# Object-Oriented Technology Transfer to Multiprocessor System-Level Synthesis[†]

Pao-Ann Hsiung
Institute of Information Science
Academia Sinica, Taipei, Taiwan.

Trong-Yen Lee and Sao-Jie Chen
Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan.

**Abstract**

*Technology transfers between software and hardware engineering date back to a decade and a half. Object-oriented technology from software engineering is one such successful transfer to hardware design. There is a natural correspondence between object-oriented concepts and hardware design. The work presented in this paper extends the basic application of object-oriented technology to system-level synthesis such that not only system modeling uses object-oriented technology, but the synthesis process itself is object-oriented. The basic object-oriented structures required for synthesis are defined. How designs can be reused by storing them in a design database and then retrieving them is explained. A simple implementation along with application example illustrate how object-oriented technology increases component design manageability, enforces synthesis efficiency, and saves design time and effort through the reuse of complete subsystems.*

## 1 Introduction

One successful technology transfer from software engineering to hardware design is the *object-oriented* (OO) *design paradigm* which first manifested itself in the form of system decomposition into modules, information hiding [13], and program families [14]. This success can be attributed to the natural concept of perceiving a hardware component as an individual class with characteristics and operations [10]. In fact, we believe that the decomposition of a system into objects is better defined and more explicit in hardware than in software because a hardware system can be easily decomposed into components, while conventional software programs are often structured using procedures that are more *control-oriented* rather than *object-oriented*. As pointed out by Smith et al. [17]: "*Parnas' concepts of information hiding and design families may contribute to reducing the cost of design development and maintenance*," which was technically proved by the recently proposed *Performance Synthesis Methodology* (PSM) [8], an object-oriented system-level design methodology for multiprocessor (MP) systems.

System-level synthesis is the process of automatic transformation from a set of system specifications including architectural and performance requirements into a high-level architecture consisting of a description of the various components such as processors, memory, and the number of each component used. For example, the MICON system [1, 6], uses a *hierarchical select-and-interconnect* methodology for system-level synthesis. System-level synthesis uses *off-the-shelf building blocks* [18], which are similar to reusable library components in software engineering. These building blocks can be modeled as classes of objects and then hierarchically classified based on the relationships between the classes. In MICON, the parts from the component library are abstracted and organized into a functional hierarchy which is a directed, acyclic graph, leaf-nodes correspond to available physical parts and internal nodes are abstract parts.

[†]This research was supported by the National Science Council, Taipei, Taiwan under grant NSC 86-2221-E002-066.

### 1.1 Object-Oriented Technology and Hardware Design

Several concepts of object-oriented design in software engineering can be applied to hardware synthesis. Some important concepts are described below. In the following, we assume that a class is an individual entity as in the OO technology and it contains attributes which may be data members or function members.

*Class Encapsulation*: Parnas' concept of information hiding or encapsulation tries to hide the portion of design that is likely to change in the future by modularization and restricting access to the internal structure through specific functions. This is similar to a hardware module. The data members of a class can be used to model the static characteristics and dynamic states of a hardware component. Similarly, the function members of a class can be used to describe the pins and module interactions of a hardware component.

*Class Inheritance*: Though *inheritance* was introduced in the object-oriented technology, yet it has been used by hardware designers since a long time. Hardware modules of the same type or functionality often differ in only a few aspects either structurally or behaviorally. Inheritance avoids the undesired replication of component information and allow quick transformation from existing classes to classes of new components.

*Class Instances or Objects*: The instantiation of a class involves a valuation of its data attributes such that after each attribute is assigned a specific value, we will have a class instance or object. Different valuations lead to different objects. This corresponds to the physical components that are available for system design.

*Class Relationships*: Two classes may be related by a relationship such as *aggregation* (assembly-component), *generalization* (superclass-subclass), and *dependence* (association). In the hardware perspective, aggregation corresponds to the composition of sub-parts into a larger module, which is what designers do when creating a 64-bit adder from two 32-bit adders. Generalization corresponds to the functional classification of hardware components. Dependence is related to how a particular component depends on another component for mutual interactions and synchronizations.

*Hierarchical Structure of Design Families*: Based on class relationships, a hierarchy of classes can be constructed. For example, a parallel computer design is an aggregation of a memory subsystem, a system interconnection network, and a processing subsystem, all of which may further be an aggregation or generalization of other component parts modeled as classes. This hierarchy represents a reusable and useful library of components that can be used for synthesis.

