generalization of Class and Principles of the Concept-Oriented Programming

Alexandr Savinov

Abstract

In the paper we describe a new construct which is referred to as concept and a new concept-oriented approach to programming. Concept generalizes conventional classes and consists of two parts: an objects class and a reference class. Each concept has a parent concept specified via inclusion relation. Instances of reference class are passed by value and are intended to represent instances of child object classes. The main role of concepts consists in indirecting object representation and access. In concept-oriented programming it is assumed that a system consists of (i) conventional target business methods (BMs), and (ii) hidden representation and access (RA) methods. If conventional classes are used to describe only BMs then concepts allow the programmer to describe both types of functionality including its hidden intermediate functions which are automatically executed when objects are being accessed.

1 Introduction

1.1 Object Representation and Access

Let us consider a conventional method call: `myRef.myMethod()`. In OOP it is assumed that a reference stores a target object identifier. It is allocated and managed by routines which are not directly controlled by the programmer. For example, memory handles are allocated by the operating system and Java references are provided by the runtime environment. In such an approach the programmer is unaware of how objects are represented and what intermediate actions are performed behind the scenes after a method is called and before its first statement starts. The traditional object-oriented program functionality is concentrated in class methods. It is important that any function executed in the program is the result of some *explicit* method invocation written by the programmer somewhere in the source code. And any object that appears in the program is the result of an *explicit* instantiation. The program itself does not create any data structures and does not perform any actions for its internal use in order to maintain user defined classes.

Such an approach to programming is known to be very simple and efficient for many types of systems because the compiler or runtime takes care of all the object representation and access issues. However, this full automation has its price: the programmer is not able to influence the object representation and access mechanism and is restricted by the standard functionality. Indeed, in many cases the following question arises: What if I want to define my own format of references and access procedures which are developed specially for my system? For example, I might want to develop my own memory manager because the objects I am going to use have a very special format and properties. Or, in addition, my system might need to carry out special security checks whenever its objects are accessed. In all these and many other cases the standard mechanisms of representation and access could be too restrictive. In particular, main memory is only one possible location for objects. In general case they may well be stored in some cache, on disk or on remote computer. Even if a memory manager is very general it cannot cover all the needs of an arbitrary program.

A new approach to programming described in this paper assumes that we can define our own format of references and our own access procedures which are adapted to the purposes of each individual program. Custom references could be defined as integers, text strings or a combination of any other fields. And the corresponding access procedures may include any code because it is written by the programmer. In this case the representation and access mechanism is an integral part of the program it is written for. However, these custom references and access procedures are not used by the programmer anywhere in the source code. It is the task of the compiler or runtime environment to activate them. Thus the method `myRef.myMethod()` does not start immediately because it is necessary to resolve the reference and to find the target object. In this case some intermediate procedures are implicitly activated and this code (which is part of the program) executes after the method call and before its first statement.

1.2 Two Types of Functionality

One of the main general assumptions of the new approach is that there exist two types of functionality:

- *business methods* (BMs) which are defined in classes and used explicitly in the program, and
- *representation and access* (RA) functionality constituting a separate cross-cutting concern and activated implicitly

Business methods or target methods are explicitly used by the programmer in order to access applied functionality of object classes in the traditional OO manner independent of how they are represented and accessed. BMs are precisely what OOP is designed for: we can easily define classes (with reuse via inheritance) and then call their object methods in the program. RA functions (or intermediate functions) introduced and studied in this paper determine how objects are represented and accessed independent of the target BM.

In the described approach we assume that a great deal of program functionality is activated and executed when objects need to be represented and accessed. It is a kind of invisible matter that cross-cuts any program because these functions are hidden, they are not called explicitly in the source code and they are executed behind the scenes. In other words, we assume that such a simple line of code as `myRef.myMethod()` may activate rather complex intermediate functions which are invisible in traditional programs. A general goal of the described approach to programming consists in making this hidden level of functionality an integral part of the program. We need to legalize these functions because they cannot be qualified as something auxiliary while the facilities provided by the standard runtime environments are rather limited for contemporary programs. The thing is that in large program systems RA functions account for most of the program complexity. This is why the level of RA functions should be an integral part of the program that has to be dealt with and developed for this very program.

If we represent a program as consisting of internal spaces (scopes, containers, layers) where objects live then RA functions can be thought of as concentrated on this space borders and automatically executed whenever a process intersects a border on its way to the target object (Fig. 1). Target BMs are executed when the process reaches the target object. According to this analogy each method call is a sequence of steps leading to the target object. (In contrast, in the conventional programming a method call is viewed as only one step leading from the source context to the target.) The target business method specified explicitly in the program is only the last step while intermediate steps involve various RA functions which are hidden and are executed seamlessly behind the scenes. In particular, in such a program source code it would be impossible to find any explicit invocation of an intermediate method. The program might consist of a relatively small number of explicit method calls but be rather complex because of the hidden functionality.

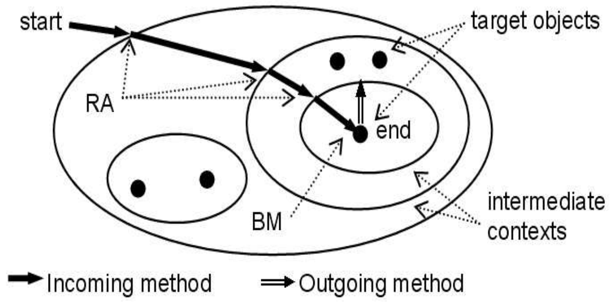


Figure 1. Intermediate borders possess important functions

One of our main assumptions is that the automatically triggered border processes account for a great deal (and even most) of the whole program complexity and reflect specific properties of this program just like its BMs. Hence the programmer needs to be able to describe these functions as an integral part of the program. In other words, a program has to consist not only of BMs (normal classes with their methods) but also include facilities for describing space borders, how objects are represented within particular subspaces and how they are accessed. In such an approach even if a program has little or no BMs at all, it may well be rather complex because of its internal space structure and RA functions associated with space borders and automatically executed when they are intersected by interacting processes.

1.3 Design Goals

It is very important that RA functionality is not associated with any target class but instead, it scatters the whole program. However, it is important that one RA mechanism be described in one place rather than distributed all over the program. This criterion is formulated as the following design goal:

DG1 [Modularization] Representation and access functions have to be described in a modular manner.

This means that if there are two custom RA methods as part of the program then they need to be described in two places. Then these RA functions can be automatically and implicitly injected in all appropriate points in the program where we find it necessary. For example, one custom reference format with its associated resolution methods should be described in one place.

Although RA functions are described in a modular manner we do not want to use them explicitly in all points in the program where they are appropriate. We want only to somehow specify those points so that the compiler could inject the necessary code automatically. It is important that all the method calls are described as usual by specifying a target object and some method. The compiler then uses declarative properties of the target class in order to choose what intermediate actions to execute. However, in the source we do not see those intermediate actions because they are hidden. So the next design goal consists in ability to make normal method calls:

DG2 [Transparency] It is an illusion of instant access. Method calls are made as usual by specifying only a reference and a method without any explicit indication of the type of intermediate actions.

A consequence of this principle is that the use of objects and their BM does not change when we change the underlying RA mechanism. For example, we may have a huge number of BM calls like `ref1.m1()`, `ref2.m2()`, `ref3.m3()` all over the program. In OOP all these objects use one and the same default mechanism of RA. However, in our approach objects of different classes may use different and rather complex RA mechanisms assigned to them (and described in this very program in a modular way). In particular, `ref1` might be identified by a primary key and accessed via JDBC protocol, `ref2` might be represented by an absolute offset in the local heap while `ref3` might be an object on the moon represented via some Universe Unique Identifier and accessed via ISS. The transparency of access guarantees that we do not need to know how objects are represented in order to access their BM. We retain an illusion of instant access when using objects and it is the task of compiler, interpreter or an execution environment to activate all the necessary RA functionality. If we change the way how our objects are represented then the source code where they are used does not change.

It is very important that one and the same RA mechanism could be used to serve many target classes in the program. For example, if we develop a complex hierarchical persistent memory manager with special access rules then it is very natural to use it for any class of object in the program. Thus the following design goal makes sense:

DG3 [Reuse] Many target classes should be able to use one RA mechanism.

Each target class in the program can be assigned some appropriate RA mechanism. And even for individual uses of classes (instantiations) it is desirable to be able to specify how this concrete object should be represented and accessed. However, we want to do it in a declarative manner rather than to control this at run-time. The compiler then uses the declarations in order to activate the necessary intermediate functionality:

DG4 [Declarativity] RA mechanism should be assigned to target classes in a declarative manner.

Assume that there is a module with intermediate RA functionality and a module with target BMs. We want these business methods be accessed via this intermediate functions. There is a design alternative: either (i) to indicate the target module in the context of the RA module, or vice versa (ii) to indicate the RA mechanism in the context of the target module. In other words, who knows whom: intermediate module knows target module, or vice versa? We choose the second alternative:

DG5 [Direction] RA modules do not know target classes they serve and it is each individual target class that should declare what kind of RA mechanism it needs.

The two types of functionality found in any system differ logically rather than physically. This means that they always exist together, cross-cut each other and in most cases cannot be separated. One and the same piece of code can be considered an intermediate function that is activated automatically and a normal business method that is used explicitly. In particular, we do not have one programming construct for business methods and another construct for intermediate functionality. We want to have one mechanism that is able to express both types of functionality:

DG6 [Integrity] A module should not have one dedicated purpose but rather it should be able to express both types of functionality simultaneously.

Assume that for our target classes we specified some concrete RA mechanism. The functions of this RA module are then used before the target methods will be executed. However, in many cases we want this very RA mechanism to rely on some other RA mechanism. In this case the structure of indirection will be nested.

DG7 [Hierarchy] RA functionality described in a modular manner should have a hierarchical structure where parent modules play a role of intermediate layers for their child modules which play a role of targets.

1.4 Concept

In this paper we propose a new approach, called concept-oriented programming (COP), which is based on a special construct called *concept*. Shortly, concept is a combination of one object class and one reference class. Object class is the conventional class as defined in OOP. What is new in this approach is the reference class which complements the object class just like RA functions complement BMs. By combining object class and reference class we make it possible to describe two sides of any system: explicit BMs and implicit RA functions. Object class and reference class have one name (concept name) and may define methods with the same name (any method has a definition within object class and within reference class). Instances of object class, called objects, are passed by reference. Instances of reference class, called references, are passed by value. Informally, the main idea is that references passed by value can represent objects.

Concepts are organized into a hierarchy by using inclusion relation, i.e. any concept has one parent concept. Concepts cannot exist outside an inclusion hierarchy — if a concept does not have a parent concept then it is assumed to be some default concept. Concept hierarchy plays an important role because its structure determines how objects in the program are represented and accessed. In other words, the format of reference and intermediate procedures used to access some target object depend on its position in the concept inclusion hierarchy. Parent concept always indirects representation and access to its child concepts. In order to specify what RA mechanism to use we simply need to include a concept into an appropriate parent concept.

One concept can be interpreted as a space with its own border (Fig. 1). If a concept is included into another concept then it is placed within this parent space. The external space is the root of the concept hierarchy while internal spaces represent its child concepts. Instances of the root concept are represented and accessed directly using some built-in RA mechanism (as if they were OOP objects). Instances of internal concepts are represented and accessed using their parent concepts.

Informally, the difference between classes and concepts is analogous to that between real numbers and complex numbers. Class reflects only an explicit (real) side of software system while concept is able to describe both sides by combining in one construct one object class and one reference class. Reference class of concept describes invisible hidden functionality of a software system like imaginable part of complex numbers. In the same way as complex numbers are much more expressive and natural for mathematical tasks, concept is much more expressive and natural for computer programming.

2 Concept Inclusion Hierarchy

Concept is a generalization of conventional classes defined as a program element consisting of two parts: (i) an *object class* with instances called *objects* and passed by references, and (ii) a *reference class* with instances called *references* and passed by value. For example, Table 1 defines one concept with name `MyConcept`. Its object class has one field referencing an instance of `OtherConcept` (line 2), and reference class has one integer field (line 8) intended for identifying objects of other classes. Both object class and reference class define `myMethod` with different implementations (lines 3 and 9). Note that we do not know here what is the format of field `ref` (line 2) because it depends on how `OtherConcept` is declared. It is a general principle of the concept-oriented programming that reference format and access methods depend on the target class declaration. This reference could have any structure appropriate for our task or we might choose to use the default reference format (OOP approach). However, we can call methods of `OtherConcept` (line 4) as usual and all resolution and other intermediate functions will be executed automatically.

Table 1. Concept is a pair of object class and reference class.

```
00  concept MyConcept in ParentConcept
01    class { // Object class
02      OtherConcept ref; // Indirect reference
```

```
 int myMethod() { // Incoming method
  return ref.getInt();
 }
}
reference { // Reference class
 int id; // Identifies other objects
 int myMethod() { // Outgoing method
  return context.myMethod() + 5;
 }
}
```

Objects are never accessible directly because they permanently exist in some kind of storage. Their position is physical in the sense that it cannot be changed. Objects are accessed via their representatives in the form of references. References on the other hand do not have a permanent position in space. They are travelling elements that move between different points by value. This property reflects the existence of two realities: (i) storage with its address system as a set of permanent addresses, and (ii) a system of information transfer which allows for interactions to be propagated all over the space of objects.

