

Hierarchical Specification and Design of Control Systems in Robotics

Valery Sklyarov, Iouliia Skliarova
Department of Electronics, Telecommunications and Informatics, IEETA
University of Aveiro, 3810-193 Aveiro, Portugal
skl@det.ua.pt, iouliia@det.ua.pt

Abstract

Control systems that manage the behaviour of collaborative robots are in general hierarchical and parallel. Hierarchy permits the required functionality to be specified at different levels of abstraction. Higher levels of abstraction include more general operations that are described in detail at lower levels. Parallelism allows several robot components to be managed simultaneously. Control systems collaborate with execution systems (that are composed of mechanical blocks) in analyzing signals from sensors and affecting actuators. The paper suggests a method for the specification of hierarchical control systems and for the design of such systems based on the model of a hierarchical finite state machine. The latter has been simulated in software using the C++ language and implemented in hardware (in FPGA, in particular) based on the developed VHDL templates. The results of experiments demonstrate that the proposed technique offers a number of advantages.

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

1 Introduction

In collaborative systems, robots interact with each other to achieve an integrated behaviour aimed at the solution of a common problem. A good example of such a system was considered in [1], which describes collaborative robot soccer with possible dynamic allocation of the roles (namely, goalkeeping, defence and offence). Agent behaviour was implemented using traditional state transition based control and one of the conclusions was that “the complexity of the system can be greatly simplified and easily managed by building a hierarchy of state transition diagrams”. A number of examples have been shown in [1], which prove that hierarchy is a very important technique in a system of collaborative robots.

It is known that modular multilevel specification can be provided using a language called hierarchical graph-schemes (HGS) [2], which permits modules (sub-algorithms) and interactions between modules to be described. Consequently, HGS offer direct support for the hierarchical specification of algorithms, for example, for the hierarchical description of the state transition diagrams considered in [1]. In addition, HGS possess another very important characteristic. Since any HGS is considered to be a collaborative set of modules, which can be changed/altered if required, direct support for reconfiguration can easily be provided and this capability will be shown in the paper. Reconfiguration is very important for many practical applications, for example, for robot navigation systems [3]. Indeed, very few mobile robots can move from one environment to another without losing some performance or capabilities [3]. The technique proposed makes it possible to modify the functionality of control systems (within the same

hardware resources) easily, just by replacing modules. In fact, just an HGS module call has to be altered to make a given change.

The technique proposed in this paper provides direct support for behaviour-based robotics with the possibility of dynamic changes. A modular specification permits a collection of behaviours (modules) to be developed and used that achieve or maintain a given set of goals. Another very important aspect of the specification method proposed is that it is directly supported by an implementation model, which is a hierarchical finite state machine [2].

The remainder of this paper is organized in six sections. Section 2 describes the hierarchical specification of control systems. Section 3 characterizes hierarchical finite state machines that allow hierarchical specifications to be implemented in hardware. Section 4 discusses software modelling of hierarchical algorithms and presents some additional details of the hardware implementation. Section 5 shows how parallel algorithms can be described and implemented. Section 6 is devoted to experiments. The conclusion is in section 7.

2 Hierarchical Specification of Control Systems

Hierarchical algorithms described in HGS are constructed from modules with the aid of the method [2]. They are very useful for a number of practical applications (see, for example, [1]). Since all the required formalities are presented in [2] we will consider just an example taken from the scope of embedded systems. Suppose a system receives messages from an external source. The messages have to be buffered and processed sequentially according to

their priority. Each incoming message changes the message sequence, because it has to be inserted in the sequence in a position that is in accordance with its priority. Let us assume that the following strategy is applied:

1. All incoming messages are stored in a binary tree, whose nodes contain four fields: a pointer to the left child node, a pointer to the right child node, a counter, and a message. The nodes are maintained so that at any node, the left sub-tree contains only messages that are of a lower priority than the message at the node, and the right sub-tree contains only messages that are of a higher priority. The counter indicates the number of occurrences of the message associated with the current node (this is because some messages can arrive more than once). It is known that such a tree can be constructed and used for sorting various types of data [4]. In order to build this tree for a given set of messages, we have to find the appropriate place for each incoming node in the current tree. In order to sort the data, we can apply a special technique [4] using forward and backward propagation steps that are exactly the same for each node. Thus, a modular (and moreover recursive) procedure is very efficient.
2. As soon as a processing unit is free, a message from the binary tree with the highest priority is selected and processed.

