HIERARCHICAL INTERFACES FOR HARDWARE/SOFTWARE SYSTEMS

Tudor NICULIU
Universitatea «Politehnica» Bucuresti,
Facultatea de Electronica si Telecomunicatii
Bd. Iuliu Maniu 1-3
77202 Bucuresti, Romania
tudor@messnet.pub.ro

Chouki AKTOUF
Institut National Polytechnique de Grenoble
LCIS-ESISAR Valence
50, Rue Barthelemy de Laffemas
BP 54, 26902 Valence, France
Chouki.Aktouf@esisar.inpg.fr

Sorin COTOFANA
Delft University of Technology
Faculty of Electrical Engineering
Mekelweg 4
2600 GA Delft, The Netherlands
S.D.Cotofana@dutepp0.et.tudelft.nl

KEYWORDS

Hierarchical, Object-oriented, AI in simulation, Combined simulation, Electronics

ABSTRACT

Competent design of hierarchical interfaces for hardware/software systems needs the convergence of three concurrent research directions: the study of hierarchy types, the intelligent communication between different domains, the formalization of verification/test. We aim to extend the theory of hierarchy types, in order to integrate communication properties as well as correctness and testability, to suit the behavioral specification of today’s complex system design. The high level approach of these problems permits the intervention of an intelligent agent for adapting techniques, models or methods to the particular design: a designer, assisted by man-machine dialog interface, or an artificial intelligence system. Behavioral design-for-testability offers a good startup. Testability measures the difficulty of test; it is used in this paper to emphasize the high-level strategy. Design-for-testability techniques (full and partial scan, test point insertion or built-in self-test) increase the fault coverage and reduce the test generation time; as they aim to modify the system's specification to improve testability, performing them at higher levels of the design hierarchy reduces the complexity of their generation and application. An intelligent use of the acquired knowledge on design for communication, verification and testability is enabled.

ARGUMENT & CONCEPTS

We consider the concept of simulation integrating: design (structural simulation of the system’s function) and verification (functional simulation of the system’s structure), as well on higher as on lower abstraction levels of different hierarchy types. The complexity of the simulation’s object, e.g. interfaces needed for communication between different domains, used to define heterogeneous systems, imposes a hierarchical approach.

Generally, multiple, coexistent and interdependent hierarchies structure the universe of models for complex systems, e.g., hard/ soft ones. They belong to different hierarchy types, defined by: abstraction levels, block and class structures, symbolization and knowledge hierarchies, whose study and formalization result in separation of basic hierarchy types, that can be interpreted as the object-oriented (Booch 1991) and the symbolization paradigm (Bibel 1993). Abstraction and hierarchy are semantic and syntactical aspects of a unique fundamental concept, the most powerful tool in systematic knowledge; hierarchy results from formalization of abstraction.

The structure of the communication between heterogeneous parts of the object-system, and with its exterior, should reflect the hierarchies of the simulation technique/ model/ method. Considering the heterogeneous relations between different functions that collaborate to build the behavior of the simulated system, we have to extend the scope of man-machine dialog, from standard I/O functions, to assistance of iterative knowledge-based co-simulation.

Representation is a 1-to-1 mapping from the universe of systems (objects of simulation) to a hierarchical universe of models, so a representation can be inverted. A model must permit knowledge and manipulation, so it has two complementary parts/ views: description and operation. If models correspond to classes, in a formal approach, specifications are instances; if models are formalized as languages, specifications are expressions.

We define a general hierarchical approach for complex simulation, applying it in handling communication between different domains implied by hard/ soft systems; e.g., combined simulation of dynamic objects (handled in software) and parallel activities (realized in hardware).

APPROACH

The planned framework permits, at any level of abstraction of the simulation hierarchy:

- description of the system in a convenient and commonly used language, e.g., C++ (Stroustrup 1997) extended for parallelism by synchronization constructs (Kumar et al. 1996);
- automatic partition of the description into hardware and software;
- correct and complete communication between heterogeneous parts and with the exterior;
- simulation and validation of the whole system during any design phase.