References are elements that exist and can be manipulated only by value. In other words, references do not have their own references and hence represent themselves. References are coordinates in some space or elements of an address system where the address system is a space of objects. For any space to exist two elements are needed: this space itself as an object and its addresses as references. Normally for one object there exist many references. An object can be thought of as a scope or space instance while its references are concrete addresses within this space. A concept then is aimed at describing both space structure and its address structure. In other words, object class describes how the space will look like and how it will function while reference class describes the format of addresses within this space. For example, a country could be viewed as a space where addresses are city names. Its concept then could be written as follows:

```
concept Country
  class { String countryDescription; }
  reference { String cityName; }
```

This concept means that there can be many country objects each having some description. A country then defines its own internal coordinate or address system in reference class. According to this address system any object within one country is identified via some unique city name. Note that here we do not know how countries themselves are identified because concepts define only their internal coordinate system for which they are responsible.

Another example is where we define our own memory manager where objects of any type can be stored. Concepts allow us to define such a storage at high level as an abstract space with its own name and then take responsibility for everything that happens inside this space. In particular, we can define characteristics of the space itself in object class and its address format in reference class. The current number of internal objects is kept in a field of the concept object class. Internal objects themselves are identified by unique integers:

```
concept MemoryManager
  class { int objectCount; }
  reference { int objectId; }
```

We may have many memory managers and then many objects of this concept will be created. Each such memory manager may create many references each of them representing some internal object. Note again that references are not objects because they are passed by value and hence do not have a position in space (their own reference). In contrast, objects have a position in space which is represented by some reference.

Using concepts we can define the format of objects and format of references that are intended to represent objects. Then the question is how do we determine what references represent what objects? For example, what references are used to represent country objects and what references are used to represent memory manager objects in the previous examples?

In order to solve this problem we use the mechanism of *inclusion relation* which means that each concept is included into a parent concept. Thus the whole program is not simply a number of concepts but rather a hierarchy of concepts. In this hierarchy any concept has one parent (explicitly defined) and a number of children (not directly known in its declaration). For example, in Table 1 `MyConcept` is included into `ParentConcept` using keyword '`in`'.

In order to determine what references represent what objects we use the following principle: *an object is represented by its parent concept reference*. This means that there is one-to-one correspondence between this concept objects and its parent concept references. Since references are passed by value they can be used to represent objects in other points of space. For example, in Table 1 all instances of `MyConcept` will be represented by instances of `ParentConcept` reference class. If we want countries to be represented by their country code then concept `Country` has to be included into the following parent concept:

```
concept CountryCode
  class { int countryCount; }
  reference { String countryCode; }
concept Country in CountryCode;
```

This means that all country objects will be represented by means of the corresponding country code. For example, variables that reference Germany will store "DE" as their values. This value will be passed to methods as parameters, returned from methods and stored in local variables and object fields. If we want our memory managers to be represented by long integers then concept `MemoryManager` has to be included into a parent concept with the long integer field in its reference class.

An advantage of such an approach is that the programmer specifies and can change the parent concept declaratively. For example, if we want to make a remote memory manager then we simply change its parent concept which supports remote references. The memory manager itself as well as all its uses in the program need not to be changed.

It is assumed that there exists one *root concept* provided by the compiler, interpreter or an execution environment while all program concepts defined by the programmer are directly or indirectly included into the root. There may be more than one root in the case the compiler provides several standard RA mechanisms, for example, local heap, global heap, managed objects, persistent objects, remote objects etc. Classes and concepts included into the root concept are represented using the system default RA mechanism like memory handles or Java references. Conventional OO program can be viewed as consisting of classes included into the root concept. The root is normally a static concept with a single well known object instance. This is why it does not need a (dynamic) reference and needs not to be resolved. If a parent concept is not specified then by convention it is assumed to be the root concept. The compiler however needs to know what default RA mechanism to use for the root. For example, if we want all objects to be finally represented by memory handles allocated by the operating system in global heap then the root concept will look as follows:

```
concept Root
  class { AllocationTable allocationTable; }
  reference { long memoryHandle; }
```

This system level concept defines its reference class as consisting of one long field and hence all objects at this level will be represented by unique long integers. For example, assume that our custom memory manager is included into such a root concept:

```
concept MemoryManager in Root
  class { Map objectIdToMemoryHandle; }
  reference { int objectId; }
```

This concept will represent all objects of its child concepts by means of integer identifiers. However, each such custom identifier will replace some root memory handle. The mapping between integer identifiers and long memory handles is stored in the field of this concept object class. If we want our target objects to be managed by this custom memory manager then we simply include the target class or concept into it:

```
class MyTargetClass in MemoryMangaer
```

After that all instances of this child target class will be represented indirectly by means of its parent references (integer identifiers). Let us consider the following code:

```
void myMethod(MyTargetClass param) {
  param.targetMethod();
}
```

Here the method parameter has a class that is included into the custom memory manager concept. Hence this parameter will be passed by using integer values. If we need to call some method of this object then this reference (integer value) has to be resolved into its own parent reference. In our case the integer value has to be resolved into some long integer which is a memory handle allocated by the root. The root reference is then used to make a *direct* method invocation. After that the access procedure returns.

An object (instance of this concept object class) where a reference was created is referred to as *context.* Context and its references belong to the same concept. We say that references exist in some context and one context may create many references. The current context is available in the program via keyword '`context`'. For example, line 10 in Table 1 accesses a method defined in the object class of this concept (line 3).

If an object class is static with no instances (with a single instance known at compile time) then such a concept is also said to be static. If a concept has no reference class defined then it is a normal class.

A typical object run-time structure is shown in Fig. 2. Dashed boxes represent contexts (objects of concepts) while black boxes are references within these contexts. The outer most dashed box is the root context, i.e., an instance of the root concept (such as the system default memory manager). The root context has several root references which represent internal child contexts at different depth (not necessarily direct children). In this example there are two child contexts belonging to one concept `MemoryManager`. (In general case there are many child concepts each crating many instances.) Thus there exist two memory managers each managing its own set of objects. For example, the first memory manager allocated two integer references in order to represent two internal objects. However, these internal objects have their own internal objects and so on.

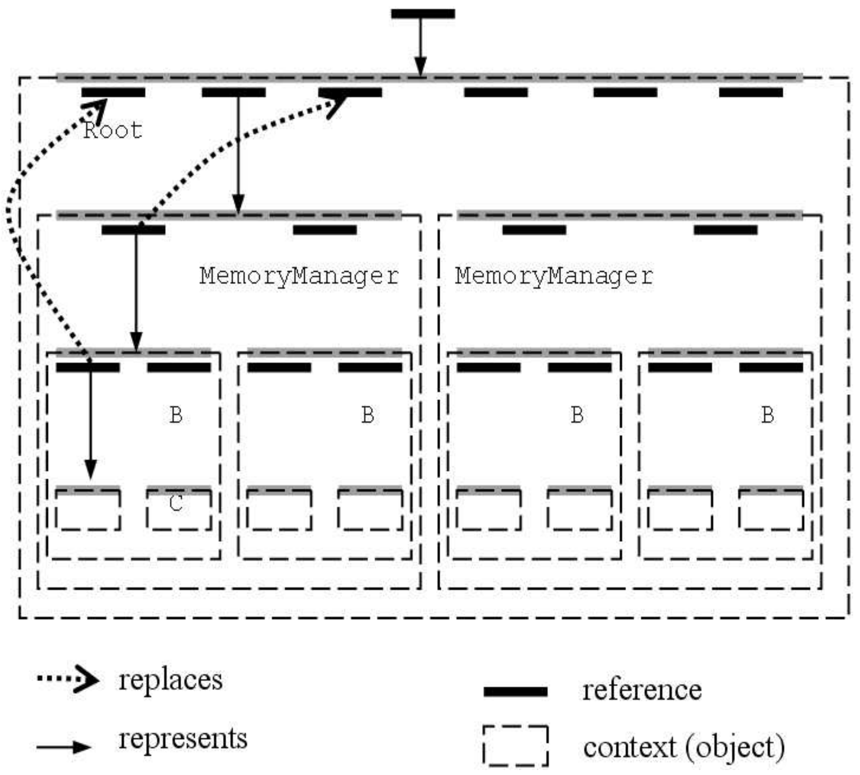


Figure 2. Context structure

It is important that any reference (black rectangle) replaces some parent reference (denoted by upward dot line pointing to a replaced parent reference). At the same time a reference represents some child object (denoted by downward line pointing to represented child object). In both cases these relationships are not necessarily direct. That is, a reference may replace a parent reference of higher order including some root reference. And a reference may represent a child object of higher order including some target object.

3 A Sequence of Access

3.1 Reference Substitution and Resolution

The system root concept provides default format for object references which are used to *directly* represent and access all objects in the program. Here direct RA does not mean that the objects will be really accessed instantly. Rather, by direct access we mean that the programmer is not able to influence this level of RA functionality. For example, we say that Java references provide direct access because the programmer is unaware of the underlying RA mechanism which actually can be rather complex. Even physical memory addresses do not provide the ultimate direct access because they are processed at hardware level and each access requires a number of hardware clock cycles. However, in a program we can assume that such hardware addresses are used for direct access because all the program objects are resolved into them.

One important use of concepts consists in indirecting object representation and access in the program by describing custom format of references and custom access procedures. Program objects are still represented by the root references however these root references are not stored and passed anywhere in the program as representatives. Instead, objects are represented by means of custom references which replace the corresponding root references. Since concepts are organized into a hierarchy, the reference substitution has a nested nature. This means that a child reference replaces some parent reference which in turn replaces its own parent reference and so on till the root which provides direct access to the target object. For example, a street within a city might define its own local notation in order to identify houses. However, in order for the basic access mechanism to work we need to map these local identifiers into the parent city-level identification format. Thus each street-level local identifier will replace some city-level parent identifier for a house. The same substitution mechanism can be used in custom memory managers which can introduce its own local format for object identification. However, these local identifiers replace parent system-level identifiers. Each memory manager is then responsible for storing this mapping and resolving its identifiers.

Table 2. Reference resolution.

```
concept A in Root
  class { static Map map; }
  reference {
    int id;
    void continue() {
      Object o = context.map.get(id);
      o.continue();
    }
  }
concept B in A
  class { static Map map; }
  reference {
    String id;
    void continue() {
      A a = context.map.get(id);
      a.continue();
    }
  }
```

For example, suppose that we want to develop a mechanism for representing our program objects by means of integer values (that replace the system default references). This means that all variables, method parameters, return values and object fields will store integers for those objects rather than the default references. These integer references will live in their own context which is included into the root context. In

Table 2 such a mechanism is described as concept A (line 0-8). Its reference class has only one integer field (line 3) that is used to identify child objects. Thus if we include any class or concept into A then all its instances will be represented by integers, which will be passed by value as the object representatives. It is important that these integers will replace parent references. In Table 2 integer references of concept A replace references provided by `Root` which have unknown format because they are provided by the compiler. For example, in Java integer reference of concept A would replace Java references.

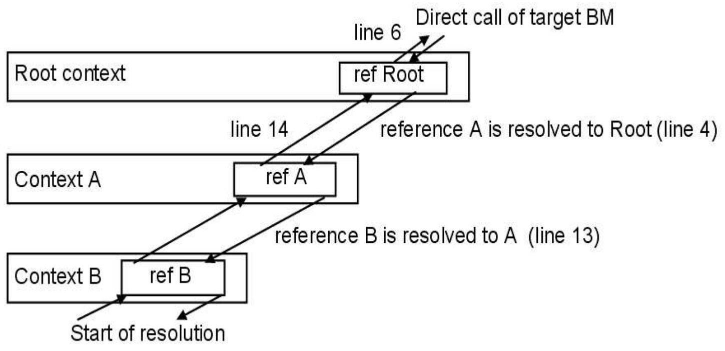


Figure 3. A sequence of reference resolution

Such a substitution is shown in Fig. 3 where the root context has one reference which directly represents some target object in a child context. However, instead of this root reference the compiler will use its substitute of integer type created in context A. Thus the root reference is actually not used anywhere in the program. If we declare a new child concept B and include it into A (line 9 in Table 2) then the substitution will have a nested character. Concept B uses text strings to identify its child objects, that is, any child object will have a text string assigned to it, which is unique within this context. This text string replaces some parent reference of integer type which in turn replaces some root reference providing a direct access to the target object. This hierarchy can be developed further by defining new concepts and including them into the existing ones. The main idea however remains the same: any child object will be automatically represented by its parent concept reference which replaces some root reference.

It should be noted that it is not necessary that a reference replaces its direct parent reference. In particular, a reference can replace its root reference. For example, in Table 2 and Fig. 3 concept `B` might well be designed in such a way that string identifiers replace root references (rather than integers of concept `A` which then replace root references). It is important also that reference substitution is performed and makes sense within some concrete context only. In particular, the context stores all the information that is necessary to resolve indirect references into their parent counterparts. In Table 2 such information is stored in a field of the concept object class (lines 1 and 10 for concepts `A` and `B`, respectively).

Concept hierarchy is intended to describe a reference substitution order where child references replace their parent references. In this way the programmer can describe internal space with its own local coordinate system that indirects the parent coordinate system. However, one of the design goals of the concept-oriented approach is that this indirection mechanism has to be transparent when it is used (DG2). In other words, when we use our target objects we do not need to know how they are represented and what is necessary to do in order to access them. In particular, we do not need to know the format of their references and how these references are resolved. For example, if we want objects of class `C` to be represented by text strings then we include it into concept `B`. After that we use objects of class `C` as usual and it is the parent concept that is responsible for the resolution of custom references.

