Hierarchical Approach for Hardware/ Software Systems

Tudor NICULIU Sorin COTOFANA* Anton MANOLESCU
Universitatea «Politehnica» Bucuresti, Facultatea de Electronica si Telecomunicatii
Bd. Iuliu Maniu 1-3, 77202 Bucuresti, Romania,
E-mail: tudor@messnet.pub.ro
(*)Delft University of Technology, Faculty of Electrical Engineering
Mekelweg 4, 2600 GA Delft, The Netherlands

Competent design 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.

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, like interfaces needed for communication between different domains, used to define and realize heterogeneous systems, imposes a hierarchical approach. 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 [1] and the symbolization paradigm [2]. 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 to handle communication between different domains implied by hardware/ software systems, that combine dynamic objects (handled in software) with 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++ [5] extended for parallelism by synchronization constructs [4];
- 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 should assist/ automate 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.
```

Figure 1: BADCCT algorithm

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: 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: 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:

(U, {H_i Î S_h}) - universe,
structured by different hierarchies H_i,
S_h - set of hierarchies defined on universe **U**:

H = (Rel_eq, {($Level_j$,$Structure_j$): j Î S_l},
Rel_ord, {A_j: j Î S_l}) - generic hierarchy:
S_l - set of hierarchy levels,
Rel_eq - equivalence relation
divides the universe in levels,
$Structure_j$ - structure defined on level **j**,
Rel_ord - order relation (total)
defined on the set of hierarchy levels,
A_j Î {(x,y): xÎ $Level_{j-1}$, y Î $Level_j$, jÎ S_l }
- relation of abstraction.

For example: The classical activities in complex systems simulation [3], 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:

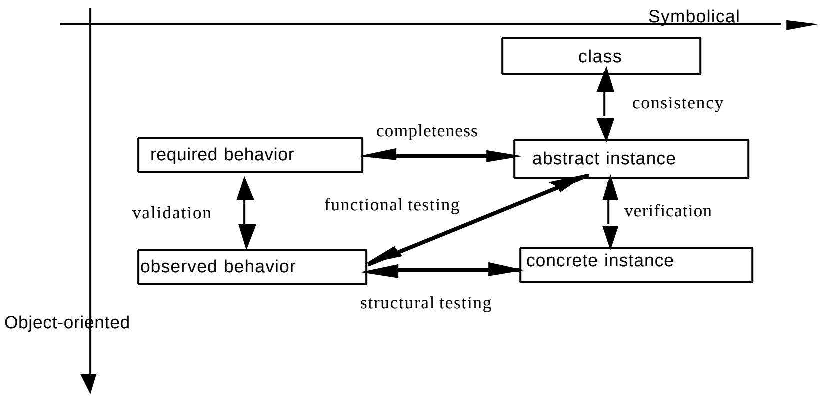

Figure 2: Hierarchical Simulation

Abstract Performance Evaluation of Dynamic Parallel Systems

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. Let us assume that the total execution time for the system's current partition should 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,
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 + const.
```

To provide an answer to this problem we need to construct a directed a-cyclic graph (DAG) 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 DAG (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, DAG - 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 can be described by:

- 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.

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

1. G.Booch, *Object-Oriented Analysis & Design*, Benjamin/ Cummings Publ. Co., 1991.
2. W.Bibel et al., *Wissensrepräsentation und Inferenz*, Vieweg, 1993.
3. D.Gajski et al., *Specification, and Design of Embedded Systems*, Prentice-Hall, 1994.
4. S.Kumar et al., *The Codesign of Embedded Systems*, Kluwer Academic Publ., 1996.
5. B.Stroustrup, *The C++ Programming Language (3rd edition)*, Addison-Wesley, 1997.