Figure 1 depicts the HGS specification for the example, which is very similar to the sorting algorithm described in [5]. The same model can be employed if there is more than one processing unit and several units are working in parallel.

The HGS Z_0 first constructs the tree incrementally (calling HGS z_1) while input data are available. The latter is indicated by the logic condition x_1 ($x_1=0$ if data are available and $x_1=1$ if the flow of incoming data has finished). Then the HGS Z_0 sorts the data with the aid of the HGS z_2 .

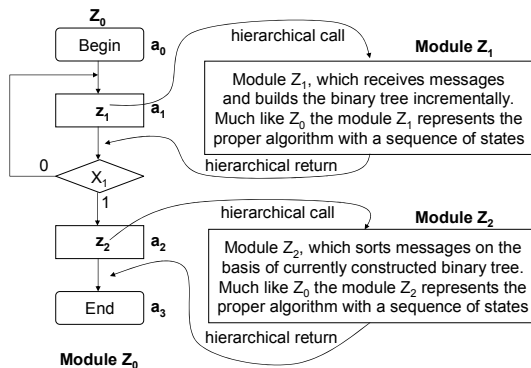


Figure 1: HGS specification of a message processing system.

Formally an HGS is a directed connected graph containing rectangular and rhomboidal nodes. Each HGS has one or more entry points (see the node *Begin*

in figure 1) and one or more exit points (see the node *End* in figure 1). Other rectangular nodes contain either *micro instructions* or *macro instructions*, or both. Any *micro instruction* Y_j , includes a subset of *micro operations* from the set $Y=\{y_1, \dots, y_N\}$. A *micro operation* is an output signal, which causes a simple action in the execution unit. Any *macro instruction* incorporates a subset of *macro operations* from the set $Z=\{z_1, \dots, z_Q\}$. Each *macro operation* is described by another HGS of a lower level. Each rhomboidal node contains one element from the set X , where $X=\{x_1, \dots, x_L\}$ is the set of *logic conditions*, i.e. input signals, which communicate the result of a test. The sets Y and X are associated with actuators and sensors of the respective execution unit. Directed lines (arcs) connect the inputs and outputs of the nodes in the same manner as for a graph-scheme (GS) [6].

Any HGS can be formally converted to a finite state machine with stack memory [2]. Figure 1 shows the states a_0, \dots, a_3 of the module Z_0 for such machine.

As we can see, in general an HGS is composed of modules. There are two acceptable and reasonable ways in which the binding between modules can be established [2]. These are before runtime and during runtime. The first kind of binding is called static because it is established outside the scope of dynamic physical control. Dynamic binding sets up links during runtime. This means that while one HGS is executing, another HGS might be either being modified or replaced. Any two HGS can be swapped by setting a new entry point in the appropriate module call. This feature provides direct support for static and dynamic reconfiguration. In addition, we can create a library of reusable modules.

3 Hierarchical Finite State Machine

An HFSM [2,5] permits the execution of an HGS (such as that shown in figure 1) and contains two stacks, one for states (*FSM_stack*) and the other for modules (*M_stack*). The stacks are managed by a combinational circuit (CC) that is responsible for new module invocations and state transitions in any active module that is designated by outputs of the *M_stack*. Since each particular module has a unique code, the same HFSM states (state codes) can be repeated in different modules. Any non-hierarchical transition is performed through a change of a code only on the top register of the *FSM_stack*. Any hierarchical call alters the states of both stacks in such a way that the *M_stack* will store the code for the new (called) module and the *FSM_stack* will be set to the initial state (normally to $a_0=0 \dots 0$, containing all zeros) of the module. Any hierarchical return just activates a pop operation without any change in the stacks. As a result, a transition to the state following the state where the terminated module was called will be performed. The stack pointer *stack_ptr* is common to both stacks. If the *End* node is reached when *stack_ptr*=0, the algorithm terminates execution.