The mechanism of reference resolution is implemented at the level of each concept that defines its own references. Thus each concept that defines its references which replace parent references is also responsible for their resolution whenever some target object needs to be accessed. Such a resolution is implemented in the special method of reference class called `continue` (lines 4 and 13 in Table 2). The role of this method consists in providing a door or portal between level. In this sense it is not a normal method in the sense of object-oriented programming but rather a mechanism for border intersection and context change. It is declared as a method because its implementation is provided by the programmer who decides what should happen when a process intersects this border (not necessarily only reference resolution). The continuation method takes no parameters and returns no value. It is applied to a parent reference in order to continue the current process in the parent context.

Whenever an indirectly represented object is going to be accessed, its reference is automatically and transparently resolved by the continuation method. Reference resolution (continuation) method is applied to an instance of reference class and executes in the context of its object class. For example, continuation method of reference `A` executes in the context of an object of class `A` while continuation method of reference `B` executes in the context of some object of class `B`. Let us assume that class `MyTargetClass` is included into concept `B` and then its business method `targetMethod` is called from somewhere in the program:

```
00  MyTargetClass in B {
01    void targetMethod() { ... }
02  }
03
04  void myMethod(MyTargetClass param) {
05    param.targetMethod();
06  }
```

Notice again that when we call the target method (line 5 above) we do not know how this object is represented and how it will be accessed. It is the task of compiler to find all the parent concept and use their functionality to organize the resolution procedure. In our example the target object is included into concept `B` and hence it will be represented by a text string. When a method of this object is invoked the compiler needs to resolve this text string into the corresponding root reference and then make a direct method call. Thus the compiler applies continuation method of reference `B` to the reference representing the target object (start of resolution in Fig. 3). The continuation method (lines 13-16 in Table 2) has to decide how to resolve this reference (text string). In our example, it restores the replaced parent reference given the value of this reference using information from the context (line 14). When the parent reference is restored it simply passes the control by invoking the parent continuation method (line 15). This means that the process intersects the border and proceeds in the parent space (in concept `A`). Continuation method of reference `A` is implemented in a similar manner. Its task consists in restoring a root reference given some integer value as a key. When the root reference is found it again proceeds by invoking the parent continuation method (line 6). However, in this case it is a root reference and hence its implementation is provided by the compiler. We do not know what concretely happens in the root continuation method however its task consists in calling the target business method (direct call of target BM in Fig- 4). It is possible now to make a direct call because the object reference is completely resolved and this is precisely what happens when we call a method in OOP. When the target method finishes the whole procedure returns and the continuation method can execute some clean up procedures.

It should be noted that the continuation method is provided by the programmer and can use any resolution strategy or include any other necessary code. It is important only that the compiler will follow the concrete sequence of steps when an object is going to be accessed. In particular, the continuation method may include more complex logic then simply object resolution. Its general purpose consists in providing a mechanism for border intersection and code that will trigger automatically whenever a process wants to intersect this border. This is precisely the code that is hidden in the conventional object-oriented programming.

3.2 Context Resolution

In the previous section we described how references are resolved in continuation method using information from the current context. However, one problem is that contexts in most cases are not static. Rather, they are normal objects with their own references as shown in Fig. 2. For example, when an integer reference is being resolved we need to access information from its memory manager which itself is represented by its parent reference. In particular, lines 5 and 14 (Table 2) cannot be directly executed because each context is represented by its parent reference (just like any other object).

An important conclusion is that it is not enough to store only an object parent reference as has been described in the previous section. For complete representation it is necessary to store also references to all the parent contexts of the target object. For example, it is not enough to store only a street name because it is specified relative to its city (context) which in turn is specified relative to its country and so on till the root context (which is static).

Such a hierarchical approach to object representation is implemented via the mechanism of *complex references*. A complex reference is a sequence of several reference *segments*. Each segment is an instance of one reference class. The very first (high) segment is of root type and represents the first context within the root where all child objects live. The next segment is of child concept type and so on till the target class. For example, in Table 2 a target object of `MyTargetClass` included in concept `B` would be represented by three segments: high root segment representing context `A`, middle segment of integer type representing child context `B`, and low string segment representing the target object. Such a reference might be equal to <`0x123, 10, "objectUniqueName"`> where `0x123` is the value of the root (system default) reference, 10 is the value of reference `A` and `"objectUniqueName"` is the value of reference `B`.

What happens if we get a reference of target class `MyTargetClass` and then call some its method `c.myMethod()`? In the previous section we described this process as a resolution of the target reference into the corresponding root reference using information in the intermediate contexts. However, now our object is represented by three segments rather than only one low segment. The first two segments represent the two intermediate contexts. One approach solving this problem consists in resolving these intermediate contexts each time we need to access them from the child context. However, this technique is rather inefficient because context is supposed to be used very intensively. An alternative approach consists in changing the sequence of access. Now high segments are resolved before low segments and the result of the resolution is accessible from all child contexts. In other words, the procedure described in the previous section is repeated for each segment of the complex reference starting from the high segment and ending with the last low segment representing the target object. The main advantage is that parent contexts are guaranteed to be resolved and directly accessible from any child context.

In our example shown in Table 2 the compiler determines that the target object of `MyTargetClass` has three parent concepts and hence is represented by three segments. Although only the last segment represents the target object, in order to resolve it, we need its two intermediate segments. So the compiler in this situation starts from resolving the very first (high) segment. It is however of the root class and hence is already in the default system format that can be used for direct access. On the second step the compiler resolves the second integer segment of concept `A`. When this segment is resolved, the corresponding root reference represents the next child context. After that the next string segment is resolved into the root reference which represents the target object.

Table 3. A sequence of access.

```
concept A in Root
  class {
    Map map;
    void continue() { // Incoming method
      continue(); // Outgoing (reference) method
    }
  }
  reference {
    int id;
    void continue() {
      Object o = context.map.get(id);
12        o.continue();
13      }
14    }
15
16  cconcept B in A
17    class {
18      Map map;
19      void continue() { // Incoming method
20        continue(); // Outgoing (reference) method
21      }
22    }
23    reference {
24      String id;
25      void continue() {
26        A a = context.map.get(id);
27        a.continue();
28      }
29    }
```

Such a sequence of access (from high to low segment) is supported by a special method of object class called `continue`. Note that this method has the same name as the method of reference class. In other words, each concept has two continuation methods: one defined in reference class used to resolve one segment (described in the previous section), and another defined in object class used to enter just resolved context. (The same is true for any other method of concept.) Object continuation method takes no parameters and returns no value (line 4 and 19 in Table 3). It is called after this object is resolved and is going to be accessed, that is, after the border intersection. Using this continuation method the process enters the scope of the object after its parent reference is resolved. In contrast, reference continuation method (line 10 and 25 in Table 3) is applied to reference and its role consists in resolving it to a system default reference that represents a child object and can be used for direct access.

A generic sequence of access using two continuation methods is shown in Fig. 4. A target object of class `C` is represented by three segments 1, 2 and 3 created in contexts `Root`, `A` and `B` respectively. After entering the root context it is necessary to resolve segment 1 which represents the next context. However, it is already resolved because all root references (1, 4 and 6 in this example) are direct and have system default type. Therefore we follow a dash line and enter the child context `A` using its continuation method (line 4 in Table 3 and thick arrow in Fig. 4).

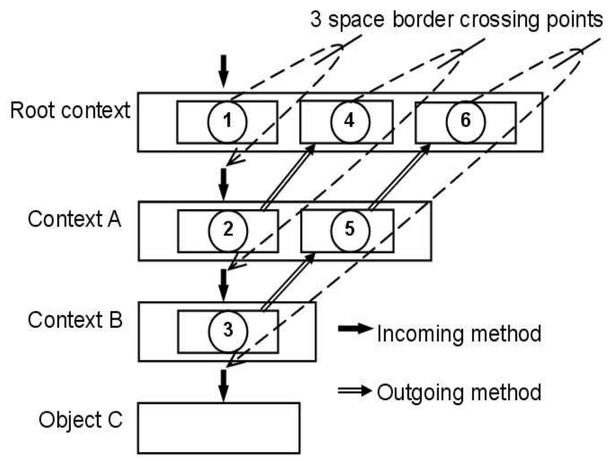


Figure 4. A sequence of context resolution

This method does nothing in our example and simply proceeds by resolving its own reference (line 5 and double arrow from 2 to 4). Root reference 4 is a direct representation of the next child context `B` so we again follow a dash line and get into the context `B` using its object continuation method (line 19 and thick line). Here we need to resolve the last segment 3 by applying continuation method of its reference (called from line 20 and defined in line 25). This method resolves reference 3 to reference 5 of its parent concept `A` using information in the context (line 26) and then calls the parent resolution method (line 27) which resolves 5 to root reference 6 in the same way. Thus reference 6 is a direct representation of the target object and we can enter its scope and call its business method specified in the source access request. Note that the sequence ⟨reference 3, double arrow, reference 5, double arrow, reference 6⟩ is the resolution process described in the previous section, which uses reference continuation method.

3.3 Dual Methods

In the previous sections we mentioned that both reference class and object class of concept may define a method with the same name. However, in the concept-oriented program a method is invoked by using only its main name with no indication whether it belongs to a reference class or an object class. Then the following question arises: which of two versions has to be executed? One example has been considered in the previous section. The special continuation method defined by the programmer and used by the compiler to organize transparent access is defined both in reference class and object class. The reference continuation method is used to resolve the current reference while the object continuation method is executed when the process enters the current context. It is also possible to define any normal method of a concept as consisting of two parts: one within reference class and the other within object class. Methods of object class are referred to as *incoming* because they are visible from outside and are executed when any process enters this object scope (double line arrow in Fig. 5). Objects of reference class are referred to as *outgoing* because they are visible from inside and are called when this object is used by its child objects (thick arrow in Fig. 5). For example, `MyConcept` in Table 1 has incoming method (line 3) and outgoing method (line 9) which have the same name.

One principle of the concept-oriented programming is that functionality is concentrated on space borders and is automatically triggered whenever a process intersects it. The idea of dual methods is that we want to separate this functionality depending on the direction in which the process intersects the border. At the same time we want to have only one method name for both functions. This allows us to call object methods by specifying as usual only its method name while one of two versions will be automatically chosen depending on the relative position of the source context. If we are calling the target object method from outside then one version will be executed. If the method is called from inside then the other version will be executed.

Using concept inclusion hierarchy this principle means that one of two versions will be chosen depending on whether the target object method is called from some its child object (inside) or from any other object. Thus any object defines a border or scope and if a method call comes from the side of its child object (lower part in Fig. 5) then the version defined in reference class is executed. If a method call comes from the side of its parent object (upper part in Fig. 5) then the version defined in the object class will be executed. However, in the source context the method call looks the same. We say that if an object is not yet reached and its reference is not resolved then the process is outside. As soon as the object reference is resolved the process intersects the border and gets inside this scope.

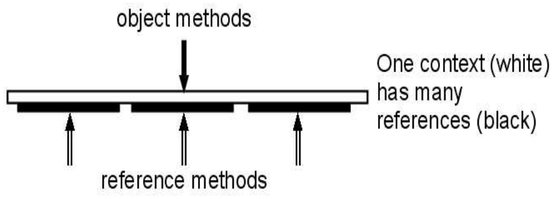


Figure 5. Object methods and reference methods of concepts

Another important idea behind dual methods is that applying a method to a reference is not considered a specification of concrete code to be executed. Methods in the concept-oriented programming are thought of as symbolic names for doors in space borders. Each intermediate border (concept) can define several doors with their symbolic names and one universal door via continuation method. Each door has two directions, name and code assigned to it. This code will be automatically executed whenever an access request with this method name intersects it. Thus method invocation in COP is a way to specify a door through which it is necessary to go rather than a concrete action or code. Reference in this case contains a path to the target.

Below some other properties and uses of dual methods are described:

- Object methods are called from parent objects (from outside) while reference methods are called from child object (from inside).
- Object method is called after complete resolution of its parent reference, i.e., it is the first method to be executed after this object reference has been completely resolved. Entering object method means intersecting this object border.
- Object methods serve external objects while reference methods serve internal (child) objects.
- Object method can delegate or forward its task to its internal object and all requests to child objects pass through the parent objects. This is analogous to intercepting methods of extensions by base objects in OOP.
- Reference method implements functionality exposed to its child objects. This is analogous to implementing methods of extensions using base methods.
- An object accepts method calls from outside using methods of object class and then serves its child objects by providing methods of reference class.

One of the most important mechanisms that can be implemented using dual methods is life-cycle management. At least two functions are needed to implement this mechanism:

- object initialization/clean up, and
- reference initialization/clean up

Object initialization method is normally called constructor while object clean up procedure is called destructor. Reference initialization and clean up procedures do not have their own conventional names because this mechanism is not part of OOP, which provides RA functionality as a default mechanism. In the concept-oriented programming both types of life-cycle management methods are equally important: object creation/deletion and reference creation/deletion. In this situation dual methods are precisely what we need to implement this mechanism. Namely, each object class has methods `create` and `delete` which are responsible for construction and destruction (initialization and cleaning up). On the other hand, each reference class has the same pair of methods (line 6, 22 for creation and 11, 27 for deletion in Table 4) that are responsible for reference allocation and deletion. The reference creation method is applied to a new (empty) instance of reference class. After return this reference has to contain some correct value identifying the child object for which it has been created.