Section 2 lists some previous related work. A more complete account of how a system can be modeled using object-oriented techniques is presented in Section 3. Section 4 describes the manipulation of designs, their reuse, basic object-oriented structures used in system-level synthesis, and how designs are stored, retrieved, and reused from a design database. Section 5 illustrates the implementation of our concepts and the application of OO techniques. Section 6 concludes the paper with some future work.

## 2 Previous Related Work

Modularization in VLSI design date back to 1983, when a research group called VMOD (Modular VLSI) [2] explored the similarities and dissimilarities between the techniques of software engineering and hardware design. The group concluded that despite their certain dissimilarities, two concepts of software change management, namely Parnas' information hiding and families of design could profitably be extended to VLSI design. This triggered researches in the application of modularization to VLSI design such as the Software Cost Reduction (SCR) project [15], Gross' abstract interface specification of VLSI designs [5], and ADAS, a software/hardware codesign tool [4]. Besides these earlier work, only recently has this field of research gained attention again. OO is now being incorporated into hardware description language such as VHDL [3]. By making use of a behavioral hierarchy, some system-level synthesis specification languages such as SpecCharts [12] also have the flavor of the OO technology. MICON Synthesizer Version 1 [6] uses a functional hierarchy for system-level design automation. OO has also been combined with formal modeling techniques such as Petri Nets for system modeling [11] and synthesis [9].

Although researches on the application of OO technology to hardware design have been going on and off since the 1980's, yet only recently an actual system-level synthesis methodology, called *Performance Synthesis Methodology* (PSM) [8], based on OO techniques was proposed. Following this, OO has also been used recently in the formal modeling of the system-level synthesis methodology resulting in an extended high-

level Petri Net called *Multi-token Object-oriented Bi-directional net* (MOBnet) [9]. These recent progresses show that efforts in this promising direction will be quite fruitful in the near future.

# 3 Object-Oriented System Model

## 3.1 Object-Oriented Structure

As far as notations and terminologies are concerned, we basically follow Rumbaugh's *Object Modeling Technique* (OMT) [16] which makes a clear distinction among the object model, the dynamic model, and the functional model. By modeling each component, either abstract or physical, as a class with relationships to other classes, a *Class Hierarchy* (CH) can be constructed, which is similar to Parnas' hierarchical structure of design families and Rumbaugh's Object Model in OMT. We distinguish the classes representing components into three kinds of nodes, namely *Aggregate node* (A-node), *Generalized node* (G-node), and *Physical node* (P-node). This distinction is made based on the relationship a class has with its child classes, if any. An aggregate node is an assembly class representing something which is the *whole* in a "*whole-part*" relationship. A generalized node is a superclass which is the parent in an "*is-a*" relationship. A physical node is a leaf node in the Class Hierarchy and represents some available physical component that can be used directly for design. This classification of classes into three kinds of nodes allows easy selection of appropriate design actions at each node which will be discussed in subsection 3.3.

A generic class has attributes including data members and function members. We classify the data members of a class into *specifications*, *pre-design characteristics*, and *post-design characteristics*, where a specification is a requirement, it may be a relation between several characteristics, a pre-design characteristic is one whose value is known before design and a post-design characteristic is one whose value is known only after design. Some function members are discussed in subsection 3.3.

## 3.2 Object-Oriented Relationships

Three kinds of relationships serve as guidelines for design automation, namely *aggregation*, *generalization*, and *dependence*. Aggregation denotes the "whole/part" relationship in which a component class is a "part-of" a class representing the whole assembly. Generalization denotes the relationship between a class and its one or more refined versions. Often the design characteristics of two adjacently-connected components in an MP system are interrelated such that the synthesis of one affects the other. This relationship is modeled as the dependence relationship. Dependence is further classified into *absolute* and *relative*. Absolute dependence is a dependence between the *specification* of one component and the *post-design characteristic* of another component such that the former component must wait for the latter to be completely synthesized before it can begin synthesis. Relative dependence is a dependence between the *specifications* of two components such that the synthesis is possible only when the dependent specification of the component to be synthesized has a value assigned by querying the other component. For example, a Memory class is said to be absolutely dependent on a CPU class because the memory access time ($m$) can be expressed in terms of the processor cycle time ($p$) as $m = k \times p$, where $k$ is a constant and it is assumed that $m$ is a specification and $p$ a post-design characteristic. Further, a Cluster Control Unit (CCU) and a System Interconnect (SI) are relatively dependent because the CCU *data transfer rate* ($d_{\text{CCU}}$) and the SI *data transfer rate* ($d_{\text{SI}}$) are related as $d_{\text{SI}} = c \times d_{\text{CCU}}$, where $c$ is a constant and it is assumed that both $d_{\text{SI}}$ and $d_{\text{CCU}}$ are the required *specifications.*

