

Reconfigurable Hierarchical Finite State Machines

Valery Sklyarov, Iouliia Skliarova
Department of Electronics, Telecommunications and Informatics, IEETA
University of Aveiro, Aveiro, Portugal
skl@det.ua.pt, iouliia@det.ua.pt

Abstract

The paper suggests design methods for reconfigurable hierarchical finite state machines (RHFSM), which possess two following important features: 1) they enable the control algorithm to be divided in modules providing direct support for “divide and conquer” strategy; 2) they can be reconfigured both statically and dynamically. Run-time reconfiguration permits virtual control systems to be constructed, including systems that are more complex than capabilities of available hardware. It is shown that RHFSM can be synthesised from specification in form of hierarchical graph-schemes with the aid of the considered in the paper VHDL templates. The results of experiments show correctness of the proposed methods and their applicability for the design of engineering systems.

Keywords: control systems, hierarchical specification, finite state machines, VHDL templates

1 Introduction

Divide and conquer is a challenging design technique for any engineering system, including robotics and embedded applications [1]. Another demanded feature is reconfigurability whose importance was reported in a number of publications [1-5].

Finite state machines (FSM) are probably the most widely used components in digital systems. For many practical applications it is desirable to provide FSM with virtual capabilities, in general, and modifiability, in particular [1]. The objectives of these are very different. For one kind of applications we might want to supply capabilities similar to those provided for general-purpose processors which support the virtual memory mechanism. In particular, this feature allows a device to be constructed on a microchip that does not have sufficient hardware resources to accommodate all the functionality of the device. For another kind of applications it may be desirable to be able to alter the behaviour depending on external events that cannot be predicted in advance (this is very important for self-reconfigurable robots [2,3] and adaptable systems [4]). In some cases we need to provide sufficient flexibility to allow for changes during the debugging stage [1] (for example, to verify alternative or competitive algorithms), etc.

This paper suggests methods of FSM synthesis (with primary emphasis on reconfiguration) that possess two following very important characteristics: 1) supporting the hierarchy and the strategy “divide and conquer”; and 2) permitting statically and dynamically reconfigurable control systems to be developed.

The main distinctive features of the paper comparing with [5-7] are listed below:

- Revision of known results (namely the modular specification and synthesis of hierarchical FSMs) and explicit definition of design templates supporting the strategy “divide and conquer”;
- Structure of RHFSM and reconfiguration methods;
- Advanced techniques for stack operations in RHFSM.

2 Modular Specification of RHFSM

Modular specification can be presented in the form of hierarchical graph-schemes (HGS) [5]. Figure 1 shows an example. Module Z_0 in figure 1 represents a top-level recursive algorithm for solving different optimisation problems over binary and ternary matrices, such as discovering the minimal cover of a binary matrix [8], Boolean satisfiability [9], graph colouring, etc. Such algorithms are required for many engineering problems, including those appearing in robotics and embedded systems [10].

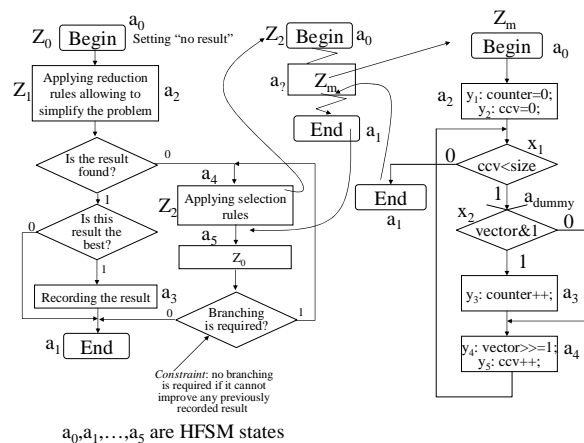


Figure 1: Example of modular specification.

One of the tasks of module Z_2 is executing operations over binary and ternary vectors. Examples of such operations are calculating the maximum number of successive 1s (0s) in a given binary vector, testing for orthogonality between two ternary vectors, comparing two vectors, etc. As an example, module Z_m in figure 1 describes a trivial algorithm for counting the number of ones in a given binary vector.

Note that for numerous optimisation tasks over matrices different algorithms have to be executed. For example, for the covering problem one of the required operations is counting 1s [8] and for the satisfiability problem “counting 1s” is not needed but checking for orthogonality of ternary vectors is required [8]. Thus, a possible approach to provide adaptability of the module Z_0 for solving different problems is to reconfigure the module Z_m .