4 Software Modelling and Hardware Implementation of Hierarchical Algorithms

All blocks of the HFSM considered in [2,5] have been described as classes in C++. The software that has been developed allows different hierarchical algorithms (HGS) to be tested, which can be read from files and executed in an HFSM. The latter is considered to be a set of objects instantiated from the appropriate classes (namely the stack class common to both stacks, and the combinational circuit - CC). CC is configured to run the supplied algorithm, which is read from the file. The algorithm can be executed in a virtual space allowing components of execution units to be modelled much like the technique proposed in [7]. A virtual execution unit is presented on a monitor screen in the form of interconnected visual objects, such as the execution part of a transfer line. A specific class in the C++ program supports each visual object.

This technique has a very important advantage. All the required algorithms can be tested in software and then converted to the proper hardware implementation. If the required execution unit cannot be found in the library, it can easily be created and attached to the system under investigation through pre-designed interfaces.

The functionality of the HFSM can be described in a hardware description language (HDL) much like it was done in [8], where all the required details can be found. The distinctive feature of the HDL code (compared with traditional FSMs) is the provision of state transitions at the following two levels:

- At the first level the code on the top register of the *M_stack* enables the HFSM to recognize the active module.
- At the second level the code on the top register of the *FSM_stack* and the values of external input variables (from the set *X*) enable the HFSM to execute the proper state transition, i.e. to form the code of the next state and, in the case of a hierarchical transition, the code of the next module.

Methods [7] make it possible to communicate between circuits implemented in hardware (in FPGA, in particular) and between virtual objects of robots and embedded systems simulated in software. This technique provides for a systematic approach to the development of hierarchically specified control systems in complex environments.

5 Parallel Logical Control

The hardware implementation of parallel algorithms is a complicated problem. In the general case it is very difficult and resource consuming to implement hierarchy and parallelism within the same control

system. We would like to underline that we are talking about real parallelism and we will not be discussing methods for the implementation of pseudo-parallel algorithms that are widely used in software for embedded systems. This section is divided into three subsections showing: 1) how to describe parallel logical control algorithms; 2) how to implement parallelism without hierarchy using a modified technique [9]; 3) how to implement hierarchy and parallelism within the same control system.

5.1 Description of Parallel Logical Control Algorithms

Parallel control algorithms can be described with the aid of the PRALU language proposed in [10]. However due to some unclear constructions in [10] that might lead to errors, the language was slightly modified, which permitted the gap between PRALU and graphical specifications (such as HGS [2,9] and GS [6]) to be bridged. Consequently not only textual specifications, but also graphical specifications can easily be used for describing parallel logical control algorithms.

Let us consider the sets $Y=\{y_1, \dots, y_N\}$ and $X=\{x_1, \dots, x_L\}$, introduced in section 2, and associated with the actuators and sensors of the respective execution unit. To make the relationship between the elements of *Y/X* and the actuators/sensors of the physical systems clearer, we will use abbreviations formed from the first letters of the relevant names in normal font for actuators, and in *italic* font for sensors. For example, the abbreviation for a “left sensor” will be written down as *ls* and the abbreviation for an actuator “move left” will be written down as *ml*. Let us introduce the following syntactic constructions for our specification, which differ from [10]:

- $-(ls=0)$ or $-(ls=1)$ – wait for *ls*=0 or *ls*=1 and only after that proceed to the operation written on the right-hand side of the relevant expression such as $-(ls=0)$ or $-(ls=1)$;
- $(ml \Rightarrow 0) / (ml \Rightarrow 1)$ – assign *ml* the value 0 / 1;
- Specifying just an abbreviation (such as *ml*) for an actuator without any value (such as $\Rightarrow 1$) means that the actuator receives an active value just during the respective transition and passive value after the transition. Later on this feature will be demonstrated on examples;
- Square brackets are used to combine expressions, for example $-(stop=0) [...] (stop=1) \rightarrow 3$ means: when *stop*=0 execute sequence in square brackets and when *stop*=1 execute transition to label 3.
- All the other constructions and rules are similar to [10].