Each type of relationship allows us to perform different synthesis actions, thus the synthesis process is guided by the relationships. When encountered with an aggregation relationship, the class which is an aggregation of other classes can be synthesized by composing one or more instances of its child classes. When a generalization relationship is reached, the superclass representing a generalization of sub-classes can be implemented by selecting one or more of its child classes. This is the design-space exploration step in system-level synthesis. On encountering a dependence relationship, a class checks with the classes having dependence relationships with it to request for or submit data values before any other synthesis actions are taken. Thus, the dependence relationship has the highest priority among all relationships that a class may have with other classes.

### 3.3 Object-Oriented Operators

As mentioned in the previous subsections, different types of classes and relationships allow different synthesis actions, we use object-oriented operators to represent these actions. The target of these operations are the classes in the Class Hierarchy. There are three operators, namely *iterator*, *generator*, and *updator* corresponding to the three relationships aggregation, generalization, and dependence, respectively.

"Iterator" is the actual synthesis operator, it is used at an A-node for synthesizing this aggregate class of components. Based on the specifications satisfaction of an A-node, the iterator *iterates* through its child classes selecting some of them to be interconnected into the aggregate. "Generator" is the design-space exploration operator, it is used at a G-node for implementing this generalized class of components. Based on the specifications satisfaction of a G-node, the generator *generates* a sequence of classes ordered in the preference of their feasibility in implementing the G-node. This order of feasibility preference may be a simple cost-based heuristic as in PSM [8] or a fuzzy comparison of specifications as in ICOS [7]. "Updator" is the query operator since its main job is to query others for specification values. This operator is quite essential for hardware consistency and feasible integration. The updator operator is used by a class whenever it has a dependence relationship with another class.

### 3.4 Class Hierarchy

The above object-oriented structure dealt with how a component can be modeled as a class and how classes are divided into three types; object-oriented relationships described how classes may be related and how relationships be used to guide synthesis; and object-oriented operators described what kinds of actions can be taken at different nodes. By modeling all the components of a system as classes, we are thus able to construct a hierarchy of classes called *Class Hierarchy*, where classes are inter-related by the object-oriented relationships. This hierarchy is usually constructed apriori just like how a software library is constructed for future use. An example of Class Hierarchy for a hierarchical parallel computer system is given in Fig. 1. The purpose of this hierarchy is to serve as a framework in which synthesis proceeds. The main concept of object-oriented synthesis can be briefly defined as follows: "Starting from the root node of a Class Hierarchy, which represents the computer system to be designed, we *traverse* down the hierarchy using class relationships as guidelines, choosing appropriate operators at each node, performing corresponding actions, and synthesizing or implementing components along the hierarchy."

## 4 Design Manipulation and Reuse

Given a Class Hierarchy representing the static structure of a target machine, this section discusses how the various component designs represented by classes in the hierarchy can be manipulated dynamically during synthesis. An important concept of object-oriented technology, namely *class reuse*, is technically applied to hardware design such that complete subsystems can be reused, thus saving design cost, time, and effort.

Design manipulation and reuse in an object-oriented environment is best discussed within the framework of an actual system-level synthesis methodology. We use *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology [7], a newly proposed methodology for the synthesis of multiprocessor systems, which is discussed briefly as follows. ICOS works in a fully object-oriented environment. Its main phases are *Specification Analysis*, *Concurrent Design*, and *System Integration*.

An ICOS designer inputs system-level specifications such as the type of processor, the memory organization, and the system interconnection scheme. After going through the three design phases, ICOS outputs a synthesized MP consisting of system-level architecture descriptions. ICOS methodology uses several techniques including object-oriented modeling, design reuse through intelligent learning, fuzzy design-space exploration, and concurrent synthesis to make the synthesis process more efficient and feasible. As shown in Fig. 2, the main phase, namely Concurrent Design, is further divided into four steps: *specification update*, *component reuse by learning*, *component synthesis*, and *design storing*.