If one of the imposed properties (design constraints) is considered as not being fulfilled after applying a technique, using a model and suitable methods for measure and improvement, different strategies permit altering one of the technique/model/method, to repeat the process for the initial behavioral specification or the one resulted from prior (insufficient) improvement. This calls for an intelligent choice of the designer or the AI system that assists/ automates the design. The methods are recursive (iterative) to handle the different components in the behavioral specification of the system. The process continuation is controlled by measurement functions, so, generally, these must be called for each call of the improvement functions, but there are also methods demanding for a global improvement based on a prior measurement. The behavioral adaptable design for communication properties, correctness and testability is synthesized in the following BADCCT algorithm:

```
class BehavioralDescription ...
BADCCT (   BehavioralDescription behavSpecif,
           Bool increment ) : BehavioralDescription
begin
techniques := Ø; models := Ø; methods := Ø; good := false;
while (not good) begin
    technique := selTech (behavSpecif,
                          techniques, models, methods);
    if (not technique in techniques) begin
           techniques.add (technique); models := Ø end;
    model := technique.selModel (behavSpecif, models);
    if (not model in models) begin
           models.add (model); methods := Ø end;
    specification := model.detSpec(behavSpecif);
    method := model.selMeth (specification, methods);
    if (not method in methods) methods.add (method);
    if (integrated) begin
       (good, enough) := method.measure (specification);
       while (not enough) begin
           specification := improveLoc (specification);
           (good, enough) :=
                          method.measure(specification)
           end
       end
    else (good, specification) :=
                      improveGlob (specification,
                          method.measure(specification));
    if (increment) behavSpecif :=
                  model.returnToBehavDescr (specification)
    end;
return model.returnToBehavDescr (specification)
end.
```

Boolean variables that control the decisions are:

- increment - decides whether to keep the more but not enough adequate specification when applying a new method/model/technique or to reset to the initial specification;
- good and enough - represent the limits corresponding to different criteria controlling the continuation of the cycles; they are actualized by the function that measures the adequacy of the specification to the necessary properties of communication, correctness, testability;
- integrated – expresses the decision to apply together, for each iteration of the model's method, the function to measure and that to improve the adequacy; otherwise, improvement is applied after having measured the entire behavioral specification.

To begin, the intelligent component that makes the design system adaptable, by selecting the next technique/ model/ method to be applied, is replaced by experiment associated to man-machine dialog: the comparison results of completeness checking versus consistency checking regarding communication, of validation versus formal verification, of structural testing versus functional testing are used to choose another technique/ model/ method.

HIERARCHY TYPES

Hierarchies are of different types, corresponding to the kind of abstraction they reflect:

- symbolization hierarchy - corresponds to formalization of all kind of types, in particular also of hierarchy types;
- conceptualization (class) hierarchy - builds a virtual framework to represent all kinds of

hierarchies, based on form-contents dichotomy (class-instance), modularity, inheritance, polymorphism; an object is defined by identity, state and behavior, being instance of a class, that defines its internal structure and behavior, as well as its external behavior;

- knowledge hierarchy - corresponds to reflexive abstraction, so each level has knowledge of its inferior levels, including itself; recurrence of structures and operations enables approximate self-knowledge (with improved precision on the higher levels of knowledge hierarchies); a continuous model for hierarchy levels would perhaps offer a better model for intelligence; a possible interpretation of such hierarchies is: real time of the bottom levels, corresponding to behavior, is managed at upper levels, corresponding to strategies, and abstracted on highest levels, corresponding to types;
- construction (simulation) hierarchy - autonomous levels for different abstraction grades of description build a design/verification (= simulation) framework; time is explicit at highest (behavioral) levels (being integrated in the model), and exterior on lowest levels (being implicit for the system's activity); artificial intelligence approaches try to configure the simulation hierarchy type as reciprocal to the knowledge hierarchy type;
- structure hierarchy - helps managing all other hierarchy types on different levels, following the principle «Divide et Impera et Intellige», by recursive decomposition in autonomous blocks.

