ICOS: An Intelligent Concurrent Object-Oriented Synthesis Methodology for Multiprocessor Systems

PAO-ANN HSIUNG
Academia Sinica
and
CHUNG-HWANG CHEN, TRONG-YEN LEE, and SAO-JIE CHEN
National Taiwan University

The design of multiprocessor architectures differs from uniprocessor systems in that the number of processors and their interconnection must be considered. This leads to an enormous increase in the design-space exploration time, which is exponential in the total number of system components. The methodology proposed here, called *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology, makes feasible the synthesis of complex multiprocessor systems through the application of several techniques that speed up the design process. ICOS is based on *Performance Synthesis Methodology* (PSM), a recently proposed object-oriented system-level design methodology. Four major techniques: object-oriented design, fuzzy design-space exploration, concurrent design, and intelligent reuse of complete subsystems are integrated in ICOS. First, object-oriented modeling and design, through the use of object-oriented relationships and operators, make the whole design process manageable and maintainable in ICOS. Second, fuzzy comparison applied to the specializations or instances of components reduces the exponential growth of design-space exploration in ICOS. Third, independent components from different design alternatives are synthesized in parallel, this design concurrency shortens the overall design time. Lastly, the resynthesis of complete subsystems can be avoided through the application of learning, thus making the methodology intelligent enough to reuse previous design configurations. Experiments show that all these applied techniques contribute to the synthesis efficiency and the degree of automation in ICOS.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CA); I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition*; *Analogies*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction* (e.g., natural, rule-base)

General Terms: Design

This research was supported by the National Science Council, Taipei, Taiwan, under Grant NSC 85-2623-D002-015.

Authors' addresses: P.-A. Hsiung, Institute of Information Science, Academia Sinica, Taipei, Taiwan; C.-H. Chen, T.-Y. Lee, and S.-J. Chen, Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan.

Additional Key Words and Phrases: Concurrent object-oriented system-level synthesis, fuzzy design-space exploration, learning

1. INTRODUCTION

Synthesis is the process of automatic transformation from a set of logically higher level design specifications into a logically lower level design architecture. Corresponding to the levels of design details, we have different levels of synthesis, such as logic level, register-transfer level (RTL), algorithmic or high level, and system level. At the logic level of synthesis, the designer inputs gate-level design specifications and obtains physical-level architecture. At the RTL, register-transfer specifications are given and gate-level results obtained. At the algorithmic or high level, an algorithm describing a particular behavior is synthesized into an RTL design architecture. Finally, at the system level of synthesis, a description of system behavior or a set of system-level specifications is transformed into an architectural description of the system such as the processor type, the memory organization, and the system interconnection network.

With technology advances, the complexity of computer system architecture has increased to the extent that synthesis tools that automate the design process, if not indispensable, are becoming a necessity for meeting the ever-shortening time-to-market requirement. Design *methodologies* for uniprocessor systems are quite mature, but system-level synthesis *tools* automating the design of such systems are still under research and development. Compared to uniprocessor systems, multiprocessor (MP) systems present many more design trade-offs and challenges; hence, the design automation of MP systems is more imperative. Recently, *Performance Synthesis Methodology* (PSM) [Hsiung et al. 1996] was proposed as a successful methodology for MP systems. The architectures considered in this article are also parallel systems that include both tightly coupled multiprocessors and loosely coupled multicomputers.

Multiprocessor system-level synthesis is a design automation process where starting from a set of system descriptions, performance constraints, and a cost bound, a multiprocessor architecture is synthesized by determining the number and type of processors used, the processing cluster organization, the type of system interconnection, and the amount of memory with its logical and physical organization. A multiprocessor synthesis system is different from currently available uniprocessor synthesis systems because the design of the particular architecture now requires exploring many more design alternatives and performance trade-offs. A uniprocessor system has only one processing element, so currently available uniprocessor synthesis systems need only consider the type of processor and determine the amount of memory to use. However, a multiprocessor system architecture has more than one processor, so we must decide how many processors to use and how to interconnect the processors using some interconnection network; determine the way in which the main memory is organized; and classify cache memory into local, primary, and secondary levels. All of these considerations are critical to the feasibility and performance of the final synthesized architecture, and they are not taken into consideration by uniprocessor synthesis systems. Furthermore, an important aspect of multiprocessor system synthesis is how the workloads are distributed into each processor and the balancing of the processor workloads in order to maximize the system performance. This distribution and balancing of workloads mainly depends on the type and number of processors available, on how the processors are interconnected, and on the design of the global control unit that distributes workloads to each cluster in a hierarchical MP system. All of these factors make the MP system synthesis special and different from the conventional system synthesis.

We define an *object-oriented* (OO) *synthesis* as the design process in which system parts are modeled as object classes interlinked by relationships in a hierarchy of classes and the desired system is synthesized by traversing the hierarchy, selecting appropriate object classes, and instantiating them. The rationale for using OO in synthesizing MP systems can be summarized as follows: First, since a large number of variations of MP systems is possible due to the numerous ways in which processors may be clustered and interconnected and memories may be organized, the *inheritance* mechanism in the OO technology significantly avoids the duplication of design data to a much larger extent than in the conventional non-OO synthesis. Second, MP systems are often modularized through processor clustering for better performance and scalability; such modularizations are very much in coherence with the OO design technology. *Design reuse* plays an important role in modeling identical clusters or modules and in reducing design time. Third, the design of complex MP systems requires a larger design hierarchy than uniprocessor systems, thus the concept of *hierarchical design process* in the OO technology becomes more useful for design management and representation. Overall, the use of OO technology is more advantageous in designing MP systems than in designing uniprocessor systems.

Conventionally, system parts are synthesized in a sequential fashion, for example, in PSM. When more than two components are allowed to be synthesized at the same time, the design process is termed *concurrent synthesis*. In this article, we concentrate on concurrent synthesis and discuss its advantages.