When a designer is synthesizing a system using ICOS, the OO design environment must provide certain structures for the methodology to use for dynamically manipulating the classes, instead of only referring

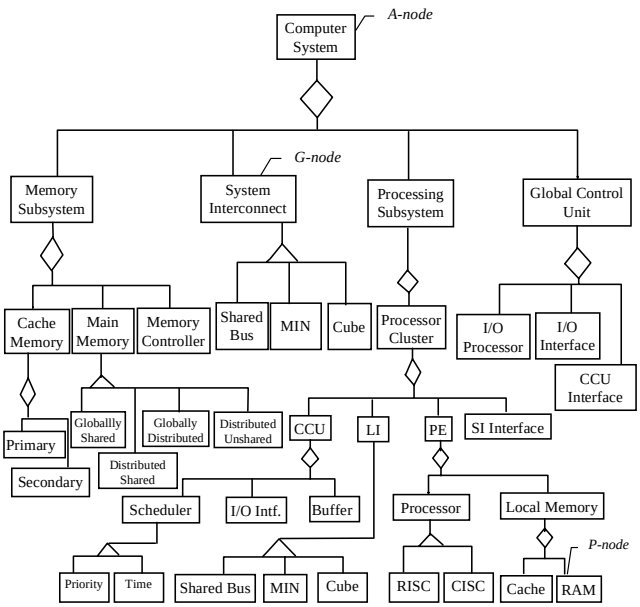


**Figure 1. Class Hierarchy**

to a static Class Hierarchy. In this paper, we assume the *concurrent* synthesis scheme similar to that in ICOS, where more than one component can undergo synthesis at the same time. Concurrently synthesized components can communicate specifications through the dependence relationships in CH. Besides CH, the OO design environment must be able to globally maintain a structure to broadcast the current design status so that components that are concurrently undergoing synthesis may know exactly what the current design status is. Such a broadcast structure is the *Design Hierarchy* as described in subsection 4.1. The OO environment must also provide a sequencer for components to be designed, so that the synthesis methodology knows which is the "next" component to begin synthesis. This is implemented as a queue structure called the *Design Queue* as described in subsection 4.2. Another structure that must be implemented in an OO design environment is the design database used for design reuse. This is called the *Learning Hierarchy* (LH) and described in subsection 4.4.

### 4.1 Design Hierarchy

In order to keep track of all the design alternatives generated during synthesis, a hierarchy of currently synthesized classes, called *Design Hierarchy* (DH), is maintained. It is an *implementation* of the Class Hierarchy, that is classes in CH are substituted by *real* designs with specifications. For example, Fig. 3 depicts a completed design alternative consisting of a shared-memory multiprocessor architecture with 1024 processors interconnected by a generalized-cube multistage interconnection network, and 8 GB of main memory. Besides representing the current state of synthesis, it may be used for various other purposes.

1. *Information Query*: When a component under design needs information related to the current design architecture, they can be answered by refering to DH. For example, an inquiry could be: "Is the current design using any secondary level cache?"
2. *Synthesis Rollback*: There may arise a situation in concurrent synthesis where a particular component cannot be synthesized under the currently derived specifications, at this point of synthesis, a rollback of design steps could possibly alter or re-design some previously designed components such that the specifications related to the unsynthesizable component are relaxed and synthesis can proceed further. Rollback may also propagate from an unsynthesizable component upwards in the Design Hierarchy.
3. *Design Completion Check*: Design Hierarchy can also indicate when a design alternative is complete for further processing such as simulation and performance evaluation.