We will divide the considered task into the following two subtasks: static hardware implementation of the modules in figure 1 (see section 3) and dynamic reconfiguration of the module Z_m (see section 4). Finally, we will show how to design hardware, which joins these subtasks and provides advanced operations for modules’ collaboration.

3 Synthesis of HFSMs

Methods of HFSM synthesis from HGS specification were proposed in [5] and we will demonstrate these methods just on an example. The following VHDL code describes the stacks (called M_stack and FSM_stack [5]) that are used as HFSM memory:

```
process(clock,reset) (1)
begin
  if reset = '1' then -- initialising
  elsif rising_edge(clock) then
    if inc = '1' then
      if -- test for possible errors
      else sp <= sp + 1; -- sp - stack pointer
        FSM_stack(sp+1) <= a0; -- ref1
        FSM_stack(sp) <= NS; -- next state
        M_stack(sp+1) <= NM; -- next module
      end if;
    elsif dec = '1' then sp <= sp - 1; -- ref2
    else FSM_stack(sp) <= NS;
    end if;
  end if;
end process;
```

Here, the signal inc is generated in any rectangular node of HGS, which calls another module; the signal dec is generated in the End node of any module.

The following VHDL fragment describes a template (skeletal code) for combinational circuit of HFSM and the module Z_m from figure 1 is completely specified.

```
process (CM,CS,X) -- current module (CM), state (CS)
begin -- X = {x1,...,xL} - input signals (2)
  case M_stack(sp) is
  when Z1 =>
    case FSM_stack(sp) is -- state transitions
    -- and output generation in the module Z1
```

```
end case;
-- .....
when Zm => -- below the complete code is given
  case FSM_stack(sp) is
  when a0 => Y <= (others => '0');
    inc <= '0'; dec <= '0'; NS <= a2;
  when a1 => Y <= (others => '0');
    inc <= '0'; NS <= a1;
    if sp > 0 then dec <= '1';
    else dec <= '0';
  when a2 => Y <= "11000"; -- y1 and y2
    dec <= '0'; inc <= '0';
    if x1='0' then NS <= a1;
    elsif x2='0' then NS <= a4;
    else NS <= a3;
    end if;
  when a3 => Y <= "00100"; -- y3
    dec <= '0'; inc <= '0';
    NS <= a4;
  when a4 => Y <= "00011"; -- y4 and y5
    dec <= '0'; inc <= '0';
    if x1='0' then NS <= a1;
    elsif x2='0' then NS <= a4;
    else NS <= a3;
    end if;
  when others => null;
  end case;
-- repeating for all modules, which might exist
end case;
end process;
```

The following VHDL code gives an example of hierarchical module call in the module Z_0 :

```
when a2 => dec <= '0'; inc <= '1'; NM <= Z1;
```

4 General Structure of Reconfigurable FSMs

Methods for synthesis of non-hierarchical reconfigurable FSMs (RFSM) were suggested in [6]. They permit to implement reconfigurable modules (such as Z_m) on the basis of RAM blocks in such a way that reloading the contents of RAM blocks enables us to reconfigure the FSM.

Let us consider the basic ideas of the cascaded RFSM model [6], which is composed of RAM blocks, programmable multiplexers (PM) and a register. By modifying the contents of the RAM blocks we can implement any desired behaviour within the scope of predefined constraints [1,6], which are the size R of the RFSM register, the number of RFSM inputs/outputs L/N and the number of reprogrammable levels G [6]. Reprogrammable levels are the primary building blocks of the RFSM combinational circuit. All the required parameterisation can be provided through VHDL *generic* and *generate* statements.

Figure 2 depicts the basic building blocks of RFSM. The PM is composed of a RAM and a multiplexer and is used for selecting appropriate input variables dependently on FSM states. VHDL code for the circuit in figure 2 was presented in [11].

Figure 3 shows a trivial RFSM [6]. It is composed of three blocks: an FSM memory, a state transition RAM (STRAM), and a programmable multiplexer (PM), which permits any input $x_i \in \{x_1, \dots, x_L\}$ of the PM to be selected in any RFSM state. Clearly we can provide any correspondence between states a_0, \dots, a_{M-1} and inputs x_1, \dots, x_L . STRAM enables us to generate codes for the next states and outputs. Obviously, any subset of output signals y_1, \dots, y_N can be generated in any state transition.