The design space in the synthesis of an architecture having n components is a subspace of $\mathscr{X}^{2n}$, represented as $D = \{\langle(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\rangle \mid x_i, y_i \in \mathscr{X}\}$, where $\mathscr{X}$ is the set of nonnegative integers. Each point in the $2n$-dimensional integer design space represents a design alternative such that for the ith component, x_i is the integer label of a physical instantiation of the component and y_i is the number of x_i used for the final design. The design space size ($|D|$) is thus the total number of design alternatives for a system under design. If the average design time for a single design alternative is τ, then the total time required for design-space exploration of a design consisting of n components, each of which has m specializations, is:

$$T(n) = \tau(m)^n \tag{1}$$

A survey of previous and related work is given in Section 2. Section 3 presents an overview of the concepts and techniques used in ICOS. The article describes the design methodology in Section 4. Implementation, design examples, and experimental observations are covered in Section 5. The last section concludes the article and describes some future work.

2. PREVIOUS AND RELATED WORK

A performance-driven, object-oriented synthesis methodology for the system-level design of multiprocessor systems called *Performance Synthesis Methodology* (PSM) [Hsiung et al. 1996] was recently proposed. Prior to PSM, there were some relevant works on automating the system-level design of computer systems, but they have been developed with a restricted scope of application, for example, the MICON system [Birmingham et al. 1989; Gupta 1993] and the Megallan system [Gadient and Thomas 1993] did not explicitly take the MP features into consideration during system synthesis; Mabbs and Forward [1994] analyzed the performance of MR-1, a clustered shared memory MP, using a queuing model and a lost request model; Chiang and Sohi [1992] evaluated the design choices for a shared-bus MP in a throughput-oriented environment using customized mean-value analysis. Distributed design-space exploration for high-level synthesis systems was discussed by Dutta et al. [1992].

The incorporation of object-oriented concepts into computer-aided synthesis has been discussed mainly in the literature [Lee and Park 1993; Kumar et al. 1994] and implemented in a few hardware description-language-oriented design tools [Chung and Kim 1990]. Reuse of specification through refinement levels has been discussed by Antonellis and Pernice [1995]. An example of learning used in the synthesis of VLSI systems is the Learning Apprentice for VLSI Design (LEAP) [Mitchell et al. 1985]. Besides this example, learning has been rarely used in synthesis. Fuzzy logic has been widely used in VLSI design such as in VLSI placements [Rezaz and Gau 1990; Lin and Shragowitz 1992; Kang et al. 1994], but not in system-level synthesis tools. This article illustrates how learning and fuzzy logic can be used for efficient and intelligent synthesis.

From Section 1, we know that an exhaustive search of the exponential design space cannot be completed in a reasonable or acceptable time period. We thus need to investigate techniques that could increase our design-space exploration efficiency without trading off design quality. PSM used a cost-based heuristic to explore design space, but this produced designs that were always the most expensive ones.

The four techniques used in our methodology to improve synthesis efficiency without trading off design quality are summarized and the reasons we use them are described as follows:

(T1) *Object-Oriented Design*. The elementary application of object-oriented techniques in PSM is extended such that not only the system modeling but the whole design process is also object-oriented, thus making the synthesis methodology more consistent and complete.

(T2) *Fuzzy Design-Space Exploration*. In order to produce more balanced designs (as compared to those produced by PSM), a fuzzy design-space exploration algorithm that considers a global trade-off of cost and performance factors is used, thus not only producing more balanced designs but also performing a more optimal search of the design space.

(T3) *Concurrent Design*. A concurrent component design method is adopted, rather than the sequential one used in PSM, mainly because synthesis efficiency can be improved. A component is not necessarily a physical one, it may represent high-level system parts or subsystems and is often a *design alternative* with respect to other components concurrently under design. Due to the large number of design alternatives, it is certainly desirable to synthesize them concurrently. This is similar to the concurrent executions of two or more branch statements in software, which leads to efficient software execution. For example, if both mesh and cube interconnections satisfy the given specifications, then two design alternatives using different interconnections can be designed concurrently as their designs are independent of each other.

(T4) *Intelligent Reuse of Complete Subsystems*. A substantial amount of design time is saved through intelligent learning and reuse of the previously designed system parts that meet current specifications.

In summary, ICOS basically uses various techniques to enhance the elementary PSM such that (1) the design process is completely object-oriented, (2) more balanced and optimal designs are produced, (3) the synthesis efficiency is improved, and (4) substantial design time is saved through intelligent design reuse.

Referring to Equation 1, as far as the design-space exploration is concerned, technique T1 reduces τ, the average design time of a single component, through efficient synthesis; T2 reduces m, the number of specializations, by considering only a suitable number of instances for each component; T3 also reduces τ by parallelizing the sequential design in PSM; and T4 reduces n, the total number of components to be synthesized, as certain components reused by learning from previous experiences need not be synthesized again.

3. CONCEPTS AND TECHNIQUES

This section presents the concepts and the background of our synthesis methodology, *Intelligent Concurrent Object-Oriented Synthesis* (ICOS), in which system parts are modeled as objects, the synthesis process is object oriented, parts are concurrently synthesized, and previously synthesized parts that meet current specifications are intelligently reused. The previous section discussed *why* these techniques are used in ICOS; the following sections discuss *how* system-level synthesis can make use of these techniques to improve design maintenance, increase synthesis efficiency, and decrease overall design time.

3.1 System-Level Specifications and Synthesis

An ICOS designer describes the desired system through specifying requirements at the system level, which include *architectural*, *performance*, and *synthesis specifications*.