The different hierarchies can be represented symbolically and object-oriented; that means: the first two enumerated types build a reference system for any hierarchy type. All hierarchy types have in common structures allowing for the following description:

$\mathbf{(U, \{H_i \hat{I} S_h\})}$ = universe -
structured by different hierarchies H_i,
$\mathbf{S_h}$ = set of hierarchies defined on universe **U**:
$\mathbf{H = (Rel\_eq, \{(Level_j, Structure_j) : j \hat{I} S_l\},}$
$\mathbf{Rel\_ord, \{A_j : j \hat{I} S_l\}) =}$ generic hierarchy:
$\mathbf{S_l}$ = set of hierarchy levels,
Rel_eq = equivalence relation -
divides the universe in levels,
$\mathbf{Structure_j}$ = structure defined on level **j**,
Rel_ord = order relation (total) -
defined on the set of hierarchy levels,
$\mathbf{A_j \hat{I} \{ (x,y) : x \hat{I} Level_{j-1},\ y \hat{I} Level_j,\ j \hat{I} S_l \}}$
= relation of abstraction.

For example: The classical activities in complex systems simulation (Gajski et al. 1994), that regard comparisons between different levels of the construction or knowledge hierarchy, as well as of the structure hierarchy, can be expressed object-oriented and simulated or formally approached by symbolization of the more abstract entities, as sketched below:

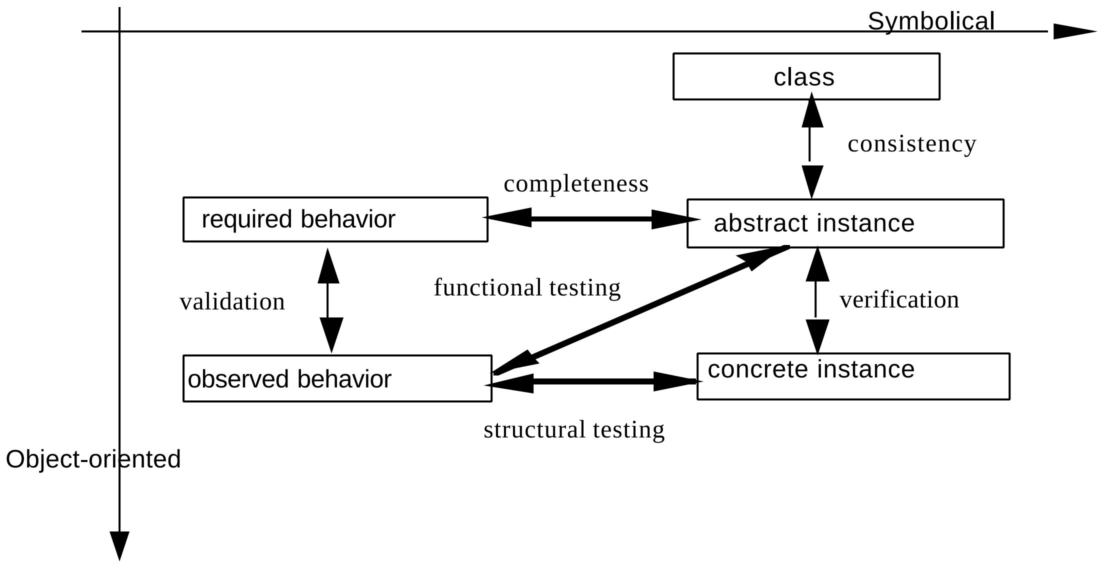

Figure 1: Hierarchical Simulation

PERFORMANCE EVALUATION

To illustrate our general approach, currently under development as an intelligent framework, we describe the performance evaluation needed for further choices during the assisted design process of dynamic parallel systems. Let us assume that the total execution time for the system's current partition has to be estimated and that the following six characteristics are known:

1. the execution time of each simple method (no calls to other methods),
2. the medium iteration count of each cycle,
3. the branch probabilities of each conditional statement,
4. what methods can be executed in parallel,
5. which objects are to be synthesized to software and which to hardware,
6. which parallel-executable hard-methods of the same object are linked by synchronization constraints of the form:

class X { ... m1 (); m2 (); ...};
#m2_calls <= #m1_calls <= #m2_calls + constant.