Each specification has to be provided in such a way that ambiguity is avoided. For example, for any line like $-(ls=0) \rightarrow (ml \Rightarrow 1) \rightarrow \dots$, which says “assign

$ml \Rightarrow 1$ when $ls=0$ ”, we also have to know what to do when $ls \neq 0$, which is unclear in expressions in [10]. If in the expression above the value $-(ls=1)$ is not explicitly specified, it denotes: “no operation until $ls=0$ and $ml \Rightarrow 1$ when $ls=0$ ”.

5.2 Implementation of Parallel Algorithms Based on One-hot Encoding Technique

It is known [9] that ordinary (non hierarchical) GS [6] can be directly mapped onto FPGA-based circuits. The primarily idea of [9] is to set a correspondence between operational (rectangular) nodes of GS and individual flip-flops in FPGA (this is a pure GS-based one-hot state encoding technique). As a result, the number of flip-flops will be essentially increased compared with binary encoding. In practice this is not a problem because the cost of a flip-flop is negligible in electronic circuits (such as an FPGA or an ASIC). The method [9] can also be used for parallel control algorithms, which will be demonstrated through an example. Figure 2 depicts two interacting objects of a self-controlled transport section: a robot on the left hand side; and a container on the right-hand side that can be transported from left to right and vice versa. The following operations (provided through the relevant actuators) have to be executed:

- take (t) – the robot takes something (on the left-hand side);
- deliver (d) – the robot delivers something to the container;
- move right (mr/MR) – the robot/the container has to be moved right;
- move left (ml/ML) – the robot/the container has to be moved left.

The following sensors will be used:

- *completed* (c) – delivery is complete when ($c=1$) and it is not complete when ($c=0$);
- *left sensor* (ls) / *LEFT SENSOR* (LS) – left sensor for the left / right transport section;
- *right sensor* (rs) / *RIGHT SENSOR* (RS) – right sensor for the left / right transport section;
- *FULL* (F) – the container is full when ($F=1$) and the container is not full when ($F=0$).

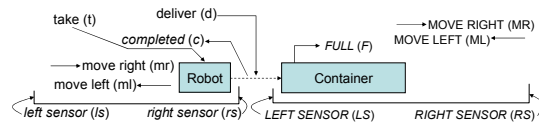


Figure 2. Self-controlled transport section

Let the container on the right-hand side be unloaded somehow (we will not worry about this). All the other