Architectural specifications mainly allow a designer to *restrict* the domain space by specifically indicating how a system part should be constructed. For example, the designer may explicitly specify that the architecture should be a hypercube-connected one with at least 1,024 processing elements and a maximum cost of $12,000. If some architectural details are left out by the designer, for instance, the amount of main and cache memories, then the synthesis system decides how much memory to use and what kind of design alternatives are feasible. For example, one design alternative may be 16Mb of main memory and 1Mb of cache memory, and another design alternative could be 8Mb of main memory and 2Mb of cache memory.

Performance specifications include the minimum system power, which is also the throughput-utilization ratio, the minimum system scalability, reliability, and fault-tolerance, all of which are defined as in PSM [Hsiung et al. 1996]. Synthesis specifications include the maximum number of design alternatives to consider for further design and the choice of whether to reuse previously learned designs or to design a system from scratch.

System-level synthesis is a process that uses the preceding architecture, performance, and synthesis specifications as input and generates a set of feasible design alternatives satisfying all specifications. This could be an empty set if the specifications cannot be satisfied by any design alternative or the specifications themselves are contradictory, for example, the total number of system processors exceed the capacity of the interconnection network chosen (say, a particular shared bus).

3.2 Object-Oriented Design

Object-oriented design includes object-oriented modeling and object-oriented synthesis, which contribute towards easier design maintenance and efficient synthesis, respectively. Hardware components or subsystems can be naturally perceived as objects and classified into some class or classes.

A successful application of object-oriented concepts and techniques in computer system design was demonstrated in PSM. Apart from the normal features of an object-oriented system, such as class encapsulation, attribute inheritance, polymorphism, and part reuse, PSM introduced the use of

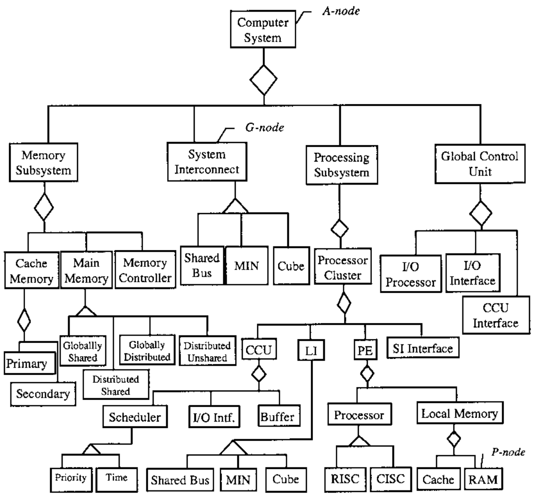


Fig. 1. Class hierarchy.

object-oriented relationships (*aggregation* and *generalization* [Rumbaugh et al. 1991]) and operators (*iterator* [Shaw et al. 1981] and *generator*) for the system-level synthesis of multiprocessor systems. Our current work, ICOS extends the use of object-oriented techniques in system-level synthesis by introducing one more relationship, *dependence*, and one more operator, *updator*.

Each component in a multiprocessor architecture is modeled by a class that may have *specifications* stipulated by the designer, *pre-design characteristics* that are known before design, and *post-design characteristics* that are known only after design. The classes are classified into three types: *A-node* (aggregate node), *G-node* (generalized node), and *P-node* (physical node) depending on whether it represents an *assembly* of subclasses, a *superclass* of some specialized classes, or a *physical* class that is available for direct integration and use, respectively. Three types of relationships are also defined, namely, *aggregation*, *generalization*, and *dependence*, of which the former two are adopted from Rumbaugh's *Object-Modeling Technique* (OMT) [Rumbaugh et al. 1991] and dependence is a newly introduced one. Dependence mainly models how a component may depend on another component due to the hardware-links between them. Two types of dependence are modeled: *absolute dependence* and *relative dependence* [Hsiung 1996].

Using the classes and relationships previously described, a hierarchy of classes called *Class Hierarchy* (CH) is constructed, which can serve as off-the-shelf building blocks for synthesis. *Class Hierarchy* is defined as a multilevel, object-oriented, hierarchically classified repository storing parts of a multiprocessor system. An example of CH is given in Figure 1.

ICOS uses OO operators, namely *iterator*, *generator*, and *updator*, for synthesis. The iterator is used to synthesize an A-node in the design process of ICOS. It iterates through each child node of an A-node, deciding whether to use it; this decision is based on the specification satisfaction of the A-node. The *generator* operator is used to synthesize a G-node and instantiate a P-node. It generates a number of acceptable specialized subclasses for a G-node or instances for a P-node, by traversing CH and checking which subclasses or instances best satisfy the specification of a G-node or a P-node, respectively. Both the iterator and the generator operators are used in the *component synthesis* step (Section 4.3.3). Updator is used to update a *specification* of a node before the node begins synthesis, as explained in the *specification update* step (Section 4.3.1).

3.3 Concurrent Synthesis

Encapsulating each system component as an individual class using OO techniques induces a certain degree of local autonomy such that a class is capable of actively synthesizing itself by traversing down the hierarchy of CH until all leaf nodes are instantiated. This is called *self-synthesis*. When two or more classes representing system parts or design alternatives actively synthesize themselves at the same time, the design process is called *concurrent synthesis*. Concurrent synthesis not only increases synthesis efficiency, but also saves design time as illustrated in this article.

In the following, we describe how more than one component may undergo synthesis at the same time in ICOS. Starting from a root class known as a `Computer System` (CS), ICOS traverses CH and checks components for synthesis. A component class is said to be *ready* for synthesis as soon as all of its specification values are available and updated. The synchronization between component design processes is maintained by the dependence relationships in CH that control the design precedence order of component classes. For example, if a class A is absolutely dependent on a class B, then the synthesis of B must be completed before A can begin its synthesis, and in the case of a relative dependence, A can begin synthesis as soon as its dependent specification is updated by querying B.