To provide an answer to this problem we need to construct a directed a-cyclic graph containing the method-calls (a-cyclic because no recurrence is permitted, for hardware compatibility), and, the following information for each method:

- a block of statements,
- a list of methods that are called in parallel (constructed from the list of methods that can be executed concurrently and are on direct paths from methods called in parallel),
- a list of synchronization constraints to which it participates (only the parts that can postpone its call, the rest retains the counterpart method),
- the number of calls,
- the cost (execution time).

The parallelism relation is not transitive (just reflexive and symmetric). Parallelism is possible only between methods that are not on the same path in the directed a-cyclic graph (don't call each other, directly or indirectly) and is conditioned by hard/soft realization (the number of soft methods in a list of parallelism is given by the number of parallel working processors). Synchronization constraints can appear only between methods of the same object. The most important classes are: Method, MethodList, Statement, SyncList, Evaluation, Stack. The textual description language for the abstract problem is:

```
method       ::=   { statement; ... statement;}
statement    ::=
     IF (probability) method ELSE method
     |  FOR (number) method
     |   method     |        parallel-call
parallel-call ::=   (method, ... , method)
```

Considering the call hierarchy of the methods (tree - no recurrence, directed a-cyclic graph - no dependence of the method execution time on the call context), the method-DAG:

1. is constructed top-down;
2. is actualized for concrete values of the method attributes and their concrete relations;
3. is visited recursively (depth-first post-order) to determine execution time for each method; each method keeps track of the number of times it has been called, to enable estimation considering synchronization constraints;

the last result is the total execution time, implying a global clock.

Of different approaches to handle synchronization constraints, we firstly experimented a simulation-oriented one: the system's behavior is simulated to estimate the execution time. Synthetically: if the synchronization constraints are not verified, the call is postponed, marking this in method-list, together with the value of the global clock; this time value will be used when the method's call will be successful, to determine the waiting time that must be added to its execution time, considering parallelism. A waiting call is retried when its counterpart is successfully called. When a method call is postponed, a dummy-return-value 0 and a list of ascendants (calling methods), whose estimation is influenced by the correction of the postponed method's time, can avoid recurrence interruption; lists of calling methods, implemented as stacks, are needed for each parallel call. The execution time of a method is computed hierarchically from the components of its block. The contribution to time estimation of a directly or indirectly parallel called method has not the same form in the case of synchronous parallelism as in that of asynchronous parallelism.

The algorithm for synchronous parallelism is:

- construction of a list of methods, representing the directed a-cyclic graph of the system, containing the methods and their relations;
- deduction of actual directly or indirectly parallel calls from the list of methods, parallelism information and hard/soft partition information - to permit concurrent execution, two methods should have only possible parallel descendants, including themselves;
- time evaluation of a method with parallelism, synchronization constraints and deadlock determination.

Presently, we attempt to accomplish multi-hierarchical communication between different domains implied in complex systems, as hardware/software ones. We next concentrate on behavioral enhancements needed to adapt such systems to simulation and test.

BEHAVIORAL DESIGN FOR TESTABILITY

Design-for-testability (DFT) must suit the behavioral specification of today's complex system design. It aims to adapt the system's specification to improve testability; to reduce test generation and application complexity, the specification must be adapted for testability at behavioral levels of the design hierarchy. Referring to high-level synthesis, design-for-testability can operate before, while or after it. The first choice permits the intervention of an intelligent agent for adapting the design-for-testability technique, model or method to the particular design. We call it behavioral adaptable design-for-testability (BADFT): it improves the testability, measured with adequate methods, direct on the behavioral specification or aided by special representations, that have to permit returning to the behavioral description after improving the testability of the system to be designed. We present synthetically high-level descriptions and design-for-testability techniques. Then, we concentrate on partial scan design-for-testability techniques, comparing structural, textual and formal approaches. The results are general enough to be valid for systems, either hard, soft or hard/soft.