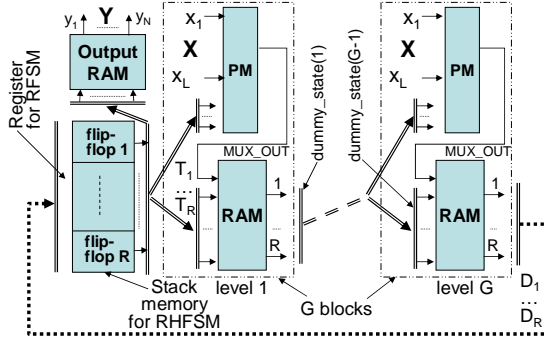


Figure 2: Basic structure of RFSM.

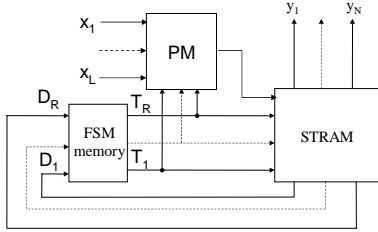


Figure 3: Trivial RFSM with limited capabilities.

By programming PM and STRAM we can implement any desired FSM behaviour within the scope of the predefined constraints. However the circuit depicted in figure 3 has a very significant limitation: any state transition can only be affected by a single input variable. An arbitrary state transition graph can be altered in such a way that allows this constraint to be satisfied by splitting state transitions and inserting dummy states as shown in figure 1 (see the state a_{dummy}). However in the general case this changes the behaviour of the FSM, increases the number M of states, and reduces the speed of state transitions. The solution proposed in [6] permits to overcome the problem. The blocks PM and STRAM are repeated for the required number of levels G , where G is the maximum number of input variables that have to be tested in any state. Dummy states can appear just in between levels (see figure 2), but they are not stored in any register and are considered as an intermediate signals between levels. Thus, any state transition can be executed during a single clock cycle.

5 Structure of RHFSM

Figure 4 depicts the basic structure of RHFSM, which permits to reconfigure the functionality of CC through reloading the RAM blocks.

Changes to the modules can be provided through re-switching segments of the same RAM using the most significant bits (MSB) of RAM address space. Thus, MSB in figure 4 permits to select the current module (CM), and the less significant bits (LSB) of RAM address space are used in the same manner as in [11].

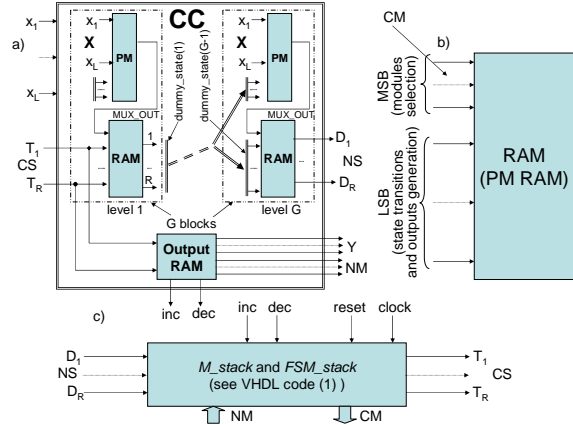


Figure 4: CC of RHFSM (a), dual-port RAM block (b), and RHFSM stack memories (c).

Note, that RHFSM in figure 4 can be reconfigured just statically. To provide for dynamic reconfigurability it is necessary to supply additional functionality, which would allow reloading RAM blocks during execution time.

Figure 5 presents a circuit, which implements such functionality. To reconfigure any module it is necessary to reload 2G memory blocks. Indeed, to modify functionality of any level in figure 2 it is necessary to reload the RAM block for PM and the RAM block responsible for state transitions (this block is designated as RAM in figures 2, 4). Activating the proper RAM block is achieved through the respective enable signals (see figure 5). In order to simplify reloading, dual port RAM blocks have been used in such a way that the first port provides a normal functionality and the second port enables the controller shown in figure 5 to change the contents of RAM. Two types of feasible reconfigurations have been considered. The first type, which is a wired reconfiguration, has been implemented and tested.

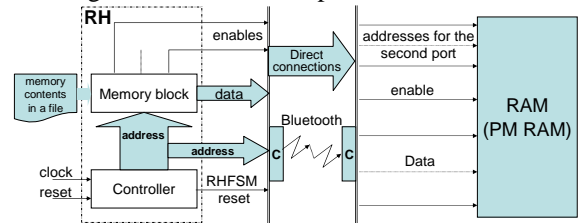


Figure 5: Dynamic reconfiguration of RHFSM.