For modeling and solving problems induced by concurrency in synthesis, a high-level Petri net model was proposed and validated [Hsiung et al. 1997]. Due to space consideration, the details of this *Multitoken Object-Oriented Bi-direction net* (MOBnet) cannot be included in this article, interested readers are advised to refer to Hsiung et al. [1997] for a complete discussion.

3.4 Intelligent Synthesis

By incorporating learning into the synthesis process, complete subsystems or system parts that meet current design specifications can be reused from previous design experiences, thus eliminating the repetition of similar design steps and saving a substantial amount of design time.

As shown in Figure 2, *machine learning* is basically classified into *Similarity Based Learning* (SBL) and *Explanation Based Learning* (EBL) [Kodratoff 1988]. There are two kinds of SBLs: *Empirical* SBL and *Ratio-*

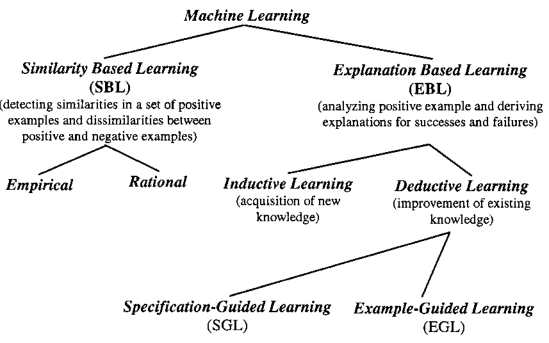

Machine Learning
Similarity Based Learning (SBL)
(detecting similarities in a set of positive examples and dissimilarities between positive and negative examples)
Explanation Based Learning (EBL)
(analyzing positive example and deriving explanations for successes and failures)
Empirical
Rational
Inductive Learning
(acquisition of new knowledge)
Deductive Learning
(improvement of existing knowledge)
Specification-Guided Learning (SGL)
Example-Guided Learning (EGL)

Fig. 2. Machine learning classification.

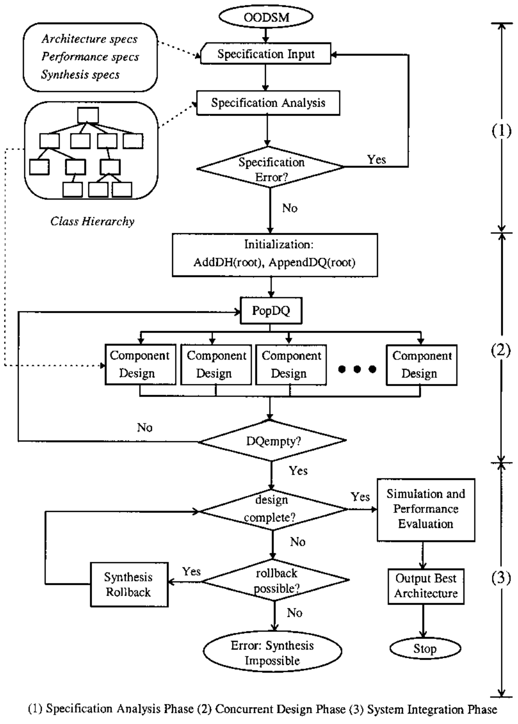


Fig. 3. ICOS design flow.

nal SBL; and EBL includes *Inductive Learning* and *Deductive Learning*. Deductive Learning is further classified into *Specification-Guided Learning* (SGL) and *Example-Guided Learning* (EGL).

ICOS applies SGL in its design process. In SGL, the *specifications* of some previously learned designs are compared with the current user specifications and, if acceptable, a previous design that best meets the current specifications is selected. Since numerous specifications have to be considered, ICOS fuzzifies the comparison between two component classes, this process is called *Fuzzy Specification-Guided Learning* (fuzzy SGL). Details of using fuzzy SGL in ICOS is described in Section 4.3.2.

4. ICOS METHODOLOGY

Having gone through **why** and **how**, techniques can be applied in system-level synthesis, an actual methodology, *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology, implementing the preceding concepts is presented in this section.

As shown in Figure 3, after a designer inputs system requirements using a *specification language* provided by ICOS, the methodology enters its three main phases: *Specification Analysis*, *Concurrent Design*, and *System Integration*. Each of these phases is discussed in the following sections and illustrated with a small running example.

4.1 Specification Language

The ICOS specification language is composed of three specifications: the *architecture*, the *performance*, and the *synthesis* specifications, described as follows:

architecture:		
system:	$AT = \{\mathrm{MP} \mid \mathrm{SM} \mid \mathrm{Hybrid}\}$	//MP=Msg. Passing, SM=Shared Mem.
	$CT = \{\mathrm{SIMD} \mid \text{a-MIMD} \mid \text{s-MIMD}\}$	//s=synchronous, a=async.
	$MT = \{\mathrm{GS} \mid \mathrm{DS} \mid \mathrm{GD} \mid \mathrm{DU} \mid \mathrm{Cache}\}$	//G=Globally, D=Distributed, S=Shared, U=Unshared.
	$SI = \{\mathrm{Bus} \mid \mathrm{MIN} \mid \mathrm{HC} \mid \mathrm{Mesh} \mid \ldots\}$	//MIN=Multistage Interconnection Network, HC=HyperCube.
	SP = Total System Processors	
	NC = Number of clusters	
cluster:	$PU = \{\mathrm{RISC} \mid \mathrm{CISC}\}$	
	$CI = \{\mathrm{Bus} \mid \mathrm{MIN} \mid \mathrm{HC} \mid \mathrm{Mesh} \mid \ldots\}$	
	CP = Total Cluster Processors	
performance:	$MaxC$, $MinP$, $MinS$, $MinR$, $MinF$	
synthesis:	NS, $ML = \{\mathrm{Yes} \mid \mathrm{No}\}$	

In the preceding architecture specifications, AT, CT, MT, SI, PU, and CI are the architecture type, control type, memory type, system interconnection, processing unit, and cluster interconnect, respectively. In the performance specifications, $MaxC$, $MinP$, $MinS$, $MinR$, and $MinF$ are the maximum cost, minimum power, minimum scalability, minimum reliability, and minimum fault-tolerance, respectively. Of particular mention are: NS, the *maximum* number of specializations to be considered at the end of *Fuzzy Design-Space Exploration* of a G-node or P-node, and ML, the option whether any machine learning is to be used. Observe that NS is used by the designer to control the size of the design-space explored at a G-node or

P-node. If this specification is not given by the user, the system default value, $MaxS$ (Equation (7)), will be used.