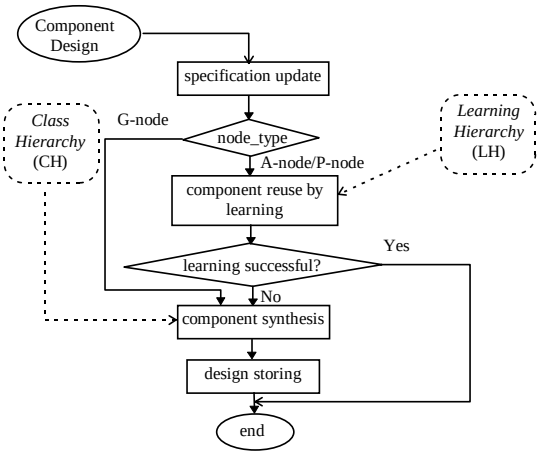


**Figure 2. ICOS Concurrent Design**

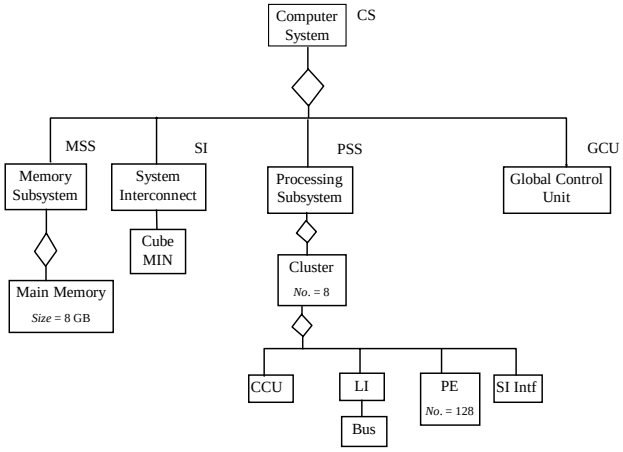


**Figure 3. Design Hierarchy**

### 4.2 Design Queue

Design Hierarchy stores the components which have already been synthesized, but there is a stage in the design life-cycle where a component is already selected or *ready* for synthesis, but has to wait for its turn. At this stage, we need a queue structure that holds the components which are ready for synthesis. We call such a queue structure Design Queue (DQ). After removing an A-node from the front of the queue, it is synthesized into several component classes which have then to be appended to DQ. After removing a G-node from the front of the queue, it is implemented into one or more specialized classes which have then to be appended to DQ. Whenever a P-node results from some component synthesis process, it is not appended to the queue, but instead it is instantiated into actual physical instances.

### 4.3 Design State

DH and DQ are global structures visible to all components. When a component is in DH or DQ, it is supposed to be in a "passive" state and when it is outside these structures, it is in an "active" state. The reason for distinguishing between passive and active states mainly lies in the fact that a component represented by a class remains in an *idle* condition in DH or DQ, whereas it *actively seeks* to synthesize itself using appropriate operators and relationships as guidelines while outside DH or DQ. Figure 4 illustrates how a component can transit from passive to active and then back to a passive state. While active, a component first updates whatever specifications are needed, then it tries to reuse previously designed components. If no such components exists, it goes on to synthesize itself. At this point, it may encounter a deadlock situation where it cannot complete synthesis due to some unsatisfiable specification, it enters a rollback state which leads to the earlier specification update state. If a component is reused or synthesized successfully, it enters DH and remains in the passive state.

### 4.4 Design Reuse

Class "*reuse*" in the object-oriented technology often means how undesired replication is avoided through the repetitive use of the same class. In this paper, reuse is distinguished into two kinds: *current* reuse and *future* reuse. Current reuse is defined as the repeated use of a class for the same design in the current synthesis. For example, a hierarchical parallel computer architecture consisting of a group of processor clusters, may have memory both *locally* in each cluster as well as *globally* in the system; hence the Memory class can be reused for cluster and system by just re-assigning some attribute values such as the memory size. Future reuse is defined as the storing of a synthesized component in some design database for retrieval and use in future syntheses. For example, when a processing subsystem, satisfying certain given architecture and