operations will be included in the specification of the algorithm. In accordance with the description above, the robot is controlled by two actuators mr and ml: if $mr \Rightarrow 1$, the robot moves to the right; when $ml \Rightarrow 1$, the robot moves to the left; $(mr \Rightarrow 1) \& (ml \Rightarrow 1) \equiv 0$. The track, where the robot is moving, is bounded by the sensors ls and rs in such a way that if $ls=1/rs=1$, the robot is at the left/right edge. The container has exactly the same behaviour, except capital letters are used instead of lower-case letters in the description. The transfer is controlled by two actuators d and t in such a way that if $t \Rightarrow 1$ the robot takes something at the left-hand side of the transfer line and if $d \Rightarrow 1$, the robot delivers something to the container. Sensor c signals ($c=1$) that the delivery has completed. Assuming that several iterations are needed for the robot to load the container, the following algorithm described in modified PRALU language is suggested (all the details of PRALU language can be found in [10]).

```

Start Parallel Algorithm:  $\rightarrow 1.2$  (1)
1:  $-(stop=0) \rightarrow [-(ls=0) \rightarrow (ml \Rightarrow 1) \rightarrow 1;$ 
    $-(ls=1) \rightarrow (ml \Rightarrow 0) t \rightarrow -(rs=0) \rightarrow$ 
    $(mr \Rightarrow 1) \rightarrow ;$ 
    $-(rs=1) \rightarrow$ 
    $(mr \Rightarrow 0) -(F=0; LS=1) \rightarrow d \rightarrow -(c=1) \rightarrow 1;]$ 
    $-(stop=1) \rightarrow 3$ 
2:  $-(stop=0) \rightarrow [-(F=0; LS=0) \rightarrow (ML \Rightarrow 1) \rightarrow 2;$ 
    $-(LS=1) \rightarrow (ML \Rightarrow 0) \rightarrow -(F=1; RS=0)$ 
    $\rightarrow (MR \Rightarrow 1) \rightarrow 2;$ 
    $-(RS=1) \rightarrow$ 
    $(MR \Rightarrow 0) \rightarrow 2;]$ 
    $-(stop=1) \rightarrow 4$ 
3,4  $\rightarrow$  end

```

Here: the symbol “-” is interpreted as “wait for”; the symbol “ \rightarrow ” means “execute”; when $stop=1$ the execution of the algorithm is ended (however any currently executing operation has to be completed before the termination of the algorithm). If there is just a semicolon on the right-hand side of the operator “ \rightarrow ”, the previous test of sensors (logical conditions) has to be repeated. For example, after the rightmost operator “ \rightarrow ” in the expression “ $-(rs=0) \rightarrow (mr \Rightarrow 1) \rightarrow ;$ ”, the logical condition rs has to be tested again in such a way that while $rs=0$, $mr \Rightarrow 1$. Chains with labels 1 and 2 are executed in parallel and they describe the control of the robot and the container, correspondingly. Note that such control is considered just for explanatory purposes and is greatly simplified. It is assumed that at the beginning the robot and the container are at arbitrary positions.

The rules (derived from [9]) for constructing a parallel control device (implementing the described algorithm) are the following:

- Use the method [9] without any change for ordinary chains like $-(F=0; LS=0) \rightarrow (ML \Rightarrow 1) \rightarrow 2;$ $-(LS=1) \rightarrow$

$(ML \Rightarrow 0) \rightarrow (F=1, RS=0) \rightarrow (MR \Rightarrow 1) \rightarrow 2;$
 $(RS=1) \rightarrow (MR \Rightarrow 0) \rightarrow 2;$

- Activate more than one flip-flop for parallel branches like *Start Parallel Algorithm*: $\rightarrow 1.2$ (we have to activate in this particular transition two flip-flops for the chains 1 and 2 in parallel);
- Use the AND function in the case of chains merging such as $3,4 \rightarrow \text{end}$. This permits the two parallel chains 3 and 4 to be terminated.

Since we do not have other types of transitions, the method [9] can be applied directly. Output functions can be implemented trivially (similarly to any finite state machine based on one-hot state encoding – see [9] for details).

Note that any description such as that considered above can be formally converted to an equivalent description in HGS, which was shown in [11]. For example, formulae (1) can be converted to the HGS depicted in figure 3,a (\overline{RE} designates that re-entrance to any module Z_1, Z_2 is prohibited). Figure 3,b demonstrates how the HGS can be extended to provide a closer relationship to (1). Figure 3,c shows how the module Z_1 (the chain 1) can be described in HGS. The other module Z_2 (chain 2) can be similarly and easily described. In an analogous way, an HGS specification can be converted to PRALU. Thus, the two specification methods considered are mutually convertible.