The reconfiguration handler (RH) in figure 5 is composed of a source for data and a controller. The controller generates memory addresses and copies data from the source to the RAM blocks activated by the appropriate enable signals. When all the blocks

are programmed the controller resets the RHFSM and the latter is set into a working mode.

6 Advanced Techniques for Stack Operations in RHFSM

This section shows that the efficiency of the RHFSMs can be significantly improved through the use of the following methods [12]: 1) supporting multiple entry points to sub-algorithms; 2) employing fast unwinding procedure for stacks used as an HFSM memory in case of recursive module invocations; 3) establishing flexible hierarchical returns based on two alternative approaches, which can be chosen depending on the functionality required; 4) the rational use of embedded memory blocks for the design of RHFSM stacks.

6.1. Providing Multiple Entry Points to Sub-algorithms

Figure 6 demonstrates a fragment of a recursive message ordering algorithm. As we can see from figure 6 any hierarchical module invocation, such as that is done in the node a_2 , activates the same algorithm once again, starting from the node *Begin* (a_0). Skipping the node a_0 removes one clock cycle from any hierarchical call. However in this case the algorithm in figure 6 must have multiple entry points and a particular entry point will be chosen by the group of rhomboidal nodes enclosed in an ellipse. This possibility is provided by the additional tests performed in the nodes with hierarchical calls (such as a_2 and a_3 in figure 6). The following fragment demonstrates how these tests can be coded in VHDL for the state a_2 .

```
when  $a_2 \Rightarrow$  -- generating outputs and the signal inc
   $NM \leq z_1$ ; --  $NM$  is the next module
  if  $x_3 = '1'$  then  $NM\_FS \leq a_1$ ; -- this is because
    --  $x_1$  cannot be equal to 1 after the state  $a_2$ 
  elsif  $x_2 = '0'$  then  $NM\_FS \leq a_5$ ;
  elsif  $x_4 = '0'$  then  $NM\_FS \leq a_2$ ;
  else  $NM\_FS \leq a_3$ ;
  end if;
```

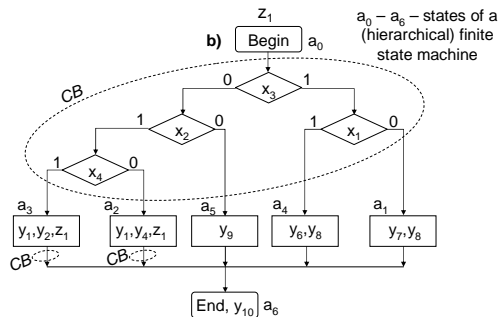


Figure 6: Message ordering algorithm.

Here NM_FS is the first state of the next module. The line *ref1* in (1) has to be changed as follows:

```
 $FSM\_stack(sp+1) \leq NM\_FS$ ;
```

6.2. Fast Stack Unwinding

Since there is just the node *End* after a_2 and a_3 , hierarchical activation of any one of the nodes a_1 , a_4 , a_5 (see figure 6) leads to termination of the module. To implement this termination in [7] the line in (1)

```
if dec = '1' then  $sp \leq sp - 1$ ;
```

is executed repeatedly until the pointer sp receives the value assigned at the beginning. In the general case, this value is assigned during the first call of the respective module (such as that depicted in figure 6) following by subsequent recursive invocations of the module. Repeated execution of the line $sp \leq sp - 1$; requires multiple additional clock cycles. To eliminate these redundant clock cycles the proposed method of fast stack unwinding is employed. The line *ref2* in (1) is changed as follows:

```
 $sp \leq sp - unwinding$ ;
```

where the signal *unwinding* is calculated as

```
 $unwinding \leq sp - saved\_sp + 1$ ;
```

and $saved_sp \leq sp$ at the first invocation of the module. Thus, redundant clock cycles for hierarchical returns will be avoided.

6.3. Execution of Hierarchical Returns

Hierarchical calls in (1) are carried out as follows:

```
if inc = '1' then
  -- error handling
   $sp \leq sp + 1$ ;
   $FSM\_stack(sp+1) \leq a_0$ ;
   $FSM\_stack(sp) \leq NS$ ; -- ***
   $M\_stack(sp+1) \leq NM$ ;
```