Object creation and deletion follow the standard sequence of access described in the previous section (Fig. 4). An example of reference creation and deletion is shown in Table 4 (object creation is a constructor which simply initializes object fields). Just like the method of continuation, creation and deletion methods take no parameters and return no values. Assume that the `MyTargetClass` is included into concept `B` and we create its instance as follows: `MyTargetClass target.create()`. This statement is equivalent to the conventional `MyTargetClass target = new MyTargetClass()`. Variable `target` will contain a reference of parent concept `B`, i.e., a string identifier as last segment. In order to initialize this reference the compiler will apply creation method of `B` (line 22). This method will generate a unique string in order to identify new object in the current context (line 23). Then it allocates a parent reference and calls the same method of creation to initialize this new empty reference (line 24). Finally the creation method remembers an association between this id and the parent reference within the current context (line 25). This information can be then used whenever this object needs to be accessed, for example, from reference continuation method.

Line 24 is a reference creation method which is applied to empty reference of concept `A`. It is implemented in the same way. First, it generates a unique integer (line 7), then it creates a parent reference (line 8) and finally it remembers an association between them in the current context (line 9). Line 8 is the most important because here we create a root reference which directly represents the target object being created. When line 8 is being executed the target object is really created at the system level. It is precisely the moment when the target object constructor (its object creation method) is called. Notice that if it were a normal business method call then at this moment the compiler would call the target object method. So the target object constructor plays the role of normal business method because it is the last method that is executed in the sequence of access. It is important that object constructor is executed after its root reference has been created and before the creation procedure returns. In other words, lines 9 and 25 are executed after the target object constructor.

Reference deletion is performed analogously. If we need to delete an object then deletion method is applied to its reference: `target.delete()`. If the reference consists of several segments then they need to be resolved as usual starting from high segment and ending with the last low segment which represents the target object. When the last segment (concept `B`) is reached, its reference deletion method is called (line 27 in Table 4). Here we resolve this reference and find its parent reference of concept `A` (line 28). Then this parent reference is deleted by calling the same deletion method of parent concept (line 29). And finally information about this identifier and the parent reference is deleted from the context (line 30). After that this reference is invalid and cannot be used to access the target object. Line 29 is a call of the parent deletion method (line 11) which works similarly. Line 13 is the most important here because it is where the root reference and hence the target object is really deleted (after its destructor).

Table 4. Creation and deletion.

```
01  concept A in Root
02    class { static Map map; }
```

```
  reference {
    int id;
    void continue() { ... }
    void create() {
      id = context.getUniqueInteger();
      Object o.create();
      context.map.add(id, o);
    }
    void delete() {
      Object o = context.map.get(id);
      Object o.delete();
      context.map.remove(id);
    }
  }
concept B in A
  class { static Map map; }
  reference {
    String id;
    void continue() { ... }
    void create() {
      id = context.getUniqueString();
      A a.create();
      context.map.add(id, a);
    }
    void delete() {
      A a = context.map.get(id);
      A a.delete();
      context.map.remove(id);
    }
  }
```

4 Uses of Concepts

4.1 Concept as a Generalized Class

Concept as a programming construct is a pair of one object class and one reference class with special responsibilities described in the previous sections. In such a form it is a rather general instrument that has many possible uses. In other words, concept can be applied in very different forms in very different programming languages depending on the goals. One possible application of concepts is interpreting them as a generalized class. The idea is that the conventional object-oriented programming languages can be then used as usual except that concepts are used instead of classes. Concept inclusion is interpreted as a generalized inheritance. Notice that if all concepts have only an object class with empty reference class then we get the conventional object-oriented case.

One important new property of the concept-oriented approach is that each intermediate object in the hierarchy has its own reference and life-cycle. In contrast, in OOP an object has always only one reference independent of the number of its base objects. For example, if class `Circle` inherits class `Figure` then in OOP all instances of class `Circle` will have a reference which is also valid for its base object of class `Figure`. In COP in general case it is not so. For example, if concept `Figure` is included into concept `Panel` then all panels will have their own unique references which are independent of references allocated for figures. For each panel identified by some reference there can be many figures with their own references allocated by its parent panel. Within one concrete panel, figure objects are identified by their short (local) references while outside in global scope figures need to be identified by their long references consisting of two segments. In larger scope we might need even more segments. For example, if panels are included into windows then each figure reference consists of three segments starting from window reference.

Internal objects have also their own independent life-cycle. We can create and delete internal objects independent of their parent objects. In OOP it is not so, and creating/deleting an object means creating/deleting all instances starting from the root class and ending with the last extension. Independent life-cycle is maintained by means of dual creation and deletion methods. If a concept is being developed to maintain its internal coordinate system then it should define the corresponding methods for reference creation and deletion that will serve its child objects. Then its child objects will have references allocated by the parent which will be also responsible for their resolution via reference continuation method.

In the concept-oriented programming the role of concepts changes significantly with respect to the role of classes. The main role of base classes is object and functionality reuse. This means that base classes are developed as pieces of generic functionality that can be then inherited and extended from child classes. The main role of concept in COP consists in implementing a scope or space border with associated functionality. Then any object that is created within this concept inherits this behaviour and can use it at run time. Objects are created and function within a hierarchical space which determines many their properties. In OOP it is done statically at compile time while in COP this inheritance of behaviour and influence of context is performed dynamically at run time. As a scope or space border any concept has to behave like an intermediate environment rather than an end point in OOP. The main goal of such an environment consists in processing incoming and outgoing access requests (method calls). For example, a concept might accept an incoming request from outside in its object method and after some processing continue its execution in a child object (delegation):

```
concept Intermediate
  class {
    int total=0;
    int requestCount=0;
    void someMethod(int amount) {
      total += amount;
      requestCount++;
      continue();
      requestCount--;
09      }
10    }
11    reference { ... }
```

Here we count the number of method calls which are currently being processed in variable `requestCount` (line 3) and also sum up all the method parameters in variable `total` (line 2). Both variables are stored in the current context (object class) so we always know how many internal objects are being currently accessed via this method (door in the border). Additionally we can determine the sum of all parameters passed to this method. Line 7 is where we pass this request to the internal object for additional processing that is specific to the extension. Here we clearly see that this method implements an intermediate functionality that is however activated automatically.

4.2 Concept as an Active Namespace

Conventional namespaces are static and passive constructs that extend class naming system. One interesting use of concepts consists in interpreting concept hierarchy (without target classes) as an active and dynamic naming system. In contrast to conventional namespaces which are processed at compile time, active namespaces implemented via concepts exist at run-time. The main idea of this mechanism is that there exist two roles of concepts: concepts as target classes, and concepts as namespaces. The former role has been considered in the previous section while in this section we consider how concepts can be used as active namespaces. For simplicity we will assume that target concepts are normal classes (without reference class and other specific features of concepts). Any target class is included into some active namespace described as a concept. However, in contrast to conventional namespaces this effectively changes their behaviour at run-time because access to all class instances is intercepted and indirected by its parent namespaces. Classes included into Root namespace will be accessed directly without any intervention while each internal namespace will add its own level of indirection and its own intermediate functionality.

The easiest way to implement active namespaces consists in using static concepts where object class has only static members and hence does not produce run-time instances. In this case each namespace can be declared as one reference class with static members belonging to object class and non-static members belonging to reference class. For example, in Table 5 we declare namespace `Persistent` (line 0). Its static fields (lines 1) belong to object class while all other (non-static) members belong to reference class. Any target object with the class included in `Persistent` namespace will be identified by a unique integer value which is supposed to correspond to this object primary key in a database. Continuation method of this namespace (line 4) is a reference method. Its task consists in transforming this primary key into the target object root reference. It has to load the target object from the database before it can be accessed (line 4) and freeing this object after the access has been finished (line 6).

Table 5. An example of active namespace.

```
00  namespace Persistent in Root {
01    static Map map; // Static member (object class)
02    int primaryKey; // Dynamic members (reference class)
03    void continue() {
04      Object o = context.restore(primaryKey);
05      o.continue();
06      context.free(primaryKey, o);
07    }
08  }
09
10  class C in Persistent {
11    void create() { ... } // Constructor
12    void delete() { ... } // Destructor
13  }
```

Using the mechanism of active namespaces the programming is reduced to designing the structure of namespace and then including target classes into them. The namespace structure accounts for a great deal of the program functionality however its functions are used implicitly. A target class may change its behaviour depending on its parent namespace.

4.3 Applications of Concepts

Concepts are useful in applications with complex structure characterized by a great deal of intermediate functionality cross-cutting the whole system. This includes the following technologies and mechanisms:

Access interception. Frequently we need to perform some actions before the target object is reached. This can be done by defining an object class method of parent concept which then will intercept all calls. If it is necessary to intercept all calls then we define object class continuation method.

Security and object protection. Before an internal child object is reached we would like to perform some security checks. This can be done in object methods of parent concepts.

Persistence. Before a target internal object can be accessed it might need to be loaded from persistent storage or activated in some other way. This can be done in object class methods as well as in the continuation method of reference.

Debugging, tracing and logging. Incoming methods can be used for auxiliary purposes such as controlling access to objects. We can define special concepts in order to control access to internal classes by intercepting specified method calls.

Internal services. Each object may define service functions to be used exclusively from inside by internal objects by means of its outgoing methods. If classes are included into such concepts then they can use these services which are not visible from outside.

Memory and life-cycle management. We can use this mechanism to implement custom memory managers. For example, it might be necessary to create an efficient memory manager for special types of objects like a hierarchical buffer or a local heap. Persistent storage can also be viewed as a special type of memory manager.

Layered structure of containers. Concepts effectively define space borders and serve as run-time containers serving their internal child objects. Such containers are environments for their objects providing all the necessary services including life-cycle management.

Remote objects. This mechanism is very suitable for implementing network protocols and remote method invocation mechanisms. Concept can be responsible for network communication and reference resolution. Its incoming methods accept remote calls while outgoing methods provide local services for internal objects as a local context.

Protocol stacks. The hierarchical structure of concepts can be used to implement the mechanism of protocol stack which is especially useful for distributed systems. In this case a reference class describes a packet header with information about the target object position. A complex reference is viewed as a nested structure of packet headers. The first high segment of the complex reference is the first header of the external packet. The body of this packet starts from the second segment of the reference and so on. Each intermediate segment (internal packet header) represents one intermediate context. Concepts allow us to create custom protocol stack for each individual program and then use it transparently.

Lazy creation and deletion. Here object reference is initialized without the real object creation. For example, we could simply generate a unique text string as object reference and exit. And only when this object is really accessed (and we cannot resolve the string identifier) the continuation method performs the rest of the creation procedure.

Transactionality. It is convenient to develop concepts which are responsible for performing operations with internal objects as one transaction. In particular, such a concept will automatically and transparently begin a transaction for each incoming access request and end this transaction after the access is finished.

Synchronization and multi-threading. Concepts can be used to implement a complex mechanism of synchronized access to some internal resources. For example, such a concept can guarantee that only one process accesses one object. It will store a list of objects being currently accessed as well as a queue for processes waiting at the border (in front of the door).

5 Related Work

An approach described in this paper is being developed within a new paradigm which covers several branches in computer science including programming, data modelling [1,2,3] and system design. The concept-oriented paradigm is based on several general principles that distinguish it from the currently existing theories and approaches. In the context of programming the most important concept-oriented assumption is that system functionality is concentrated on space borders. In contrast, in object-oriented paradigm it is assumed that most of functionality is concentrated in objects themselves. Concept as the main programming construct allows the programmer to describe effectively not only what happens in objects but simultaneously what happens when they are being accessed.

The concept-oriented programming can be considered a continuation of a very general and deep principle of Separation of Concerns formulated by Dijkstra [4]. The main idea of this principle is that any problem or system functionality can be viewed from different points of views or concerns. One specific feature of our approach is that we distinguish two main concerns any program consists of: BMs and RA. Currently there exist different techniques for separating business methods from representation and access functionality but they can be broken into two main categories: methods based on dedicated middleware and approaches based on programming languages.

The idea of middleware-based approaches consists in creating special software and hardware environments where a conventional program will run. Such an environment offers a number of functions that are intended to support custom RA functionality. This special environment can exist and be accessible to running programs in very different forms, for example, as part of an operating system, an object container, a service, a dynamically or statically linked library etc. However, the main property of this approach is that the programming language remains the same while the support is provided by developers of the middleware. In particular, it is not easy to develop a new custom environment or adapt the existing environment to the purposes of each concrete program.

One wide-spread class of middleware is techniques for remote procedure calls. Examples of such middleware platforms are CORBA and RMI/EJB [5,6]. Such an environment provides facilities for creating remote references and then making transparent method calls. Although such an approach provides much more flexibility in comparison with the manual remote method invocation, they still have serious limitations. First of all we are not able to change the format of remote references and the underlying invocation protocol. Such middleware platforms may fit well to the purposes of one system but may be inappropriate for another system. Their adaptation possibilities are very limited and such an environment is separated from the rest of the program.

A more flexible approach to separating two concerns consists in using reflective environments and metaobject protocols [7,8,9]. The idea of this approach consists in providing a mechanism for changing the behavior of the language from this very language. Normally programming languages are defined in such a way that their behavior cannot be changed. In particular, we cannot change how objects are represented and accessed because it is defined at the level of the language environment. The reflective approach allows the programmer to change this environment and to influence its behavior. Such an approach can be viewed as an intermediate between middleware and programming languages because on one hand the programming language (reflective) environment is separated from the language itself like in middleware approaches. On the other hand, the programming language has special constructs for influencing and changing the environment where it will run.