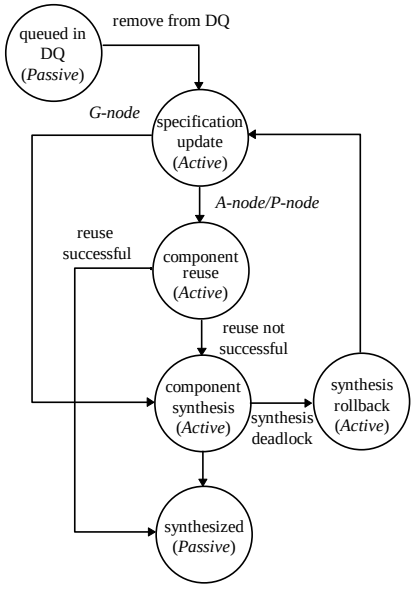


**Figure 4. Class State Transition Diagram**

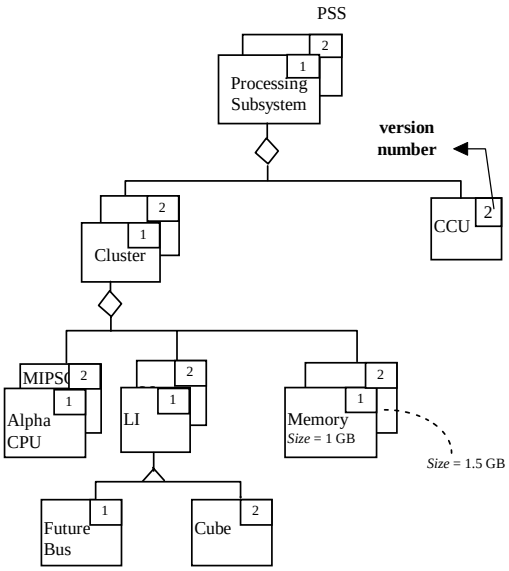


**Figure 5. Learning Hierarchy**

performance specifications, is synthesized, it may be stored so that whenever similar specifications are given, that designed instance may be reused. Current reuse can be accomplished by traversing the Class Hierarchy and identifying similar components. Future reuse needs a design database for storing synthesized component designs. One implementation of such a design database is the *Learning Hierarchy* as described below.

#### 4.4.1 Learning Hierarchy and Object Storing

Learning Hierarchy (LH) is a hierarchy of synthesized objects instead of abstract classes as in the Class Hierarchy. Its main purpose is to store designs learnt from previous synthesis experiences for future reuse. An example of LH is given in Fig. 5. A *design version* of a component is defined as one instance of the component design satisfying a set of valuations for the specifications of all constituent objects. For example, a Memory Subsystem class, an aggregate of main memory, cache memory, and memory controller, may have a design version which was synthesized satisfying specifications of 20 GB main memory with 6 ns access time, 1 MB cache memory with snoopy-write-invalidate coherence protocol and a memory controller costing $ 2,000. Each component from CH may have one or more design versions stored in LH. Whenever a component, which may be complete subsystem, is synthesized, it is stored in LH as follows:

(a) Represent the component as a sub-tree of the Class Hierarchy.
(b) Each constituent object of the sub-tree will have a *unique* name as specified in CH, but the *same* version number, which is different from all other previously assigned version numbers.
(c) All kinds of specifications, including architectural requirements, performance constraints, and synthesis-related restrictions must be included with the constituent objects along with the corresponding value assignments.
(d) Integrate the component into the current LH by storing each constituent object in its corresponding location in the hierarchy.

The relationships among the Class Hierarchy, Design Hierarchy, and Design Hierarchy are illustrated in Fig. 6.

#### 4.4.2 Object Searching and Retrieval

Learning Hierarchy is maintained as one *single* hierarchy instead of *multiple* hierarchies for ease of searching and retrieval. A single hierarchy allows synthesis systems to traverse down the hierarchy searching in a

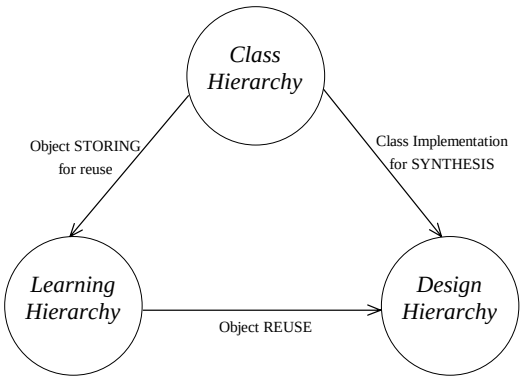


**Figure 6. Relationships among the CH, DH, and LH**

```
generic.update_spec(){
 if (generic.type==P-node) return 0;
 for each spec ∈ generic.SPEC do
  i = 0;
  while(spec=NULL) do
    if(last_dep_class()) break;
    query_spec(spec, dep_class[i++];
  endwhile
  if(spec==NULL) query_user(spec);
 endfor
}
```

```
generic.reuse(){
 switch(generic.type){
  case "A-node": if(generic.synthesized) fsgl();
                 else return 0; break;
  case "G-node": if(generic.synthesized) egl();
                 else return 0; break;
 }
}
```

```
iterate(){
 for each generic.child do
  if required(generic.child){
    AddDH(generic.child);
    if((generic.child).type != P-node)
        AppendDQ(generic.child);
    else instantify(generic.child); }
  explore_design_space();
 endfor
}
```

```
generic.synthesize(){
 if(generic.type==A-node)
  synthesize() = iterate();
 else if(generic.type==G-node)
  synthesize() = generate();
 else synthesize() = NoOp();
}
```

```
generate(){
 for each generic.child do
  if(acceptable(generic.child) {
    AddDH(generic.child);
    if((generic.child).type != P-node)
      AppendQ(generic.child);
    else { instantify(generic.child); dse(); }
  }
 endfor
}
```

**Figure 7. Some Class Functions**

depth-first search manner for the node location of the component under design. Once the location is found, an appropriate design version of the component is selected. This selection depends on the synthesis methodology. For example, ICOS uses a fuzzy specification-guided learning process [7]. After selecting a design version, using the version number as a *search key*, LH is then traversed in a breadth-first search manner so that all constituent objects of that design version can be found.