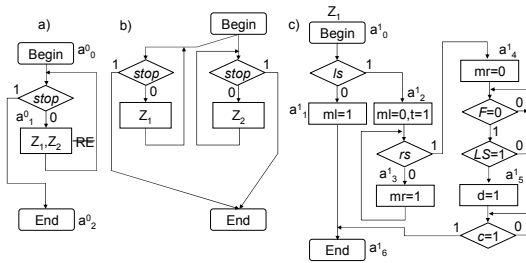


Figure 3: HGS for the example in figure 2 (a), interpretation of parallel chains (b), HGS for the module Z_1 (c)

Figure 4 demonstrates how to implement the HGS depicted in figures 3,a and 3,b in hardware. Labels a^i_j in figure 3 correspond to FSM states (see [2,9] for details), where i is an index of module Z_i and j is a number of state in the module Z_i . In one-hot state encoding for each state a flip-flop is allocated (see figure 4).

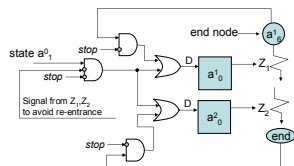


Figure 4. Hardware implementation of parallel branching and merging operations

Let us assume that initially all the flip-flops (of D type) are reset to 0 (i.e. all their outputs shown in figure 4 are set to 0). To start the algorithm we have to activate the state a^0_0 (see figures 3,4). This can be done by setting the flip-flop a^0_0 in the state “1”. Subsequently, if $stop=0$ a transition to the state a^0_1 is executed and then two flip-flops a^0_1 and a^0_2 are set to 1 in parallel and they force parallel execution of the modules Z_1 and Z_2 . After any transition to the “end” node, the condition $(stop=0) \& (\text{“end node is active”})$ is tested. If $stop=1$, the first state in the relevant module (either Z_1 or Z_2) is not activated and all the flip-flops used for this module will be set to 0. When both Z_1 and Z_2 are terminated a transition to the state a^0_2 is executed. Thus, the modules Z_1 and Z_2 cannot be executed until the flip-flop a^0_0 is set to “1” again. Thus, to start execution again it is necessary to reset a^0_2 and to set a^0_0 . It should also be mentioned that the requirement considered above, namely “any currently executing operation has to be completed before termination of the algorithm”, is satisfied. Implementation of any sequential algorithm, such as that is shown in figure 3, c, can be provided using the method [9] without any change.

Note that it is not necessary to allocate states such as a^1_0 and a^2_0 . Indeed, for each entry point to the modules Z_1, Z_2 we can use states immediately following the node “begin”, such as a^1_1 and a^1_2 in figure 3,c. However, the relevant logical conditions (such as ls in figure 3,c) have to be properly tested.

5.3 Hierarchical Parallel Logical Control Systems

It is known [10] that PRALU descriptions are synthesizable. The relevant hardware circuits can be built on the basis of a parallel FSM with partial states in such a way that the FSM can be in several of these states at the same time. This is achieved through a special ternary encoding allowing state codes to be assigned such that they are intersected for parallel algorithmic branches (chains) making it possible to recognize two or more different states at the same time. For example, the code 0-1-0 enables us to recognize the state 011-0 for one chain and the state 0-100 for another chain. The details of this technique are rather complicated and we will omit them due to limitations in the length of this paper. This method is entirely described in [10]. We would like to underline that using the method [10], parallel algorithms can be implemented based on ternary state encoding allowing the size of state registers to be decreased significantly compared with one-hot state encoding (although the actual size is, as a rule, much larger than $\lceil \log_2 R \rceil$, where R is the number of states). Decreasing the size of state codes is very important for stack-based implementations. Since the PRALU language can be formally converted to HGS and the latter can be implemented in hardware using an HFSM model, we can synthesize circuits for parallel hierarchical

algorithms (although some constraints that are presented in [12] have to be taken into account). In other words using the HFSM model we can also provide a stack-based invocation of sub-algorithms (modules/chains) allowing us to take advantage of the method [2]. Since the mechanisms that support hierarchy are based on pre-established operations with a stack memory, and are separated from the logic synthesis of primary combinational circuits, we can also apply the methods for the synthesis and verification of parallel algorithms that are considered in [10,11].