In the approaches based on programming languages an environment is created within the language itself and using this very language. In other words, the program is responsible for creating and maintaining its own run-time environment. The functionality, which is normally implemented in some standard middleware, is now an integral part of the program written in the same programming language as the rest of the system.

One wide-spread technique to automating intermediate RA functions consists in using static or dynamic proxies [10]. Proxy is a special class that emulates an interface of the corresponding target class but inserts some intermediate functionality. These intermediate functions of the proxy class are called before target methods and hence they effectively intercept all target object method invocations. The trick here consists in using proxy class instead of the target class. Thus it is not a real interception but rather a normal sequence of method calls. In other words, in the source context a reference to the proxy instance is created and hence its methods are called when it is used. Then it is the task of the proxy to decide what to do if some its method has been called. Normally, after some processing the corresponding target method is called. One disadvantage of this approach is that it requires significant manual support and is not very general. It is more a special technique or programming pattern rather than a programming paradigm. Here are other disadvantages of this approach:

- If a target class changes we need to manually change its proxy class.
- For each target class we need to develop its own proxy while in many cases proxy functions are rather general and can be used by many target classes.
- It is difficult to impose behaviour in a nested manner (creating a proxy for proxy).
- It is difficult to develop custom references which are stored by value instead of native references.

An interesting solution to the problem of developing custom references and intermediate behaviour consists in using smart pointers in C++ [11]. However, it is also a specific technique rather than a general programming approach. A more general solution consists in using the mechanism of annotations. The idea of this approach (called attribute-oriented programming) consists in marking places in code where some intervention is needed by special tags. Other related approaches that can be used to automate intermediate RA functionality are mixins [12,13], subject oriented programming [14] and multidimensional separation of concerns [15]. All these methods allow the programmer to specify how behavioural granules (concerns) have to be distributed throughout the system.

Probably the most interesting approach to solving the problem of separation of concerns is aspect-oriented programming (AOP) [16]. Aspects describe intermediate functionality (and data) injected into the points in the program which are specified by means of regular expression. Thus aspect can be viewed as a special programming construct that modularize intermediate functionality. An important property of this approach is that aspects know explicitly the points where the intermediate functions will be injected while the target classes do not know what other code will modify their behaviour (Fig. 6). Such a structure of relationships between the module with the code to be injected and the modules where it has to be injected can be viewed as declaring in an aspect all the target classes (the target classes being unaware of this aspect). In this sense our approach is characterized by the opposite direction of this relationship (see DG5). Namely, the module with the code to be injected is unaware of the points where it will be used (the target modules). These are the target modules that declare the modifications they need.

6 Conclusions

In the paper we introduced a new programming construct called concept. Concept is defined as a pair of one object class and one reference class having their own fields and methods (possibly with the same name). Concepts are organized into a hierarchy using inclusion relation with the main purpose to specify how objects have to be represented and accessed. The main idea is that an object is represented by its parent reference which replaces a system default reference. An approach to programming based on this new construct is called concept-oriented programming. This approach assumes that a system consists of two

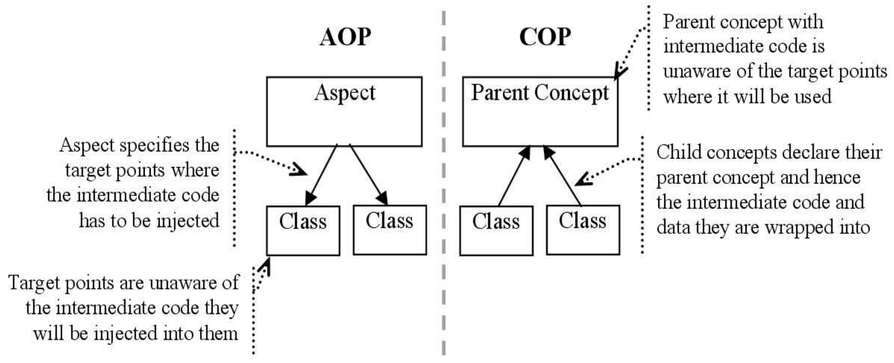

AOP
COP
Aspect
Parent Concept
Class
Class
Class
Class
Aspect specifies the target points where the intermediate code has to be injected
Target points are unaware of the intermediate code they will be injected into them
Parent concept with intermediate code is unaware of the target points where it will be used
Child concepts declare their parent concept and hence the intermediate code and data they are wrapped into

Figure 6. Aspect-oriented programming vs. concept-oriented programming

types of functionality: target BMs and intermediate RA functionality. Accordingly, it is important to be able to implement both types as an integral part of one program using one programming language. This new approach to programming can be applied to very different complex problems such as access control and interception, security and object protection, persistence, debugging, tracing and logging, memory and life-cycle management, containers, remote objects, distributed computing, protocol stacks and many others.

References

[1] Savinov, A. *Principles of the Concept-Oriented Database Model*, Institute of Mathematics and Informatics, Academy of Sciences of Moldova, Technical Report, 54pp., November 2004.

[2] Savinov, A. Hierarchical Multidimensional Modelling in the Concept-Oriented Data Model, Proc. the 3rd international conference on Concept Lattices and Their Applications (CLA'05), Olomouc, Czech Republic, September 7-9, 2005, 123–134.

[3] Savinov, A. Grouping and Aggregation in the Concept-Oriented Data Model, ACM Symposium on Applied Computing (SAC 2006), April 23-27, 2006, Dijon, France (accepted).

[4] Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, 1976.

[5] Roman, E., Sriganesh, R.P., Brose, G. Mastering Enterprise Java Beans. Wiley; 3 edition.

[6] Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/

[7] Cazzola, W., Ancona, M. mChaRM: a Reflective Middleware for Communication-Based Reflection. Technical Report DISI-TR-00-09, DISI, Universita degli Studi di Genova, May 2000. 29 pages.

[8] Kiczales, G., Rivieres, J., Bobrow, D.G. The Art of the Metaobject Protocol. MIT Press, Cambridge, 1991.

[9] Kiczales, G., Ashley, J.M., Rodriguez, L., Vahdat, A., Bobrow, D.G. Metaobject protocols: Why we want them and what else they can do. In: Paepcke, A. (ed.) *Object-Oriented Programming: The CLOS Perspective*, 101–118, The MIT Press, Cambridge, MA, 1993.

[10] Blosser, J. Explore the Dynamic Proxy API, *Java World*, November 2000. http://developer.java.sun.com/developer/technicalArticles/ DataTypes/proxy

[11] Stroustrup B. *The C++ Programming Language*, Second Edition, Addison Wesley, 1991.

[12] Smaragdakis, Y., Batory, D. Implementing layered designs with mixin-layers. *Proc.ECOOP'98*, 550–570, 1998.

[13] Bracha, G., Cook, W. Mixin-based inheritance. *Proc. OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, 25(10), 303–311, 1990.

[14] Subject-Oriented Programming, http://www.research.ibm.com/sop

[15] Multi-Dimensional Separation of Concerns, http://www.research.ibm.com/hyperspace/MDSOC.htm

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming, *Proc. ECOOP'97*, LNCS 1241, 220–242, Jyvaskyla, Finalnd, 1997.

Alexandr Savinov,
Institute of Mathematics and Informatics,
Academy of Sciences of Moldova
str. Academiei 5,
MD-2028 Chisinau, Moldova
E–mail: *savinov@conceptoriented.com*
Home page: *http : //conceptoriented.com/savinov*

Received December 21, 2005

Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

CGEM for Moldova with Factor Markets and Intermediate Inputs

Elvira Naval

Abstract

In this article the attempt to examine the effects of tax policy in the framework of two-sector model with factor markets and intermediate goods is considered. In this connection the difference between intermediate and final goods is introduced. Model was extended to introduce the imported and domestic intermediate goods and production functions to produce domestic goods and exports.

The basic model refers to one country with two producing sectors and three goods. Two commodities that the country produces are an export good, which is sold to foreigners and is not demanded at home, and a domestic good, which is only sold domestically. The third good is an import, which is not produced domestically. There is one consumer who receives all income. The country is small in world markets, facing fixed world prices for exports and imports. Three economic agents operate in the model: a producer, a household, and the rest of the world.

Adjustments in the real exchange rate in response to exogenous shocks to the economy and three price-wedge government policy instruments: an import tariff, an export subsidy, and an indirect tax on domestic sales are included. The single household saves a fraction of its income. Real government expenditure is assumed fixed and the government deficit or surplus is subtracted or added to aggregate savings. The balance of trade is assumed to represent foreign savings.

In order to estimate efficiency effects of tax policy, it is very important to distinguish between intermediate and final goods. In this scope the Calculated General Equilibrium Model (CGEM) is extended to introduce the imported and domestic intermediate goods and also production functions that use primary factors of production (capital and labor) to produce domestic and exports goods.

The model contains most features of the basic model, although for simplicity the savings and investment have been eliminated, and the balance of trade is assumed fixed. The separate production functions for exports and domestic goods with associated demand functions for capital, labor, and intermediate goods as inputs are introduced. Equilibrium conditions are included for each factor that induces an assumption of full employment of fixed aggregate stocks of capital and labor. There is the distinction of two types of imports (each with its own tariff rate): consumption imports, which together with domestic output provide the composite good, and intermediate imports, which in combination with domestic output produce composite intermediate goods required to produce domestic and exported goods. The domestically produced and imported intermediate goods, as with consumption, are assumed to be imperfect substitutes, so the demand for each depends on its relative price and the elasticity of substitution. It is assumed that all tax revenue is rebated to consumers in a lump-sum fashion, so the government spending appears as a component of total income. And the foreign capital inflow is also added to private income. Now this model will be used to define the optimal structure of the tariff rates.

Further the equations of the CGEM with factor markets and intermediate inputs are presented. In the case of Republic of Moldova there were examined: production functions for exported and domestically fabricated goods which use primary factors of production (capital and labor).

Presented model was used for the Moldova economy which is highly depended on imported consumption and intermediate goods in order to evaluate tariffs policy when composite consumption Q^S is maximized.

Let's describe briefly all equations of the model. First, two equations (eq.1-2) are the production functions for exports and domestic goods. Equations (7-12) are the associated demand functions for capital, labor and intermediate goods as inputs. Equilibrium conditions are included for each factor (eq. 26-28) supposing full employment of fixed aggregate stocks of capital and labor.

Flows

$$E = K_e^{1-\alpha_e} L_e^{\alpha_e} \tag{1}$$

$$D = K_d^{1-\alpha_d} L_d^{\alpha_d} \tag{2}$$

$$Q^S = Q(M_q, D_q) \tag{3}$$

$$N = N(M_n, D_n) \tag{4}$$

$$M_q/D_q = f_1(P_q^m, P^d) \tag{5}$$

$$M_n/D_n = f_2(P_n^m, P^d) \tag{6}$$

$$N_e = a_e \cdot E \tag{7}$$

$$N_d = a_d \cdot D \tag{8}$$

$$W_K = \partial E/\partial K_e \cdot P_e^v \tag{9}$$

$$W_K = \partial D/\partial K_d \cdot P_d^v \tag{10}$$

$$W_L = \partial E/\partial L_e \cdot P_e^v \tag{11}$$

$$W_L = \partial D/\partial L_d \cdot P_d^v \tag{12}$$

$$Y = W_K \cdot \overline{K} + W_L \cdot \overline{L} + R \cdot \overline{B} + G \tag{13}$$

$$Q^D = Y/P^q \tag{14}$$

Prices

$$P^e = (1 + t^e) \cdot R \cdot P_w^e \tag{16}$$

$$P^q = (P_q^m M_q + P^d D_q)/Q^S \tag{17}$$

$$P^n = (P_n^m M_n + P^d D_n)/N \tag{18}$$

$$P_q^m = (1 + t_q^m) \cdot R \cdot P_{w_q}^m \tag{19}$$

$$P_n^m = (1 + t_n^m) \cdot R \cdot P_{w_n}^m \tag{20}$$

$$P^t = (1 + t^d) \cdot P^d \tag{21}$$

$$P_e^v = P^e - a_e \cdot P^n \tag{22}$$

$$P_d^v = P^d - a_d \cdot P^n \tag{23}$$

Equilibrium Conditions

$$D_n + D_q - D = 0 \tag{24}$$

$$Q^D - Q^S = 0 \tag{25}$$

$$N_e - N_d - N = 0 \tag{26}$$

$$K_e + K_d - \overline{K} = 0 \tag{27}$$

$$L_e + L_d - \overline{L} = 0 \tag{28}$$

$$P_{w_q}^m \cdot M_q + P_{w_n}^m \cdot M_n - P_w^e \cdot E = \overline{B} \tag{29}$$

$$T - G = 0 \tag{30}$$

$$R \equiv 1 \tag{31}$$

Identities

$$P_q \cdot Q^Q \equiv Y$$

$$P_e^v \cdot E \equiv W_K \cdot K_e + W_L \cdot L_e$$

$$P_d^v \equiv W_K \cdot K_d + W_L \cdot L_d$$

$$P^q \cdot Q^s \equiv P_q^m \cdot M_q + P^d \cdot D_q$$

$$P^n \cdot N \equiv P_n^m \cdot M_n + P^d \cdot D_n$$

Table 1: **Endogenous Variables**

E	Export good	D	Supply of domestic good
N_e	Interm. demand by E	N_d	Interm. demand by D
N	Supply of comp. interm. input	Q^S	Supply of comp. final good
Q^D	Demand for comp. final good	D_q	D used in consumption
D_n	D used in interm. inputs	K_e	Capital used to produce E
L_e	Labor used to produce E	K_d	Capital used to produce D
L_d	Labor used to produce D	M_q	Imports used in consum.
M_n	Imports used in N demand	P^e	Export price
P^d	Domestic producer price	P^t	Domestic consumer price
P_q^m	Price of consumption imports	P_n^m	Price of interm. imports
P_e^v	Value added price of exports	P_d^v	Value added price of D
P^n	Price of interm. input	P^q	Price of composite good
W_K	Return to capital	W_L	Wage rate
R	Exchange rate	Y	Total income
T	Tax revenue	G	Government transfers

Table 2: **Exogenous Variables**

$\overline{L}$	Total labor supply	$\overline{K}$	Total capital stock
$\overline{T}$	Total tax accumulation	$\overline{B}$	Balance of trade
P_w^e	World price of E	$P_{w_q}^m$	World price of M_q
$P_{w_n}^m$	World price of M_n	t^e	Export subsidy rate
t_q^m	Tariff on M_q	t_n^m	Tariff on M_n
t^d	Indirect tax rate		

There are two types of imports, each with its own tariff rate. Consumption imports (M_q) which in combination with domestic output (D_q) produce composite good (Q^S). Intermediate imports (M_n) in combination with domestic output (D_n) produce composite intermediate goods (N) required to fabricate domestic and exported goods. The domestically produced and imported intermediate goods (N) as well as consumption are assumed to be imperfect substitutes, so the demand for each depends on its relative price and the elasticity of substitution (eq. 6).