As shown in Figure 4, a small example is used to illustrate each of the design phases. The designated system is a SIMD message-passing architecture with a `Multi-stage Interconnection Network` (MIN) or a `Hypercube` (HC) interconnection. The design specifications are as follows:

architecture:	
system:	AT = MP, CT = SIMD, SI = MIN $\vee$ HC
cluster:	PU = RISC, CI = Bus
performance:	$MaxC$ = \$10,000, $MinP$ = 8Mflops, $MinR$ = 0.9
synthesis:	ML = Yes

This small example with the preceding specifications will be synthesized in the following sections. Note that not all specifications need to be input by the designer. A check for completeness and compatibility has to be performed first.

4.2 Phase I. Specification Analysis

User-given specifications may contain logical, technical, or typographical errors, which must be detected and eliminated. ICOS begins with analyzing the design specifications, which is mainly done using first-order logic rules and is based on common architecture assumptions. Some of the main assumptions are: a message-passing architecture is not supposed to share any global main memory; a shared-memory architecture should not use the direct-connection networks such as hypercube or mesh; the total number of system processors should be equal to the number of clusters times the number of processors per cluster; and the cost bound should be at least the minimum cost of a uniprocessor system. Figure 5 shows how specification errors such as contradictions between specifications (e.g., AT and MT have incompatible values assigned), unsatisfiable specifications (e.g., $SP \neq NC \times CP$), and incomplete specifications (e.g., CP, SP, and $MaxC$ are all not given) are detected. The analysis is done per specification category, as well as, between the architecture and the performance specification categories. The purpose of this phase is to uncover inconsistencies in the design specifications at the very beginning of the design process so we can avoid futile efforts in synthesizing an impossible system.

For example, continuing with our small example, some rules for analyzing its specifications are described in Figure 5. Since the architecture type desired is message passing (AT = MP), we make necessary assumption that the memory type is distributed unshared (MT = DU). The analysis is performed under assumptions that USC = \$1,000, LPC = \$500,

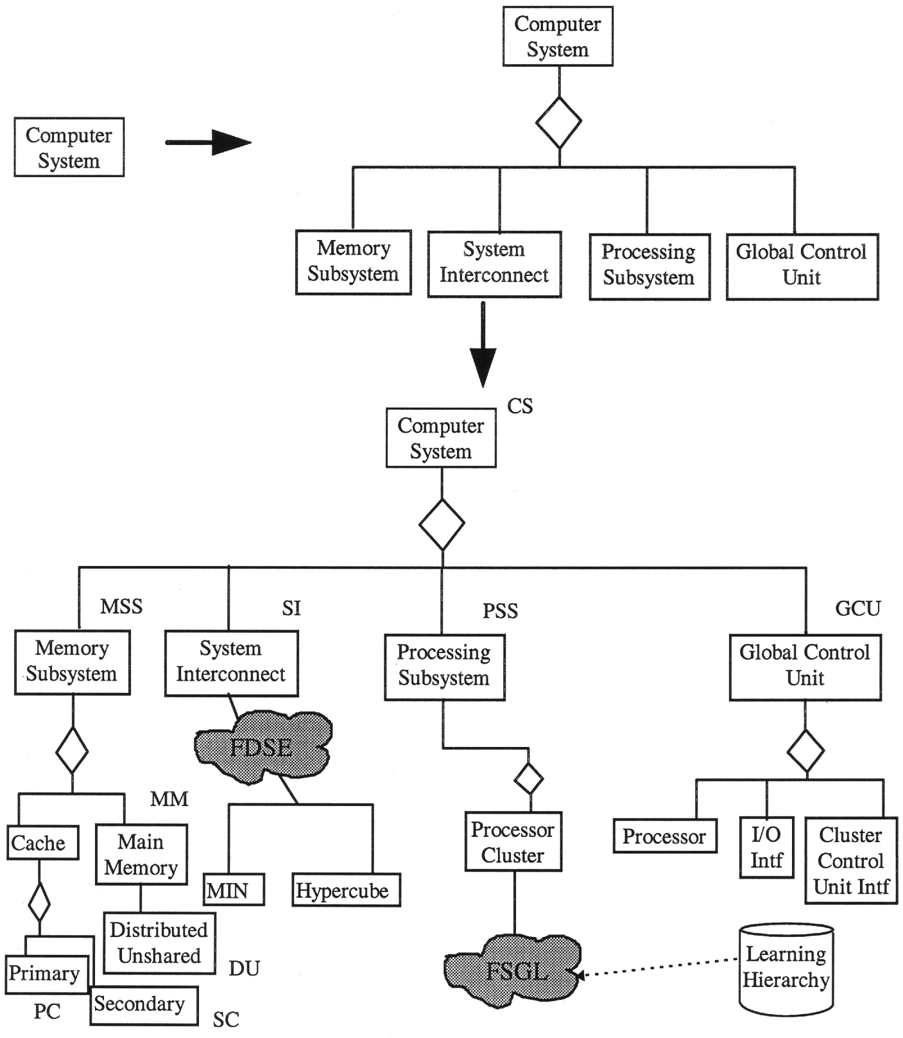


Fig. 4. A small illustrative example.