The line in (3) marked with asterisks sets the code of the next state NS during a hierarchical call. As a result, after a hierarchical return the top register of the FSM_stack contains the code of the proper HFSM state (i.e. no additional clock cycle is required). Since the NS is determined before the invocation of a module, the latter cannot affect the state transition, i.e. any possible change of the conditions x_1, \dots, x_4 (see figure 6) in the module cannot alter the previously defined next state. Very often this does not create a problem. However, for some practical applications it is a problem and it must be resolved. The following code gives one possible solution:

```
if rising_edge(clock) then
  if inc = '1' then
    -- error handling
     $sp \leq sp + 1$ ;
     $FSM\_stack(sp+1) \leq NM\_FS$ ;
     $M\_stack(sp+1) \leq NM$ ;
```

After a hierarchical return from NM , the code (4) sets FSM_stack to the state where the hierarchical call of the NM was executed. This enables us to provide correct transitions to the next state because all logic conditions that might be changed in the called module

NM have already received the proper values. However, this gives rise to another problem; namely it is necessary to avoid repeating invocation of the same module *NM* and iterant output signals. The following code overcomes the problem:

```

if rising_edge(clock) then
  if inc = '1' then
    -- error handling
    sp <= sp + 1;
    FSM_stack(sp+1) <= NM_FS;
    M_stack(sp+1) <= NM;
  elsif dec = '1' then
    sp <= sp - 1;
    return_flag <= '1';
  else
    FSM_stack(sp) <= NS;
    return_flag <= '0';
  end if;
end if;

```

The signal *return_flag* permits module invocation and output operations to be activated during a hierarchical call and to be avoided during a hierarchical return. Indeed, the *return_flag* is equal to 1 only in a clock cycle when the signal *sp* is decremented (see the code (5) above). As soon as the currently active module is being terminated, the control flow will be returned to the point from which this module was called. Thus, the top of the *M_stack* will contain the code of the calling module and the top of the *FSM_stack* will store the code of the calling state. The *return_flag* enables us to eliminate the second call of the same module. This is achieved with the aid of the following lines that have to be inserted in the code (2):

```

when state_with_module_call => NS <=
  -- testing the conditions and
  -- computation of the next state
  if return_flag = '0' then
    inc <= '1';
    -- specifying outputs
    NM <= -- assigning the next module
  else
    inc <= '0';
    Y <= (others => '0');
  end if;

```

6.4. Using Embedded Memory Blocks

Synthesis of RHFSMs has shown that stack memories (1) are very resource consuming. However, for implementing the functionality (1) embedded memory blocks can be used (such as that are available for FPGAs and shown in figure 7).

Embedded RAM was constructed using VHDL structural specification (for experiments Xilinx library component RAMB4_S8_S8 was used). Signal *sp_1* forms the RAM address and it is defined in VHDL architecture as follows:

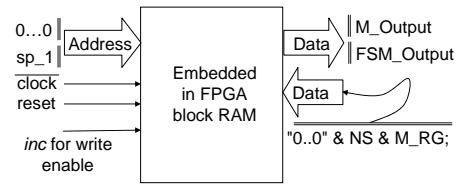
```

sp_1 <= (others => '0') when reset = '1'
  else sp - unwinding when dec = '1'
  else sp;

```

As a result, the signal *sp_1* provides for fast stack unwinding if required. Note that in figure 7 just a small part of the available memory block was used.

That is why unnecessary address inputs were set to 0 and some outputs were left unconnected.



```

process(clk,rst)
begin
  if reset = '1' then -- initialization
  elsif rising_edge(clock) then
    if inc = '1' then
      -- error handling
      sp <= sp + 1; FSM_RG <= NM_FS; M_RG <= NM;
    elsif dec = '1' then
      sp <= sp - unwinding; M_RG <= M_Output;
      FSM_RG <= FSM_Output;
    else
      FSM_RG <= NS;
    end if;
  end if;
end process;

```

Figure 7: Embedded memory block.

Since the stack pointer is common to both stacks, two segments of the RAM data bus are used for the *FSM_stack* and the *M_stack*, respectively. Two signals *FSM_RG* and *M_RG* enable the process shown in figure 7 to function at a single level. Switching to different levels of stack memories is provided through copying the RAM block outputs to the *FSM_RG/M_RG*, which is needed just for hierarchical returns. This permits very fast HFMSMs to be constructed. The same approach can be used for distributed memory available for Xilinx FPGAs. The following code demonstrates how to use the distributed library component RAM16X1S:

```

RAM16X1S_instM0 : RAM16X1S
generic map (
  INIT => X"0000")
port map (
  M_Output(0),
  sp_1(0), sp_1(1), sp_1(2), sp_1(3), M_RG(0),
  inverted_clk, -- inverted_clk <= not clock;
  inc);

```

N-bit stacks ($N > 1$) can be built from N components shown above.