The production structure may be represented as follows. Real value added and intermediate inputs are combined in fixed proportions to produce output. There are two CES functions: one for labor and capital to produce real value added and one for imported and domestic intermediates to produce the composite intermediate input. Fixed coefficients are used for the demand for composite intermediate input and allow to define value added price for each sector.

The year 2004 was selected as base year, and all macroeconomic indicators are presented both in real and nominal prices. All prices are equal to unity in this base. Using base year data and known values for the elasticity of substitution between imports and domestic goods (σ) equal to 2 for consumer goods and 0.5 for intermediate goods there are calculated the analytical expressions for export (E), demand for domestic goods (D), supply of composite goods (Q^S), volume of intermediate goods (N), ratios between imported consumption goods and domestic consumption good and ratio between imported intermediate inputs and domestic inputs. So, for six functions enumerated earlier the explicit analytical expressions with constant coefficients are obtained. The obtained expressions for these functions are annexed here:

$$E = K_e^{0.702} \cdot L_e^{0,398}, \quad D = K_d^{0.66} \cdot L_d^{0,4545},$$

$$Q^S = 1,136 \cdot (0,5148 \cdot M_q^{-0,5} + (1 - 0,5418) \cdot D_q^{-0,5})^{-1/0,5},$$

$$N = 2,001 \cdot (0,9854 \cdot M_{nq}^1 + (1 - 0,9854) \cdot D_n^1)^1,$$

$$M_q/D_q = (0,5148 \cdot P_q^m/((1 - 0,5418) \cdot P^d)^2,$$

$$M_n/D_n = (0,9854 \cdot P_n^m/((1 - 0,9854) \cdot P^d)^{0,5}.$$

In order to evaluate efficiency resulting from taxation just as in [1], it is assumed that all tax revenue is returned to consumers as a unique sum, so that government spending appears as a component of total income in (eq.13). Finally, foreign capital inflow is also added to private income.

Further, this model is explored for answering the question about optimality of the uniform tariff rates. It is well known view that differences among tariff rates create a distortion in the economy. Different

Table 3: **Optimal tariffs with fixed indirect tax**

%	BV	15%	10%	5%	Opt.	-5%	-10%	-15%
t^d		13,5	12,9	12,3	11,7	11,1	10,5	10,0
Optimum tariff rates								
t_m^q		13,5	14,7	17,0	18,0	19,7	23,5	24,0
t_m^n		28,4	25,9	22,6	18,0	15,6	8,8	9,2
Opt. quant.(Ratio(%) to base value)								
E	64,0	99,20	99,19	99,17	100,0	99,12	99,10	99,08
D	51,0	101,31	101,33	101,36	100,0	101,40	101,42	101,44
M_q	52,0	99,10	99,08	99,06	100,0	99,01	98,99	98,97
M_n	28,0	100,31	100,31	100,31	100,0	100,31	100,31	100,31
Q	100,0	100,77	98,39	96,28	100,0	96,28	98,39	100,77

sectors tariff rates imply that the relative domestic prices of two traded goods are not equal to their relative world prices. If world prices are viewed as the appropriate "shadow prices" of this traded good, a varied tariff structure represents a distortion.

As stabled in [2] there are other distorting taxes in the economy, then the shadow prices of this traded good in this environment may not equal to world prices. In particular, if the domestic indirect tax structure is not optimal, the optimal tariff structure will generally not be uniform. Starting with various assumptions about the level of domestic indirect taxes the optimal patterns of tariffs will be determined. The numerical solution of the formulated optimization problem using statistical data for the year of 2004 will be obtained because the introducing of the intermediate goods complicates the model so much that the model cannot be solved analytically.

Base year data for numerical application are presented in Table 4. Here all data are calculated in % to GDP which is equal to 100%, and with the exported national economy equal to 64%. The export sector is capital-intensive and uses domestic and imported intermediates. The domestic sector (D) also uses domestic and imported intermediates.

Table 4: **SAM for Moldova Numerical Model**

Moldova	**Expenditures**							
	E	D	Q	N	L	K	C	Rw
Export (E)								64
Domestic (D)			48	3				
Final Good (Q)							100	
Interm. Good (N)	15	16						
Labor (L)	10	8						
Capital(K)	39	43						
Consumer					18	82		
Rest of World			52	28				
Total	64	67	100	31	18	82	100	64

The balance of trade is equal to 17% and the single consumer thus demanded 100% units of composite consumer good Q. Since all prices and wages are equal to one, the SAM (Social Accounting Matrix, see Table 4) indicates real as well as nominal magnitude. The value added production functions are assumed to be Cobb-Douglas.

It is supposed that government receives revenue from tariffs on final and intermediate goods and on indirect tax on domestic sales. Export subsidy is equal to zero. Government requires total tax revenue of 25%. The base year data given in the Table 4 represent a unique solution of the model with all tax rates set to zero. In the optimal tax experiments tax rates are redefined as variables offering freedom of choice. Optimal taxes are obtained by maximizing Q, which satisfies the equations of the model in examination plus the government revenue constraint. Alternative scenario is obtained by fixing one tax, the indirect tax on domestic goods (T^d), and solving for optimal level for remaining tax rates.

The results of calculus are given in Table 2. The optimal pattern of tariffs is uniform only in this case when all tariff rates including the indirect tax on domestic goods is also set optimally. When the indirect tax on domestic goods is set below its optimal value then optimal tariff structure consists in higher tariff on the final goods than on the intermediate goods. When the indirect tax is above its optimal, the opposite is true. In this case when domestic indirect taxes are too low rather than too high the appropriate policy rule is that tariff rates on imported intermediate goods should be lower than the rates for imported consumer goods.

So it is no reason to move toward equal rates by raising the lowest tariffs and lowering the highest ones. Yet in our country tariffs on intermediate goods are lower than those on final goods so in this circumstances moving toward equal rates would lower welfare.

Along with a highly variable tariff structure the second scenario shows smaller variations in real variables. But in this case aggregate welfare demonstrates small changes across both scenarios. This result is consistent with results from a large number of empirical studies.

Given that, the model is solved as a nonlinear programming problem using SOLVER package from Excel. The solution generates dual or shadow prices for all constrains. The solution value for the shadow price on the government revenue constraint (eq. 30) directly measures the welfare cost of raising an additional unit of tax revenue (T). In Table 3, there are presented these shadow prices for the cases in which the indirect tax rate is set below its constrained optimal value. Results for two alternative models are calculated. One model is solved for optimal changed tariffs - as in table two. And the other model in which tariffs are constrained to be the same for both goods. In this case, the single tariff rate is uniquely determined by the government revenue constraint and the fixed indirect tax rate. There are no policy degree of freedom.

Let's comment the results obtained in Table 3. The results of calculus show that in both scenario there are observed the diminishing of the marginal welfare cost when additional government revenue is received from increasing tax revenue. So in the case of Moldova there exist reserves in rising indirect tax rate. Yet in the case of equal tariffs there are more possibilities in reducing welfare cost by absolute value, than in the case of differential tariffs.

These results, as mentioned in [1], are only suggestive because of the stylized nature of the model, but they create some theoretical under-

Table 5: **Welfare Cost of increasing Tax revenue**

	Marginal welfare cost of increasing tax revenue as a % of additional revenue		
Indirect tax rate (%)	Sc.I (equal tariffs)	Sc.II (differential tariffs)	Ratio (%) of Sc.I to Sc.II
11,7 (optimal)	0,0	0,0	
11,1 (-5%)	-6,42	-6,0	106,5
10,5 (-10%)	-6,54	-5,3	123,4
9,9 (-15%)	-6,67	-5,2	127,8

pinning for the common policy rule that countries should unify their tariff structure. And the results from this simple two sector model appear vigorous in comparison with larger applied models [2].

References

[1] Sh. Devarajan, J.D Lewis and Sh. Robinson, *Policy lessons from trade-focused, two-sector models.* Journal of Policy Modeling, 1990, 12(4), pp.625–657.

[2] Dahl H., Devarajan S., and Wijnbergen S., *Revenue-Neutral Tariff Reform: Theory and an Application to Camerun. Discussion Paper No. 1986-25. Country Policy Department. The World Bank.* 1986(May).

[3] Harberger A., *Reflections on Uniform Taxation.* 44-th Congress of the International Institute of Public Finance, Istambul, 1988, (August).

[4] Pyatt G. and Round J.T. Eds., *Social Accounting Matrices: A Basis for Planning.* Washington, DC: World Bank. 1985. 1970, p. 387–392 (in Russian).

Elvira Naval,
Received December 3, 2005

Institute of Mathematics and Computer Sciences,
Academy of Sciences, Moldova
5, Academiei Str., Kishinev,
MD–28, Moldova
E–mail: *nvelvira@math.md*

Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

An Interactive Web-based Environment using Human Companion

Tahar Bouhadada   Mohamed-Tayeb Laskri

Abstract

This paper describes the architecture of an Interactive Learning Environment (ILE) on internet using companions, one of which is a human and geographically distant from the learning site. The achieved system rests on a three-tier customer/server architecture (customer, web server, data and applications server) where human and software actors can communicate via the internet and use the DTL learning strategy. It contains five main actors: a tutor actor in charge to guide the learner; a system actor whose role is to manage and to control the accesses to the system; a teacher actor in charge of the management and the updating of the different bases; a learner actor who represents the main actor of the system for whom is dedicated the teaching. Also, a learning companion actor whose role can be sometimes as an assistant, and other times as a troublemaker.

Keywords: Interactive learning environment, LCS, DTL strategy, companion, distance learning, troublemaker.

1 Introduction

The distant teaching pedagogy differs from the teaching in a classroom. Indeed, the absence of the teacher influences the incentive and the concentration of the learner, what encourages the isolation feeling and so, moves him away of the stimulating context as in a real classroom.

In a distant learning context, the pedagogical triangle [1],[2] must take into account two elements that, in this case, take a particular importance: the group and the mediation context (Figure 1).

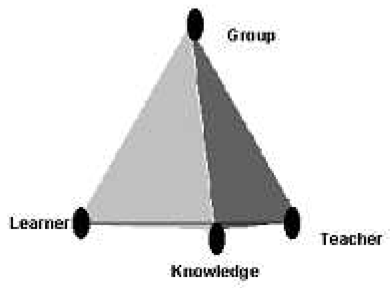


Figure 1. The pedagogical triangle

The group is an instituted set of learners and teachers in interaction, sharing some common objectives. The introduction of the group element puts in evidence the social character of the knowledge construction [3]. Indeed, the group constitutes a psychological support factor [4]. The mediation context constitutes the material or a virtual environment in which occurs the interactions.

In the present work, we describe an interactive learning environment (ILE) in a distant-teaching context with learning companions and using Internet as the environment of communication and interaction. The achieved system is a software framework dedicated to the learning of the relational databases whose customer/server architecture is based on multi-agents approach. For the communication between the learners, we used tools, more powerful, as the electronic mailing, the forums, that have already been integrated in many distant-training frameworks as support for collective learning activities [5],[6].

Several works showed that in a learning environment, the social interaction and the cooperative work in a community of learners has an influence on the intern structure of the learner's cognitive form [7], [8].

Our gait is based on the principle that the learning enriched also itself through the exchanges, the confrontations, the negotiations, the competition and the interactions between persons.