Memory elements - registers (arrayed flip-flops)/ flip-flops/ latches (unclocked flip-flops) - are represented in behavioral hardware descriptions by variables or signals. Variables ("containers") are description objects local to processes/ subprograms, used to store intermediate values between sequential statements, characterized by free assignment (exception: global variables). Signals ("wires") are permanent description objects to link concurrent elements: components/ processes/ concurrent assignments, demanding synchronized assignment, declared locally - within architecture, block or other declarative region, or globally - in extended package (Fleury, Aktouf, Robach, 1999).

In the context of a process synchronized by a clock signal, in a behavioral description, signals implicated in simple/ multiple signal assignment generate memory during synthesis. Instances of this rule are:

- multiple synchronization points, i.e., several wait statements with identical synchronization conditions) infer memory elements, allowing to describe Finite State Machines without declaring the state variables; several wait statements with different synchronization conditions are not yet synthesizable;
- if a signal is read (its value used) before being assigned it infers memory; a particular case of this rule is a conditional statement that does not affect a signal in every of its branches (conditional signal assignments have equivalent processes containing conditional statements).

An analog rule can be formulated for variables: Inside a process, a variable that must hold values between iterations of the process implies memory elements; that is: a variable which is set but not used between synchronization statements infers memory; a variable which is read before being assigned also infers memory.

The context is not restrictive, as all concurrent statements are equivalent to processes (excepting direct/ component instantiation). For called subprograms the rules of memory inference can be deduced directly: pure functions (no side effects) do not - while procedures (side effects) do infer memory elements.

BEHAVIORAL PARTIAL SCAN

Scan techniques imply a test mode added to the design, with or without separate clock: when this mode is active, all registers - for full scan, or just a part of them - for partial scan, are connected to form a shift register. Such a design for testability technique introduces two extra primary inputs: for test data and test enable, and an extra primary output, for test data. This allows controlling and observing the states of the scanned registers, reducing the complexity of the necessary test for the sequential system almost to the one demanded by a test for a combinational system.

The partial-scan problem is the selection of the scan registers following a strategy to find an optimum compromise between testability improvement and the implied cost. Most of the methods are applied by now on register-transfer level and lower levels of the design hierarchy. The applied methods choose different selection criteria based on combinations of different kind of models, listed below:

- Structural (graph-based): a weighted directed graph (S-graph), models the flip-flops as nodes and the combinational paths between them as arcs; testability is represented by the weights, but is also related to total length of feedback cycles and to sequential depth (maximum length of a path in the S-graph that links an input to an output).
- Formal (FSM-based): a Finite-State-Machine network, corresponding to the behavioral specification, models sequential behavior; the operation of this model is simplified using implicit techniques for state enumeration, efficiently generating the state transition graph and storing it as characteristic functions in Reduced Ordered Binary Decision Diagrams, thus allowing to estimate testability of different elements and to guide scan selection.
- Textual (HDL-based): a Hardware Description Language behavioral specification guides the testability measurement and improvement; the measure is a combination of different aspects that contribute to the low testability of the hardware corresponding to signals/ variables in the behavioral specification of the system/ component.

In a structural approach testability is related to cycles and sequential depth, but can also be represented, partially or totally, by the node weights. The S-graph is used as intermediate format of high-level synthesis in literature (Cheng, Agrawal, 1990), but can be managed as model of the behavioral description.

From structural point of view, feedback cycles among registers are mainly responsible for low testability, as their total length influences exponentially the complexity of test generation. Only the maximal strongly connected sub-graphs must be considered when reducing the cycle-size-sum; strong connection is dual to a-cyclic. The next structural attribute that testability depends on (linearly) is sequential depth.