Note that similar FPGA RAM blocks and distributed memory blocks can be used for all memories shown in figure 4. Thus, the proposed RHFSM is very well suited for implementation in commercially available FPGAs.

7 Experiments

The primary goal of the experiments was to prove on arbitrary selected examples that the results presented in the paper are correct and can be used for practical applications. The RHFSMs were implemented in FPGA of Spartan-IIIE family of Xilinx and were tested using virtual execution units described in [13]. All the experiments were performed using the stand-alone

board TE-XC2Se [14]. For dynamic modifications we used dual port RAM library components, such as RAMB4_S8_S8 of Xilinx. Thus, the first port took part in the RHFSM state transitions whilst the second port was used to reprogram the RAM from an external source. Consequently, FSMs with dynamically alterable behaviour were implemented on statically configurable FPGAs. The results of experiments have shown that the proposed methods and tools are correct and can be used for practical applications.

The experiments have also shown that the considered advanced techniques for stack operations in RHFSM permit to reduce the number of clock cycles (on average by 9%) and the required hardware resources (on average by 14%). The latter is basically achieved with the aid of replacement of an arbitrary logic with embedded in FPGA memory blocks.

8 Conclusion

The paper presents novel methods for the design of RHFSMs from hierarchical specifications. All the suggested structures are supported by VHDL templates (skeletal code, which can directly be used in engineering practice). The majority of the proposed methods have been verified in commercial hardware (in Xilinx FPGA). The proposed RHFSMs have two very important features: 1) they support the design hierarchy (including possible recursive calls, if required); and 2) they are statically and dynamically reconfigurable, which is very important for adaptable systems.

9 References

- [1] V. Sklyarov, A.A. da Rocha, A.B. Ferrari, "Synthesis of reconfigurable control devices based on object-oriented specifications", in J.C. López, R. Hermida, W. Geisselhardt, *Advanced Techniques for Embedded Systems Design and Test*, Kluwer Academic Publisher, pp 151-177 (1998).
- [2] W. Xu, S.G. Wang, A.L. Wang, G.B. Wang, "Toward an efficient self-organizing reconfiguration method for self-reconfigurable robots", *Journal of Intelligent and Robotic Systems*, 37, pp 415-425 (2003).
- [3] H. Bojinov, A. Casal, T. Hogg, "Emergent structures in modular self-reconfigurable robots", *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, USA, pp 1734-1741 (2000).
- [4] Y. Meng, "A dynamic self-reconfigurable mobile robot navigation system", *Proceedings of the 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, USA, pp 1541-1546 (2005).
- [5] V. Sklyarov, "Hierarchical finite-state machines and their use for digital control", *IEEE Transactions on VLSI Systems*, 7(2), pp 222-228 (1999).
- [6] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs", *Journal of Systems Architecture*, 47, pp 1043-1064 (2002).
- [7] V. Sklyarov, "FPGA-based implementation of recursive algorithms", *Microprocessors and Microsystems, Special Issue on FPGAs: Applications and Designs*, 28(5-6), pp 197-211 (2004).
- [8] I. Skliarova, A.B. Ferrari, "The design and implementation of a reconfigurable processor for problems of combinatorial computation", *Journal of Systems Architecture, Special Issue on Reconfigurable Systems*, 49(4-6), pp 211-226 (2003).
- [9] I. Skliarova, A.B. Ferrari, "A software/reconfigurable hardware SAT solver", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(4), pp 408-419 (2004).
- [10] R. Feldman, C. Haubelt, B. Monien, J. Teich, "Fault tolerance analysis of distributed reconfigurable systems using SAT-based techniques", *Proceeding of FPL*, Portugal, pp 478-487 (2003).
- [11] V. Sklyarov, I. Skliarova, "Design of digital circuits on the basis of hardware templates", *Proceedings of International Conference on Embedded Systems and Applications – ESA'03*, USA, pp 56-62 (2003).
- [12] V. Sklyarov, I. Skliarova, "Recursive and iterative algorithms for N-ary search problems". *Proceedings of 19th World Computer Congress, Professional Practice in Artificial Intelligence, Santiago, Chile*, pp. 81-90, August (2006).
- [13] V. Sklyarov, "Hardware/software modeling of FPGA-based systems", *Parallel Algorithms and Applications*, 17(1), pp 19-39 (2002).
- [14] Trenz Electronic, "Spartan-IIIE development platform", <http://www.trenz-electronic.de>, visited on 10/10/2006.