Indeed, in the learning psychosocial model, learner doesn't learn

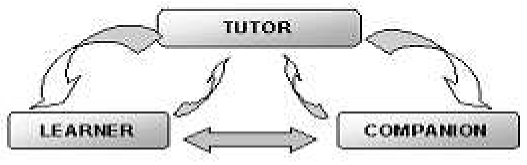


Figure 2. General architecture of an LCS

alone, but with confronting his thought and his actions to the material and social reality. The social psychology of the cognitive development opposes to epistemic individualism and substitutes to the bipolar centering *ego-object* of the cognitive psychology a tripolar relation *ego-alter-object*. According to this approach, the interactions with others play an essential role in trainings. In particular, they are going to permit to disapprove the initial conceptions and to create some favorable dissonances for the construction of a new knowledge. It is the socio-cognitive conflict mechanism [9],[10].

2 The Learning Systems Using Companions

The learning systems using companion rest on a software companion where the behavior and reactions are entirely simulated and often, follow a linear and recurrent structure. Several systems using software companions showed the recurrence in a learning situations of the behavior of the companion in a cooperative and collaborative environment [11],[12],[13],[14].

The structure of a Learning Companion System (LCS) described by Chan [12] implies three basic actors (Figure 2):

A tutor actor (software teacher) whose role consists fundamentally to provide matter to teach, to offer examples, indications, and commentaries to the learner and the companion. A learning companion actor whose objective is to stimulate the collaboration with the learner through the competition. This actor can have several roles; he can play the role of an assistant to whom the learner can ask for help and assistance, sometimes as a competitor. In other systems, he can be a troublemaker. The third actor, a learner, who is a committed and active person in an acquirement process or a knowledge perfection.

The approach adopted in the present work goes in the setting of the CSCL context (Computer Supported Collaborative Learning) that constitutes an evolution from a distant interactive environment to environments supporting the collaboration to enrich the collective and social construction of the knowledge [15],[16],[17].

In our system, we introduce three learning companions: A human companion and two software companions.

- *The human companion:* He is a learner who follows his training in the same title and at the same time as the system learner and to whom he can bring assistance. This companion can be any other learner connected on-line on the network and that the learner can solicit him. In case of absence of a human companion, the learner can solicit the machine companion which is created for such situations.

- *The machine companion:* He takes the role of an assistant, and other time, the role of a troublemaker, giving some erroneous answers voluntarily to put the learner in a doubt situation and so, to test his confidence and his convictions.

3 The DTL Strategy

A typical learning session that uses the Double Test Learning (DTL) strategy [11], [13] starts with a Pre-Test phase in which an initial learner model is created. In the second phase (Learning phase), the system dispenses the teaching and the co-learners benefit of the same training that the human learner, so, at the end of this phase, the three learners have the same level of knowledge.

In the third phase (Post-Test1), the tutor tests the co-learners. The human learner will be in the place of an active observer. He will follow the questions/answers sequence between the tutor and the co-learners.

The learner has in his possession a notebook on which he can mention all useful observation. At anytime that the co-learners give the solution of the given problem, the tutor values their answers. If their answers are incorrect and that of the human learner is correct, this last must justify and explain his answer to the co-learners. When the co-learners finish the Post-Test1 phase, the tutor turns then toward the human learner and the last phase (Post-Test2) begins. Here the learner's notebook is withdrawn, and therefore, he has access to his memory only and to the knowledge that he has acquired lately through the co-learners answers. At the end of this phase, the tutor values his answers in order to attribute to him a score and, determine his new profile.

4 The Society of Actors

An actor represents a set of coherent roles played by an external entities (human user, device system), that interact directly with the studied system.

Our system includes five main actors, implying human actors and software actors:

- *The system actor:* It's a software actor whose role is the management of the accesses to the system and the control of the registration or the suppression of users (learners or teachers).

- *The tutor actor:* It's a software actor; its role is to assure the pedagogical progression of the learner during his training. It puts to his disposition the courses, explanatory examples and exercises with solutions and arguments. He has also the task of the evaluation during the test phases (Pre-Test, Post-Test1, Post-Test2).

- *The teacher actor:* He is a human actor who has in charge to update courses and exercises. He is responsible of the choice and the definition of the pedagogical strategy to be adopted. He can also consult any registered learner's profiles in the system.

- *The learner actor:* He is a human actor, he represents the main actor for whom the teaching is dedicated.
- *The companion actor:* It can be human or software :
  - *The human companion:* He is a learner connected on-line on the system whose learning is not the principal objective for the system. His role is essentially to assist the learner during the Post-Test1 phase. His presence is not certain. He can be solicited by the learner at any moment.
  - *The machine companion:* This companion is solicited in case of absence of a human companion on the network. Its role is to simulate the human behavior. The system introduces two software companions whose behaviors are simulated; one of them plays the role of an assistant and the other one a troublemaker by introducing disruptions during the Post-Test1 phase in the goal to test the insurance and the conviction of the learner. The answers provided by the troublemaker companion are, in most time, incorrect voluntarily.

5 The Software Architecture

The DB-Tutor++ system has been conceived according to the three levels customer/server architecture (Architecture 3-tiers): a customer level, a data and applications server level and a web server level (Figure 3).

- *Customer level:* It represents the different services asked by a customer, learner or teacher.
- *Web server level:* It constitutes the interface between the customer and the data server while transmitting the customer's request toward the data server, and the achieved service by this last toward the customer.
- *Data and applications level:* It represents the different services of data management offered to the customers (teachers, learners).

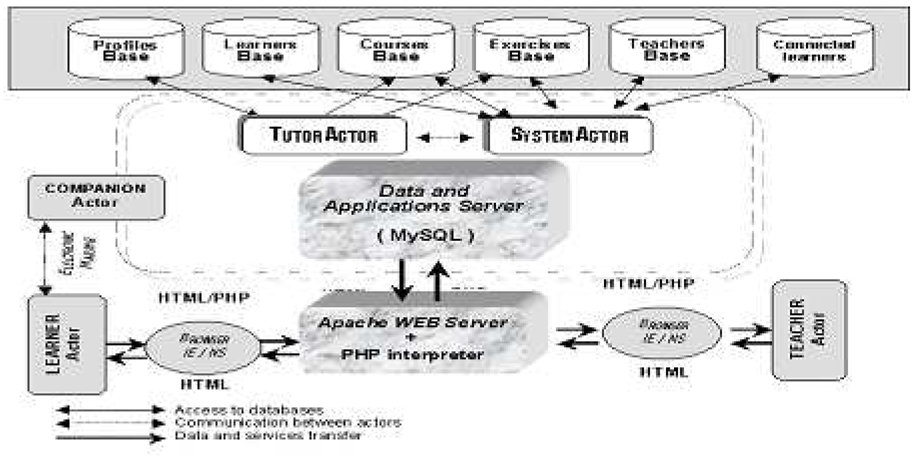


Figure 3. General architecture of DB-Tutor++

In our data server, we distinguish two main actors that achieve these services according to the customer's request: the system actor and the tutor actor. These two actors use a whole of databases for managing their services:

- A learners' base that contains the personal information about the learners.
- A teachers' base that contains information concerning the teachers.
- A profiles' base that contains the historic of the different learner's behavior during the different sessions.
- A courses' base whose structure is hypertextual that contains the whole of courses, structured in levels.
- An exercises' base that contains the list of exercises for every test phase and distributed in different levels.

- A connected learners' base that contains the list of learners online on the system.

6 Teaching material

Databases are dispensed in all training programs in computing science.

Particularly, the relational databases constitute the most merchandised database systems and the most used in the enterprise's computer systems. The courses base of the DB-Tutor++ system is organized in levels. A level represents a state of knowledge acquired by the learner. A level contents concepts and meta-concepts. A concept is a knowledge element. A meta-concept is composed of a whole of concepts. A course is constructed about meta-concept, and a whole of examples. The passage from a level to a superior level requires the acquisition of the concepts introduced in the lower levels. The courses are organized as a hypertextual form.

In its present version, the system's courses base includes 54 meta-concepts, and 218 concepts distributed in 5 levels, numbered from 1 to 5 (Table 1).

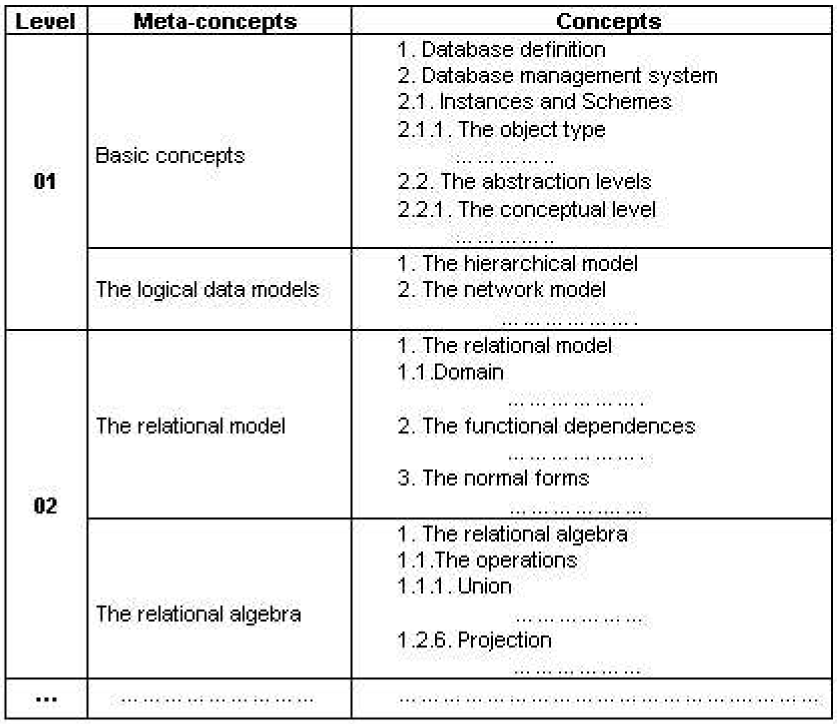
7 The Evaluation

The evaluation is a process that consists in determining or to assign a level to the learner in a learning session. For the learner evaluation, we defined two categories of Multiple Choices Questions (MCQ). The first category includes simple questions and the second, questions with proof.

For simple questions, the learner must introduce the number of his answer. For this kind of question two (02) tokens are assigned for a correct answer, and zero (00) for an incorrect answer.

For questions with proof, the learner must answer by *yes* or by *no*, and his answer must be justified by a proof.

- If the answer is correct, two (02) tokens are attributed.



Level	Meta-concepts	Concepts
01	Basic concepts	1. Database definition 2. Database management system 2.1. Instances and Schemes 2.1.1. The object type … … … … .. 2.2. The abstraction levels 2.2.1. The conceptual level … … … … ..
	The logical data models	1. The hierarchical model 2. The network model … … … … … … .
02	The relational model	1. The relational model 1.1.Domain … … … … … … . 2. The functional dependences … … … … … … . 3. The normal forms … … … … … …
	The relational algebra	1. The relational algebra 1.1.The operations 1.1.1. Union … … … … … … 1.2.6. Projection … … … … … …
…	… … … … … … … … … …	… …

Table 1. Description of the contents of the levels

- If the proof is correct, the score will be increased of two (02) other tokens.
- In the case where the learner does give an incorrect answer, no token will be attributed (even if the proof is correct).

7.1 Acquisition of a Level

To every i phase a general score *(ScoreG)* equal to the sum of tokens attributed to the n Q questions of the phase is associated:

$$ScoreG_{phase_i} = (\sum_{k=1}^{n} tokensQ_k), \qquad (1)$$

where:

n: is the number of questions.

i : Pre-Test phase, Post-Test1 phase, Post-Test2 phase.

The average score *(ScoreM)* for a learner in a session is calculated as follows:

$$ScoreM = (\sum_{i=1}^{n} ScoreG_{phase_i})/2. \tag{2}$$

The final score ($Score_{Final}$) gotten by a learner is equal to the sum of acquired tokens during every phase:

$$Score_{Final} = \sum_{i=1}^{3} Score_{phase_i}. \tag{3}$$

So, for a learner, to reach to the immediately superior level, it is necessary that:

$$Score_{Final} >= ScoreM. \tag{4}$$

For a new registered learner, the assigned level is determined by the score gotten during the Pre-Test phase:

$$ScoreG_{pre-Test} = \sum_{k=1}^{n} TokensQ_k. \tag{5}$$

Questions of the Pre-Test phase concern the immediately lower level.

So, for a learner, to be registered in a level L, it is necessary that:

$$Score_{Pre-Test} >= ScoreM_{Pre-Test}, \tag{6}$$

where $ScoreM_{Pre-Test}$ is the requisite average score for this phase:

$$ScoreM_{Pre-Test} = (ScoreG_{Pre-Test})/2. \tag{7}$$

8 The Implementation

The development of distance learning systems requires languages dedicated to the implementation of applications on Internet network. The realization of an environment according to 3-tier architecture requires navigation, interpretation and communication tools very powerful. DB-Tutor++ has been achieved with a language oriented to customer and a language oriented to server.

The system has been developed on the basis of the APACHE server and uses its PHP interpreter for the interpretation of the different interactions. For the realization of the courses base, we used the XML language, more adapted for the development of hypertext systems. Finally, for the management of the different bases, we opted for MySQL whose performances are especially indicated for this kind of application.

9 Users Scenarios

In order to fear the working and the global dynamic of the system, and more particularly, the interactions between the different actors (human and artificial), we present users scenarios of the application for a learner user, a teacher user and for an administrator user (Figure 4).

Figure 4. Screen "Home Page"

The learner, the teacher or the administrator introduces the username and the password that have been assigned to him at their account creation time. After verification of the identity by the system actor, the interface of the corresponding user (learner, teacher or administrator) is displayed.