The multiple-version single-hierarchy structure of LH also allows one to directly reuse a partial design of some stored design version because each constituent object has its own set of specifications and a valuation. For example, suppose a Processing Subsystem class has a design version stored in LH, and currently we need to synthesize a Cluster Interconnection class, which is part of the Processing Subsystem and satisfies all our specifications, thus it can be directly reused. This allows flexible reusing of all parts of a design version.

# 5 Application Example

The application of object-oriented technology to system-level design automation as discussed in Sections 3 and 4 was implemented using the C++ object-oriented programming language. Some of the generic class functions are described in Fig. 7.

## 5.1 Application Example

Our target system is specified to be a tightly-coupled shared-memory multiprocessor architecture with a maximum cost of $ 12,000, at least 1 GB main memory and 1024 processors interconnected using a multistage interconnection network. The design steps are given in Table 1 and the intermediate status of DH are given in Fig. 8.

Starting from the root node, the target system is iteratively synthesized by traversing down the Class Hierarchy towards the leaf nodes (P-nodes). The root node in this example is Computer System (CS) which is an A-node, the relationship it has with its child classes is aggregation, hence we use the "iterator" to *synthesize* CS into Memory SubSystem (MSS), System Interconnect (SI), Processing SubSystem (PSS), and Global Control Unit (GCU), all of which are appended to the Design Queue (DQ). This completes step (a) in Table 1. Now, in step (b), MSS is removed from the front of DQ. Though MSS is an A-node, there is a memory subsystem in LH that satisfies all the specifications of MSS, and that subsystem is *reused* for MSS

and thus no component is appended to DQ in this step. Next, in step (c) SI which is a G-node is removed from DQ and a design-space exploration (DSE) is performed using the "generator" at SI, which results in the two alternative multistage interconnection networks (MIN): Cube and Omega. These MINs are physically available components so they are not appended to DQ. They are, in fact, instantiated into actual usable objects in ICOS. In step (d), PSS is synthesized into Cluster. In step (e), GCU is synthesized by reusing an acceptable design version from LH. In step (f), Cluster is synthesized into CCU, LI, and PE. In step (g), CCU is synthesized by reusing a design version from LH. Steps (h) and (i) complete the synthesis process by synthesizing LI and PE through DSE and reuse. This synthesis process terminates when DQ is empty.

**Table 1. Illustrative Example Design Steps**

| Step | Node | CT | Op | Action | DQ Status |
|---|---|---|---|---|---|
| (a) | CS | *A* | itr | synth | {MSS,SI,PSS,GCU} |
| (b) | MSS | *A* | itr | synth | {SI,PSS,GCU} |
| (c) | SI | *G* | gen | DSE | {PSS,GCU} |
| (d) | PSS | *A* | itr | synth | {GCU,Cluster} |
| (e) | GCU | *A* | itr | reuse | {Cluster} |
| (f) | Cluster | *A* | itr | synth | {CCU,LI,PE} |
| (g) | CCU | *A* | itr | reuse | {LI,PE} |
| (h) | LI | *G* | gen | DSE | {PE} |
| (i) | PE | *A* | itr | reuse | {} |

CT = Class Type, itr = iterator, gen = generator, and synth = synthesis

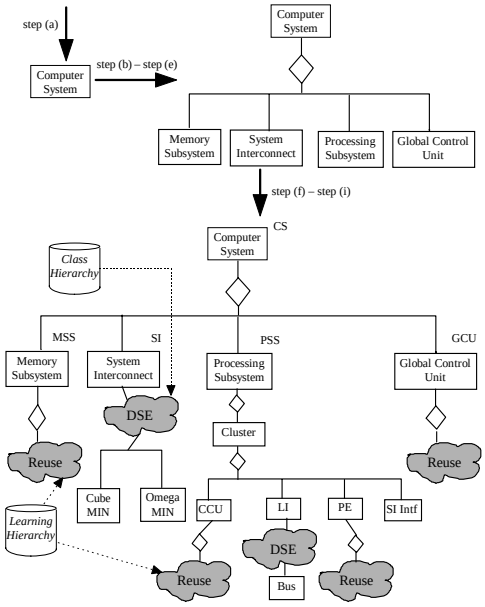


**Figure 8. An Illustrative Example (intermediate DH status)**

The above example shows how classes are synthesized or implemented in an object-oriented design environment. It also shows how the three hierarchies CH, DH, and LH are used for synthesis. A more elaborate synthesis methodology based on the object-oriented design environment presented in this paper can be found in [7].