6 Experiments

The primary goal of the experiments was to prove on arbitrarily selected working examples that the methods presented in the paper are correct. The control systems were implemented in FPGAs of Spartan-IIIE family of Xilinx. We have used the stand-alone board TE-XC2Se [13]. The results of experiments have shown that the proposed methods and tools make possible to construct control systems implementing hierarchical or parallel algorithms and can be used for practical applications. We found that although hierarchical and parallel algorithms can be implemented within the same system, using hierarchical parallel algorithms is very resource consuming and many constraints have to be taken into account. In many cases, autonomous HFSMs working in parallel are better. Hierarchical circuits based on the proposed templates are very fast and do not require many resources. Parallel implementations based on one-hot encoding technique are also quite efficient in terms of resources and execution time.

7 Conclusion

The paper describes two types of specifications that can be efficiently used to describe the functionality of parallel and hierarchical control systems. Such systems can be seen as embedded controllers and they are needed for many practical applications in robotics. The specifications considered (namely HGS and PRALU) are mutually convertible and permit the methods and tools available for them to be combined. For example, an HGS is a formally synthesizable specification based on the model of a hierarchical finite state machine and PRALU is supported by numerous methods for the synthesis of the combinational part for a parallel FSM and for formal verification. Finally, the paper shows how to design hierarchical, parallel and hierarchical parallel systems from specifications presented in the languages considered and summarizes the advantages and disadvantages of each particular implementation. The tools that have been developed for software/hardware modelling of control systems interacting with execution units are also described.

Acknowledgment

The authors would like to acknowledge Ivor Horton for his very useful comments and suggestions.

8 References

- [1] G. Sen Gupta, C.H. Messom, S. Demidenko, "State transition based (STB) role assignment and behaviour programming in collaborative robotics", *Proceedings of the 2nd International Conference on Autonomous Robots and Agents*, Palmerston North, New Zealand, pp 385-390 (2004).
- [2] V. Sklyarov, "Hierarchical finite-state machines and their use for digital control", *IEEE Trans. on VLSI Systems*, 7(2), pp 222-228, (1999).
- [3] Y. Meng, "A dynamic self-reconfigurable mobile robot navigation system", *Proceedings of the 2005 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, Monterey, USA, pp 1541-1546 (2005).
- [4] B.W. Kernighan, D.M. Ritchie, *The C programming language*, Prentice Hall (1988).
- [5] 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).
- [6] S. Baranov, *Logic synthesis for control automata*, Kluwer Academic Publishers (1994).
- [7] V. Sklyarov, "Hardware/software modeling of FPGA-based systems", *Parallel Algorithms and Applications*, ISSN 1063-7192, 17(1), pp 19-39 (2002).
- [8] V. Sklyarov, "Models, methods and tools for synthesis and FPGA-based implementation of advanced control systems", *Proc. of the 2nd International Conference on Mechatronics*, Kuala Lumpur, Malaysia, pp 1122-1129 (2005).
- [9] V. Sklyarov, A. Ferrari, "Design and implementation of control circuits based on dynamically reconfigurable FPGA", *Proceedings of IEEE International Conference on Electronics, Circuits and Systems*, Lisbon, Portugal, pp 527-530 (1998).
- [10] A. Zakrevskij, *Parallel algorithms of logical control*, Minsk, Academy of Science (1999).
- [11] A. Zakrevskij, V. Sklyarov, "The specification and design of parallel logical control devices", *Proceedings of PDPTA'2000*, Las Vegas, USA, pp 1635-1641 (2000).
- [12] V. Sklyarov, "Graphical description and hardware implementation of parallel control algorithms", *Proceedings of PDPTA'99*, Las Vegas, USA, pp 1390-1396 (1999).
- [13] Trenz Spartan-IIIE development platform, www.trenz-electronic.de, visited on 7/10/2006.