The algorithm used to eliminate cycles is minimum feedback vertex set = finding the smallest set of (weighted) nodes whose removal results in a directed a-cyclic graph. Self-loops and other loop-structures that do not pose test problems can be excepted, e.g., by absorbing them in nodes. As this algorithm is NP-complete different solutions to reduce the complexity must be applied to the graph. Formal or heuristic testability measures (to be improved) can be defined on the structural model, e.g., probabilistic metrics for signals/ variables (randomness, transparency) operated within a Markov chain model, respectively, with composition rules. The weights of the S-graph retain the cost/ gain to scan a node.

An other form of graph model, with weighted vertices or arcs, could be used to integrate a testability measure in a less complex graph-theoretical algorithm to eliminate cycles (e.g., maximum-flow problem).

An intelligent interface assures the translation, in both senses, from behavioral hard/ soft description to a structural representation of the required behavior, that guides the partial-scan selection, using a knowledge base to generate the weighted directed graph (flip-flops, combinational paths) and to return to text the differences caused by transformation for testability improvement. The rules of correspondence between description object (signal/ variable) assignments and registers, as well as rules to translate the data flow in the behavioral specification to weighted arcs in the graph counterpart and to combine different testability measures in node weights, guide the first step, while incrementing rules for hard/ soft description solve the last one.

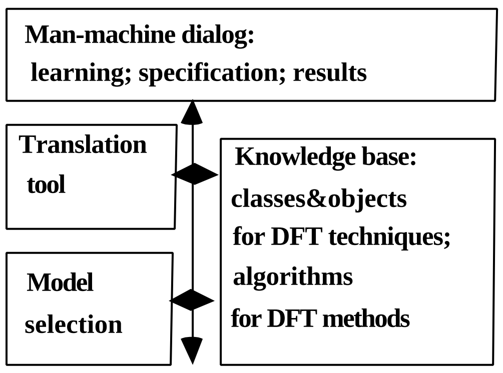

Figure 2: Behavioral Adaptable DFT framework

Testability for behavioral description is based on data objects (like signals or variables) and functions (relations, operations, transformations); it results, as usual, from controllability and observability. If the test vectors applied on the input of a description module form a complete vector set, i.e., that causes every element of the module, on the current description level, to traverse its whole defined value range for objects and input range for functions, the module is controllable. If the values of all elements of the module can be determined of its response to a complete vector set, the module is observable.

The full-scan case is solved in (Fleury, Aktouf, Robach, 1999). First, memory elements are located among the hardware-objects; each is related to a number of flip-flops (a register whose length is) depending on the type of the considered variable/ signal. Then, a scan chain is built that contains all design flip-flops.

The difference to full-scan needed by partial-scan for the return translation reduces to a pointing scheme for the scanned objects among signals/ variables of the behavioral specification; this can be managed by an adequate data structure or by using access types or attributes in hardware description languages. What remains to add is the register selection for partial-scan. Register elements are flip-flops; when the context permits, “register” is used also for register element. Latches are not present, as we suppose synchronized specifications. In principle, flip-flops are selected for scan, but when a register is used parallely, it is candidate entirely for scan. For partial-scan, the variables/ signals inferring memory are sorted to select incrementally the scan elements that will be eventually mapped to the scan register. The selection is “integrated”, applying together, for each iteration of the graph’s method, the function to measure and that to improve testability.

The testability measure is a combination of different aspects that contribute to the low testability of the memory elements corresponding to objects in the behavioral description (signals/variables) of the system/ component; these aspects can be modeled by weights in the behavioral S-graph, in addition to the structural testability measures based on feedback cycles in the graph and sequential depth: the relationship degree between nodes modeling memory elements, represented by arc weights, can be reduced by conditions; arc weights determine node weights through topology, finally reflecting the hardness to test the behavioral object corresponding to the node; node weights are further influenced by characteristics that can be determined by an analysis of the description text: restricted value range of an object in a description statement and non-uniform distribution caused by an operation on the object’s values. These measures are compatibilized by algebraic operations and combined to reflect the testability of a node (modeling a variable/ signal). The graph representing the interconnected autonomous modules, separately processed for testability improvement, has analog properties to a testable behavioral S-graph; so the same algorithm is performed at the higher structural level.