## 6 Conclusion and Future Work

The intuitive correspondence between object-oriented technology in software engineering and hardware design in system-level synthesis was discussed, explored, and realized in a working object-oriented design environment. Besides the static modeling of components, it was shown how classes can be dynamically manipulated during synthesis, what kind of structures or hierarchies are required for synthesis, and how designs can be reused in the current synthesis as well as for future syntheses. Our initial concepts were realized into an actual environment by implementing it and running examples. PSM [8] and ICOS [7] are two methodologies that have been based on this environment. MOBnet [9] was a recently proposed theoretical framework for such an OO design environment. All these go to show that our OO design environment is conceptually, technically, and theoretically applicable to system-level synthesis.

Future work in this direction will try to incorporate some further OO principles and engineering wisdoms. Hardware-software codesign is a field where OO will certainly be a productive technology as both hardware and software components can be modeled as classes which are implementation independent.

## References


[1] W. P. Birmingham, A. P. Gupta, and D. P. Siewiorek. The MICON system for computer design. In *Proc. 26th ACM/IEEE Design Automation Conference*, pages 135–140, 1989.

[2] F. R. Jr. Brooks, R. R. Gross, and L. S. Heath. Transfer of software methodology to VLSI design. Technical Report TR 84-007, Univ. of North Carolina, Chapel Hill, 1984.

[3] M. J. Chung and S. Kim. An object-oriented VHDL design environment. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 431–436, 1990.

[4] G. A. Frank, C. U. Smith, and J. A. Cuadrado. An architecture design and assessment system for software/hardware codesign. In *Proc. 22nd ACM/IEEE Design Automation Conference*, June 1985.

[5] R. R. Gross. *Using software technology to specify abstract interfaces in VLSI design*. PhD thesis, Univ. of North Carolina at Chapel Hill, June 1985. Dept. Computer Science Tech. Rep., TR-85-017.

[6] A. P. Gupta, W. P. Birmingham, and D. P. Siewiorek. Automating the design of computer systems. *IEEE Trans. on CAD*, 12(4):473–487, April 1993.

[7] P.-A. Hsiung, C.-H. Chen, T.-Y. Lee, and S.-J. Chen. ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems. *ACM Trans. on Design Automation of Electronic Systems*, 3(2), April 1998.

[8] P.-A. Hsiung, S.-J. Chen, T.-C. Hu, and S.-C. Wang. PSM: An object-oriented synthesis approach to multiprocessor system design. *IEEE Trans. on VLSI Systems*, 4(1):83–97, Mar. 1996.

[9] P.-A. Hsiung, T.-Y. Lee, and S.-J. Chen. MOBnet: An extended Petri net model for the concurrent object-oriented system-level synthesis of multiprocessor systems. *IEICE Trans. on Information and Systems*, E80-D(2):232–242, Feb. 1997.

[10] S. Kumar, J. H. Aylor, B. W. Johnson, and Wm. A. Wulf. Object-oriented techniques in hardware design. *IEEE Computer*, 27(6):64–70, June 1994.

[11] Y. K. Lee and S. J. Park. OPNets: An object-oriented high-level Petri net model for real-time system modeling. *Journal Systems Software*, 20:69–86, 1993.

[12] S. Narayan, F. Vahid, and D. D. Gajski. System specification with the SpecCharts language. *IEEE Design and Test of Computers*, pages 6–13, Dec. 1992.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[14] D. L. Parnas. On the design and development of program families. *IEEE Trans. on Soft. Eng.*, SE-2(1):1–9, Mar. 1976.

[15] D. L. Parnas. The modular structure of complex systems. *IEEE Trans. on Soft. Eng.*, SE-11(3):259–266, Mar. 1985.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, 1991.

[17] C. U. Smith and R. R. Gross. Technology transfer between VLSI design and software engineering. *Proc. of the IEEE*, 74(6):875–887, June 1986.

[18] J. R. Tobias. LSI/VLSI building blocks. *IEEE Computer*, 14(8):83–101, Aug. 1981.