(a) **architecture specification:**

<u>False Statements:</u> $(AT{=}\text{MPA}) \wedge !(MT{=}\text{DU})$

$(AT{=}\text{SMA}) \wedge (SI{=}\text{HC})$

$SP \neq NC \times CP$

<u>Implications:</u> $(AT{=}\text{MPA}) \Rightarrow (MT{=}\text{DU})$

$SP = NC \times CP$

(b) **performance specification:**

<u>False Statements:</u> $MaxC < USC$

(c) **synthesis specification:**

<u>False Statements:</u> $NS \leq 0$

(d) **architecture/performance specification:**

<u>False Statements:</u> $MaxC < SP \times LPC + LC(SI) + LC(MT)$

<u>Some assumptions:</u>

USC:Uniprocessor System Cost
LPC:Least Processor Cost
LC(SI):Least Cost of System Interconnection
LC(MT):Least Cost of Memory Type

Fig. 5. Specification analysis.

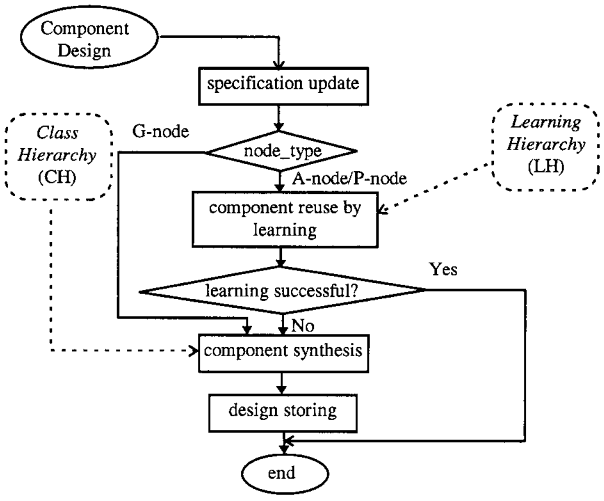


Fig. 6. Component d esign.

LC(*SI*) = \$150, *LC*(*MT*) = \$150, where the meanings of *USC*, *LPC*, *LC*(*SI*), and *LC*(*MT*) have been given in the figure.

4.3 Phase II. Concurrent Design

Concurrent design is the main phase in which components are concurrently synthesized or reused by learning. Here, a *component* modeled as a class, is a system part that may be part of a design alternative. At this stage, specifications are free of errors. A root node representing the `Computer System` (CS) to be synthesized is given to start the *Design Hierarchy* (DH), a subset of CH used to keep track of the system structure under design. A *Design Queue* (DQ), being used to keep track of components ready for synthesis, is also initialized similarly. After initialization, the design of a component begins by removing the root node from the DQ. As shown in Figure 6, there are basically four steps in component design: *specification update*, *component reuse by learning*, *component synthesis*, and *design storing*.

To handle the architectural dependence of a component on another component in concurrent synthesis, we model this dependence in the Class Hierarchy itself using the dependence relationships defined in Hsiung [1996]. During the actual synthesis, a component updates its specifications; if it is dependent on another component, it will have to wait till that component is able to pass over the required information to it. When performance constraints are violated at some stage of synthesis, a rollback process occurs in the bottom-up direction of the hierarchy such that the component-violating performance constraints sends rollback messages to its parent class and dependent classes, both of which in turn either resynthesize themselves or propagate rollback messages upwards in the

Table I. Design Steps: The Small Illustrative Example

Step	Object Ready	Class Type	Operator	Design Method	Design Queue
(a)	CS	*A*	iterator	synthesis	{MSS,SI,Pss,GCU}
(b)	MSS	*A*	iterator	synthesis	{SI,PSS,GCU,Cache,MM}
(c)	SI	*G*	generator	Fuzzy DSE	{Pss,GCU,Cache,MM}
(d)	PSS	*A*	iterator	synthesis	{GCU,Cache,MM,Cluster}
(e)	GCU	*A*	iterator	synthesis	{Cache,MM,Cluster}
(f)	Cache	*A*	iterator	synthesis	{MM,Cluster}
(g)	MM	*G*	generator	synthesis	{Cluster}
(h)	Cluster	*A*	iterator	Fuzzy SGL	{}

MM = Main Memory

Class Hierarchy. The details of this rollback process can be found in Hsiung et al. [1997].

Synthesizing the small example as specified earlier, the design steps are given in Table I. The last column gives the resulting DQ obtained by synthesizing the ready-for-synthesis object in that step (column 2). ICOS methodology stops when DQ becomes empty, which occurs in a finite number of steps as the number of components are finite in a system.

4.3.1 *Specification Update*. A component class may have characteristics that depend on its parent class or dependent classes; hence, it must update all of the related specifications before the synthesis begins. This is done using the *updator* operator. A class queries its parent class, as well as, all the classes having a *dependence relationship* with it, for any missing specification values. After all the queries have been answered, if there are still some specifications that do not have values assigned, the designer of the system will be queried for the specific values. Once all specifications of a class are updated, the class is considered to be *ready* for self-synthesis, which is described in the following steps.

4.3.2 *Component Reuse by Learning*. Before the actual synthesis, a component class checks whether learning from previous design experience is possible. In this step, fuzzy SGL is applied to an A-node.