The behavioral metrics is not targeted for any particular design for testability improvement technique, so they can be useful for any of them. They can be translated to weights for the nodes of the behavioral S-graph and combined to weights reflecting scan-depth contribution and feedback cycle contribution of the nodes. For partial-scan, the behavioral objects inferring memory are sorted to select incrementally the scan elements that will be eventually mapped to the scan register. Therefore, a metric for testability, integrating the others is defined for the behavioral S-graph nodes representing variables/ signals that will be synthesized to memory elements.

Cycle breaking automatically extends to the greater cycles it is embedded in; so, a cycle list memorizes the representative cycles (that do not include one-another). Only strongly connected components (there exist directed paths between any pair of nodes of the sub-graph) are considered for minimum feedback vertex set, because nodes of different such components can not share a cycle, i.e., there is no cycle outside of the strongly connected components of a directed graph; as usual, the “Divide et Impera et Intellige” strategy is guided by the strength of communication between sub-objects (here strongly connected sub-graphs). Each iteration can be enriched by selecting more nodes, that are independently breaking cycles, reducing sequential depth and improving the testability measures that reflect restricted value range, operation asymmetry and statement reachability:

```
class SGraph ...
List vertexList, arcList;
public: ...
FVS (SGraph dg) : SGraph × List
begin
List cycleList := cycles (vertexList, arcList);
List selVert, vL; List fvs := ∅;
while (cycleList) begin
        selVert := minWeight (cycleList);
        vL := vertexList;
        while (vL) (selVert, vL) :=
                nextIndependentMinWeightVertex
                                        (selVert, vL);
        cycleList := removeCycles (selVert);
        (dg, fvs) := (weight (vL - selVert,
           arcList - arcs (selVert)), ins (fvs, selVert));
        end;
return (dg, fvs)
end
```

CONCLUSIONS

Formalizing hierarchical descriptions, we create a theoretical kernel that can be used for systematic hardware/ software co-simulation. A new perspective on simulation is gained by unifying representation for design and verification, separating it from the general methods of multi-hierarchical operation; this will permit theoretical development, as well as efficient application to hierarchically built interfaces for hardware/ software systems. As an aid to keep in our formalization process close to real problems, we intend to propose and develop an integrated programmable system for design and verification of hardware/ software systems.

REFERENCES

Booch, G., 1991, *Object-Oriented Analysis & Design*, Benjamin/ Cummings Publishing Company.

Bibel, W. et al., 1993, *Wissensrepräsentation und Inferenz,* Vieweg.

Cheng, K-T., V.Agrawal, 1990: "Partial Scan Method for Sequential Circuits with Feedback", IEEE Transactions on Computers, vol.39, no.4, pp.544, April.

Fleury, H., C.Aktouf, C.Robach, 1999, "A Practical Technique for Scan Insertion at Behavioral Level", *Proceedings of the International Test Synthesis Workshop*.

Gajski, D. et al., 1994, *Specification, and Design of Embedded Systems*, Prentice-Hall.

Kumar, S. et al., 1996, *The Codesign of Embedded Systems*, Kluwer Academic Publishers.

Stroustrup B., 1997, *The C++ Programming Language (3rd edition),* Addison-Wesley.

BIOGRAPHY

Dr. Tudor NICULIU, born 1961 in Bucharest, graduated Electronics (1985, Technical University in Bucharest) and Mathematics (1994, University in Bucharest). 1995 he obtained the PhD degree in Microelectronics from the Technical University in Bucharest. He is Associate Professor at the Electronics Department of the Technical University in Bucharest, teaching courses on Programming techniques, AI in simulation, Formalization of hardware & software design. He worked as guest researcher at the Technical Universities in Braunschweig (1991-1992), Darmstadt (1994, 1995, 1997, 1998), Valence-Grenoble (1999-2000) as well as at the University of Southern California in Los Angeles (1996).