9.1 A "Learner" Scenario

- **Connection / disconnection of a learner:** At the connection of a learner, two actors, the companion actor and the trouble-maker actor, are created and enter to the system.

  The tutor actor is informed about his connection, via the system actor, that goes to re-actualize the advancement state with taking into account the profiles base, then, to present the companions to the learner. After this, the learner lunchs the Pre-Test phase, and the other phases (the Learning phase, the Post-Test1 phase, and the Post-Test2 phase) according to the kind of learner.

  For the disconnection, the learner must inform the system actor about his exit so that it frees the occupied resources and companions (Figure 5).

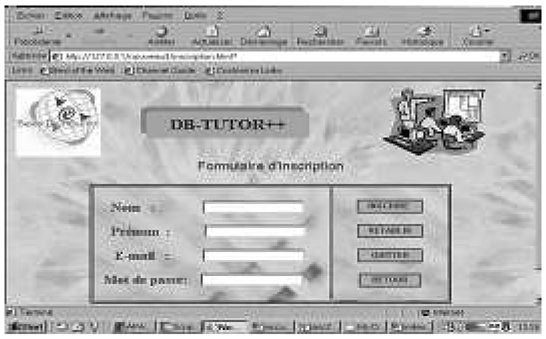
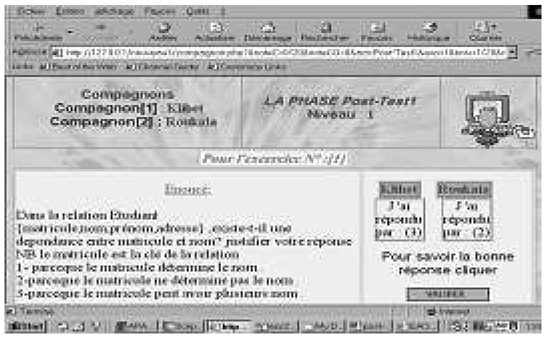


Figure 5. Screen "Connection / disconnection of a learner"

- **Request for a companion:** At the connection, the learner sends to the system actor a request for a learning companion. This one verifies if there exists a human companion connected on-line in the system, in the contrary case, he creates two software companions, which one of both is a troublemaker (Figure 6).

Figure 6. Screen "Request for a companion"

- **The learning:** The learning starts at the end of the Pre-Test phase. The learner signals to the tutor that he is ready to follow the training. The tutor transmits to him courses corresponding to the determined level in the Pre-Test phase
- **Post-Test1 Phase:** As soon as the training session is finished, the learner attends the Post-Test1 phase as an observer. He observes reactions of his companions during the questions/answers sequence proposed by the tutor. He can also take notes and remarks on his notebook (Figure 7).
- **Post-Test2 Phase:** In this phase, the human learner will be tested. At this level, the notebook is not on his possession. He is submitted to a set of questions and exercises to which he must give answers. The tutor recovers answers, value them and assign a score. At the end of this step, the tutor displays the final score of the learner.


Compagnons
Compagnon[1] : Kliber
Compagnon[2] : Roukaia

LA PHASE Post-Test1
Niveau 1

Pour l'exercice N° :[1]

Enoncé:

Dans la relation Etudiant
(matricule,nom,prénom,adresse) ,existe-t-il une
dependance entre matricule et nom? justifier votre réponse
NB le matricule est la clé de la relation
1 - parceque le matricule détermine le nom
2-parceque le matricule ne détermine pas le nom
3-parceque le matricule peut avoir plusieurs nom

Kliber	Roukaia
J'ai répondu par (1)	J'ai répondu par (2)

Pour savoir la bonne réponse cliquer

Figure 7. Screen "Post-Test1 phase"

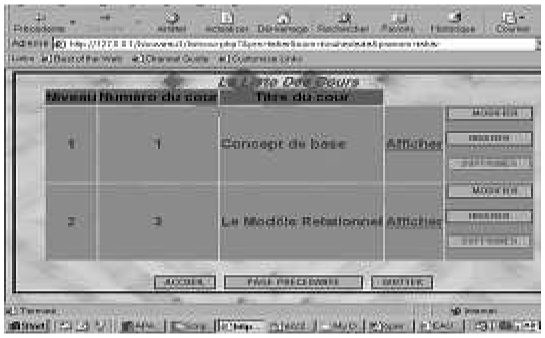
9.2 A "Teacher" Scenario

- **Connection/disconnection of a teacher:** When a teacher connects himself to the system, the system actor asks him for his identification in order to verify his access right. The disconnection is achieved by the teacher on his demand.

- **Courses/exercises updating:** When the teacher wants to add or to withdraw a course or an exercise that he judges useless, or to modify it, the system puts to his disposition a list of the available courses/exercises in the base, then the teacher will select the number of the course or of the exercise to be deleted or to be modified. In the case of a new exercise, the statement must be joined by its solution (Figure 8).

- **The updating of the pedagogical strategies:** The teacher can at any time define or modify the educational rules according to the learner's profile and the previous definite pedagogical objectives.

- **Consultation of learner's profiles:** At any moment, the teacher can consult learner's profiles by a demand to the system. This last displays the list of learners and their individual

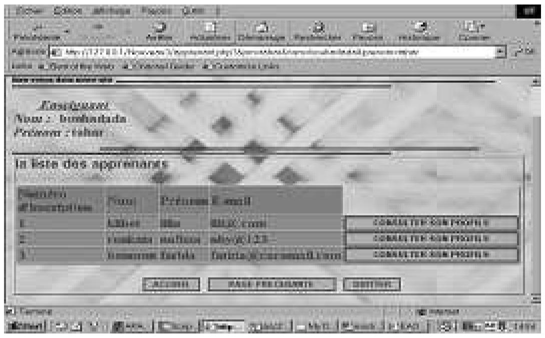


La Liste Des Cours

Niveau	Numero du cour	Titre du cour	
1	1	Concept de base	Afficher
2	3	Le Modèle Relationnel	Afficher

Figure 8. Screen "Courses / Exercises updating"

profiles as well as the historic of their behaviors in the different situations of the learning sessions (Figure 9).



Enseignant
Nom : bouhadada
Prénom : tahar

la liste des apprenants

Numéro d'inscription	Nom	Prénom	E-mail
1	[illegible]	lila	lila@.com
2	[illegible]	sofiane	sbv@123
3	[illegible]	farida	farida@caramail.com

Figure 9. Screen "Consultation of learner's profile"

9.3 An "Administrator" Scenario

We mean by administration, the insertion and the deletion of teacher account or learner account. The creation and the suppression of an account are system procedures that permit to introduce or to suppress users from the system, as well as the updating of the base of the profiles in the case of an inscription of a new learner.

10 Conclusion

We described an interactive learning environment dedicated to teaching the relational databases on Internet. The system DB-Tutor++ that uses the DTL learning strategy, in its new version, implies a community of learners, and human and machine companions.

The system adopts a three-tier customer/server architecture (web server, data and applications server and customer), where human and software actors can communicate through the Internet network.

The system adopts a collaborative pedagogical method that permits a constant solicitation of the learner, a permanent evaluation, a multiplication of paths, and multimedia tools that encourages using a maximum of learning channels implying a community of human and machine actors.

The ambition of the present project is to offer a collaborative learning environment on Internet, what requires complementary pluridisciplinary contributions.

The gaits are undertaken currently to shelter the system on the university web site in order to be able to experiment it with students of the 3rd year of the engineers cycle.

References

[1] Meirieu, P. *Apprendre... Oui, Mais Comment?* ESF, Paris, France. (1989)

[2] Doise, W., Mugny, G. *Social Interaction and the Development of Cognitive Operations.* European Journal of Social Psychology, vol. 5, no. 3, pp.367–383. (1975)

[3] Houssaye, J. *La Pédagogie: Une Encyclopédie pour Aujourd'hui*, ESF, Paris, France.(1993)

[4] Mugny, G., Doise, W. *Socio-Cognitive Conflict and Structure of Individual and Collective Performances.* European Journal of Social Psychology, vol. 8, pp. 181–192.(1978)

[5] Faerber, R. *Apprentissage Collaboratif à Distance: Outils, Méthodes, et Comportement Sociaux.* In Proceedings of 5th Biennale Internationale des Chercheurs et des Praticiens de l'éducation et de la Formation. April. Paris, France.(2000)

[6] Ecoutin, E. *Etude Comparative Technique et Pédagogique des Plates-Formes pour la Formation Ouverte et à Distance.* ORAVEP Report. Ministry of Research (DT/SDTETIC), France (2000).

[7] Chan, T. W., Baskin, A. B. *Studying with Prince: The Computer as a Learning Companion*, in Proceedings of Intelligent Tutoring Systems (ITS'88), June Montréal, Canada.(1988)

[8] Zidane, A., Djoudi, M., Zidat, S., Talhi, S. *CHELIA: Un Environnement Coopératif pour l'apprentissage sur Internet.* In Proceedings of 10ème Colloque Africain sur la Recherche en Informatique (CARI'02), October, Yaounde, Cameroon.(2002)

[9] Bouhadada, T., Laskri, M. T. *DB-TUTOR: Un Système Tuteur Utilisant un Compagnon Perturbateur.* In Proceedings of 10th Colloque Africain sur la Recherche en Informatique, CARI'02, October, Yaounde, Cameroon.(2002)

[10] Cerisier, J. F. *Environnement d'apprentissage Collectif en Réseau. Enseignement et Apprentissage en Réseaux.* CRDP Report, Poitou-Charente, Mars, France.(1999)

[11] Bouthry, A., Chevalier, P., Shaff, J. L. *Choisir une Solution de Téléformation.* France.(2000)

[12] Fahmi, M., Aimeur, E. *RACSY: An Intelligent Tutoring System Based on the Double Test Learning Strategy.* In Proceedings of the 5th Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'98), December, Tunisia.(1992)

[13] Uresti, J.A.R., De Boulay, B. *Expertise, Motivation and Teaching in Learning Companion Systems.* International Journal of Artificial Intelligence in Education, (14), pp.193–231.(2004)

[14] Laperrousaz, C., Teutsch, P. *Un compagnon logiciel capable de dialoguer.* CLAVIE: Compagnon Logiciel d'aide aux Apprenants dans l'enVIronnement croisièrEs. Environnement Informatique pour l 'Apprentissage Humain, EIAH'03, Strasbourg, France.(2003)

[15] Faraco, R. A., Rosatelli, M. C., Gauthier, F. A. O. *Adaptivity in a Learning Companion System.* IEEE International Conference on Advanced Learning Technologies, ICALT'04, pp.151–155.(2004)

[16] Chou, C., Chan, T., Lin, C. *Redefining the Learning Companion: the past, the present and the future of educational agents.* Computers and Education, (40) , pp.255–269.(2003)

[17] Jaillet, A. *Apprentissage à Distance, une Révolution pour les Enseignants.* Louis-Pasteur University, Strasbourg, France,(1999).

T. Bouhadada, M.-T. Laskri

Received December 13, 2005

Research Group on Artificial Intelligence (GRIA/LRI)
University of Annaba
Department of computing
BP:12 Annaba 23000 Algeria
Phone/Fax: +21338872436/+21338872756
E–mail: *bouhadadat@yahoo.fr*; *mtlaskri@wissal.dz*

The XIV Conference on Applied and Industrial Mathematics dedicated to the 60th anniversary of the foundation of the Department of Mathematics and Computer Science of Moldova State University

Chisinau, Republic of Moldova, August, 25–27, 2006

Organizers

- Romanian Society of Applied and Industrial Mathematics – ROMAI;
- Mathematical Society of the Republic of Moldova;
- State University of Moldova;
- Tiraspol State University;
- Institute of Mathematics and Computer Sciences of the Academy of Sciences of the Republic of Moldova;
- Center for Education and Research in Mathematics and Computer Science at Moldova State University.

THE ROMANIAN SOCIETY OF APPLIED AND INDUSTRIAL MATHEMATICS (ROMAI) was founded in 1992. It is a non-profit scientific association, whose members are mathematicians, physicists, engineers, chemists, biologists, economists, and other users of mathematics. ROMAI encourages studies relating mathematics and non-mathematical fields of knowledge.

The principal activities in ROMAI are the annual Conferences on Applied and Industrial Mathematics (CAIMs) coorganised together with an university or/and other institution. So far, ROMAI together with University of Oradea, University of Pitesti, Tiraspol State University, Institute of Mathematics and Computer Sciences from Chisinau and the City House and Local Council of Mioveni-Arges organized 13 editions of CAIM. Three CAIMs were held in Romania & Republic of Moldova and the others only in Romania.

CAIM 2006 Objectives

CAIM 2006 provides a forum for the review of the recent trends in theoretical, numerical, and experimental applied and industrial mathematics as well as in computer sciences.

The Conference is dedicated to the 60^{th} anniversary of the foundation of the Department of Mathematics and Computer Science of Moldova State University.

Conference Sections

- Algebra, mathematical logic, topology.
- Ordinary differential equations and finite dimensional dynamical systems.
- Functional analysis and partial differential equations.
- Analytical and numerical methods and applications. Industrial mathematics.
- Theoretical and applied computer sciences.
- Education.

Contacts

- Prof. Mitrofan CIOBAN (Chairman of the Organizing Committee, Chisinau): *mmchoban@mail.md*
- Anca-Veronica ION (Member of the Scientific Committee, Pitesti): *anca_veronica_ion@yahoo.com*