The rationale of applying Fuzzy Specification-Guided Learning (fuzzy SGL) to an A-node is that *if the design of a partial system, represented by an A-node, can be substituted directly by some previously stored designs, the whole subtree rooted at that A-node need not be synthesized again.*

Consider a component class *cls* in CH, having a set of *k* specifications, SPEC(cls) = $\{s_1, s_2, \ldots, s_k\}$. Suppose that the *n design versions* of *cls*, $V_{cls} = \{cls_1, cls_2, \ldots, cls_n\}$ obtained from previous design experiences are stored in the *Learning Hierarchy* (see Section 4.3.4) and have the following sets of specification values, respectively.

$$X_i = \{x_{ij} \mid x_{ij} \text{ is the value of } s_j \text{ w.r.t. } cls_i, j = 1,2, \ldots, k\}, i = 1,2, \ldots, n \quad (2)$$

Table II. Types of Specifications and Partial Proximity Values

Type of Specification	Example Specifications	Partial Proximity $\widehat{\mu_P}(x_{ij})$
Exact value or set enumeration	AT, CT, MT, SI	-1 if $x_{(n+1)j} \notin ENUM\{x_{ij}\}$ w_j if $x_{(n+1)j} \in ENUM\{x_{ij}\}$
Minimum value (lower bound)	$MinP$, $MinS$, $MinR$, $MinF$	-1 if $x_{ij} < x_{(n+1)j}$ $w_j(x_{ij} - x_{(n+1)j})/M$ if $x_{ij} \geq x_{(n+1)j}$ & $M > 0$. 0 if $M = 0$
Maximum value (upper bound)	$MaxC$, NS	-1 if $x_{ij} > x_{(n+1)j}$ $w_j(x_{(n+1)j} - x_{ij})/M$ if $x_{ij} \leq w_j x_{(n+1)j}$ & $M > 0$. 0 if $M = 0$
Approximate value	buffer size	$w_j(\lvert x_{(n+1)j} - x_{ij}\rvert)^{-1}/M$ if $x_{(n+1)j} \neq x_{ij}$ w_j if $x_{(n+1)j} = x_{ij}$

$M = Max_{1 \leq i \leq n}\lvert x_{(n+1)j} - x_{ij}\rvert$, w_j is the weight associated with s_j and $\Sigma_{j=1}^{k} w_j = 1$

Assume that *cls* is currently to be synthesized again for the $(n + 1)$th time, with the specification values,

$$X_{n+1} = \{x_{(n+1)j} \mid x_{(n+1)j} \text{ is the value of } s_y \text{ w.r.t. } cls_i, j = 1,2, \ldots, k\}.$$

A fuzzy comparison between the values of a current user specification and those of each previous design is made using a fuzzy set (P) that represents the *functional proximity* of previous design versions to the current one under design. The membership function of (P) is defined as follows:

$$\mu_P : V_{\text{cls}} \leftrightarrow \begin{cases} [0,1], & \text{if all specifications are satisfied} \\ (-\infty, 0), & \text{if some } s_i \in \text{SPEC}(cls) \text{ is not satisfied} \end{cases} \tag{3}$$

In Equation (3), when a design version does not satisfy the specifications of the component under design, μ_P is assigned a negative value in $(-\infty,0)$ so that it is not considered as an acceptable design version for reuse. Depending on the type of specification, the *proximity* of cls_i, $\mu_P(cls_i)$ is calculated as a sum over all the specification values,

$$\mu_P(cls_i) = \sum_{j=1}^{k} \widehat{\mu_P}(x_{ij}) \tag{4}$$

where $\widehat{\mu_P}(x_{ij})$, the *partial proximity* of cls_i corresponding to specification s_j, is defined in Table II for each type of specification.

Based on the type of specification, there are different ways to compare how two components differ with respect to a certain specification. A weight (w_j) is assigned to each specification (s_j) representing the importance of the specification in the final design. The weights may be all equal; that is, $w_j = 1 / n$, if all the specifications are equally important. The specifications are classified into four types: (1) exact value or set enumeration (e.g.,

Table III. Fuzzy Specification-Guided Learning at Cluster Class

Design	*CP* / *NC*	**MinP**$_{\text{(MFlops)}}$	**PU**	**CI**	**MaxC**($)	μ_{p}
A	4	2.0	SuperSPARC	Bus	4,200	− 1.400
B	3	1.8	PA-7100	MIN	3,000	− 2.840
C	2	1.0	MIPS-R4400SC	Bus	2,500	− 0.400
D	2	1.1	PowerPC-601	Bus	1,200	0.620
E	2	1.2	Alpha-21064	Bus	1,100	0.647
F	2	1.1	PowerPC-601	MIN	1,500	− 1.580
G	2	1.0	Alpha-21064	Bus	1,000	0.613
Current	2	1+	RISC	Bus	1,200	1.000

μ_{p} is calculated using Equation (4) and Table II.

the CPU must be a RISC CPU); (2) minimum value or lower bound (e.g., the reliability should be at least 98.5%); (3) maximum value or upper bound (e.g., the cost should be at most $ 100,000); and (4) approximate value (e.g., the buffer size should be approximately 1Kb). In Table II, the value of a specification s_j of a design version cls_i in V_{cls} is denoted as x_{ij} and the currently desired specification value is $x_{(n+1)j}$. When a specification s_j is satisfiable by a design version cls_i, the comparison is made between the two values x_{ij} and $x_{(n+1)j}$ by a *weighted normalized difference*, such as $w_j(x_{ij} - x_{(n+1)j})/M$, where M is the maximum difference over all the design versions in V_{cls}. When a specification is not satisfiable, a negative value of -1 is assigned so as to eliminate the consideration of that design version.

The set of design versions considered to be *similar* to the current one under design is called the *similarity set*, $\Delta_{\text{cls}} = \{cls_i | cls_i \in V_{cls}, \mu_P(cls_i) \geq \delta\}$, where δ is a threshold value known as the *degree of similarity*. The higher the value of δ, the smaller is the cardinality of the similarity set, and hence, the greater is the degree of similarity required between the design versions. If the similarity set is not empty, the design version having the maximum $\mu_P(cls_i)$ is selected as the partial-design to be reused for the object in the current synthesis.

For example, step (h) in Table I involves a fuzzy SGL process at the `Cluster` class, suppose the specifications of `Cluster` are: $CP / NC = 2$, $MinP = 1$ MFlop per 100% utilization, $PU =$ RISC , $CI =$ Bus , and $MaxC = \$1{,}200$. Notations are given in Section 4.1. Table III shows how fuzzy SGL is performed at the `Cluster` class. Assuming $\delta = 0.62$, it is observed from Table III, that the similarity set $\Delta_{\text{Cluster}} = \{D, E\}$, and E is the design with maximum μ_P; hence, the design E is reused for the current `Cluster` synthesis.

4.3.3 *Component Synthesis*. Any system part modeled as an individual class in CH is called a "component."*Component synthesis* is the core part of component design. When no reuse by learning is possible or ML is set to

"No" in the specifications, the component is *synthesized* in this step. A P-node can be viewed as a G-node at the leaf of the class hierarchy. Hence, the instantiation process of a P-node is similar to the synthesis process of a G-node because the instances of a P-node can be viewed as the specializations of a G-node.

(a) *Synthesis of an A-node*. Recalling that an A-node has the aggregation type of relationship with its child nodes, an object-oriented operator known as the *iterator* is used to synthesize an A-node. The iterator iterates through each child node deciding whether to use it; this decision is based on the specification satisfaction of the A-node. Child nodes to be used for synthesis are added to DH. If the child node is a P-node, it is instantiated; otherwise, it is appended to DQ for further synthesis. For example, steps (a), (b), (d), (e), and (f) in Table I, all synthesize an A-node using the iterator.

(b) *Synthesis of a G-node and Instantiation of a P-node. A Fuzzy Design-Space Exploration*. The fuzzy DSE method is used to select a suitable number of acceptable design components that are among the best specializations of a G-node (G-specialization) or instances of a P-node (P-instance). The object-oriented operator used in fuzzy DSE is known as the *generator*, since it "generates" a suitable number of acceptable specializations or instances.

As shown in Equation (1), we know that the synthesis of computer systems often requires the exploration of a very large design-space containing several G-specializations or P-instances. Although the specializations or the instances of a component class have common functionality, the order of preference among them might be quite difficult to determine. Often the comparison between two specializations or two instances is not crisp or clear as one has to compare several different specifications that have trade-off relationships when certain goals or constraints are considered. For example, a higher fault-tolerance would require a higher total system cost.

Modeling how a component affects each performance factor of the whole system by a fuzzy membership function (Equation (5)) and composing these functions by a linear combination into a composite fuzzy membership function (Equation (6)), we can actually compare two components and determine the order of preference when a selection is required.

Each G-specialization or P-instance is assigned a *penalty factor* f that determines its membership grade in a *fuzzy decision set* D. Let S_{cls} be a set of acceptable specializations or instances for some class cls, $\{C_1, C_2, \ldots, C_n\}$ be a set of constraints, and $\{G_1, G_2, \ldots, G_m\}$ be a set of goals, we define the following membership functions as mappings from S_{cls} to a real number between 0 and 1.

$$\begin{aligned}
&\mu_{C_i} : S_{cls} \mapsto [0,1], i = 1,2, \ldots, n \\
&\mu_{G_j} : S_{cls} \mapsto [0,1], j = 1,2, \ldots, m \qquad (5) \\
&\mu_D : S_{cls} \mapsto [0,1], \text{ where } D = C_i \oplus_{i,j} G_j
\end{aligned}$$

where the penalty factor, which is the fuzzy membership function of the decision set D, is defined as the linear combination $(\oplus_{i,j})$ of all μ_{C_i} and μ_{G_j}.

$$f(s) = \mu_D(s) = \sum_{i=1}^{n} u_i \mu_{C_i}(s) + \sum_{j=1}^{m} v_j \mu_{G_j}(s), \sum_{i=1}^{n} u_i + \sum_{j=1}^{m} v_j = 1, \forall s \in S_{cls} \qquad (6)$$

where u_i and v_j are the weights associated with C_i and G_j, respectively. An implementation example of Equation (6) is given later in Equation (8).

Using Equation (6), we can assign a partial order of preference to any set of specializations or instances by assigning each specialization or instance with a penalty factor f and ordering them ascendingly by f. The specialization or instance with the least penalty $\mathrm{Min}_{s \in S_{cls}}\{f(s)\}$ is locally the best choice. To obtain more than one final design alternative, a larger design-space consisting of more than one specialization or instance is explored. The greater the number of specializations or instances considered, the larger will be the design-space, and thus the less efficient will be the synthesis process. This trade-off between synthesis quality and synthesis efficiency has been experimentally explored and the result of this experimentation indicates the following number of specializations ($MaxS$) to be an appropriate choice.

$$MaxS = \left| \left\{ s \;\middle|\; s \in S_g, \mu_D(s) \leq \frac{\sum_{s \in S_g} \mu_D(s)}{|S_g|} \right\} \right| \qquad (7)$$

where $S(g)$ is the set of acceptable G-specializations for g in DH. Similarly, Equation (7) also holds for the case of P-instances.

In fact, Equation (7) indicates that we should only consider the specializations that have their penalty factors not greater than the average penalty factor.

For example, step (c) in Table I involves fuzzy DSE at `System Interconnect` (SI). Let s_i be a specialization of SI; implementing Equation (6), we define the *partial* fuzzy penalty factors corresponding to the constraint of Cost (C_1) and the goals of Power (G_1), Reliability (G_2), Fault Tolerance (G_3), and Scalability (G_4) as follows:

$$\mu_{C_1}(s_i) = \frac{\mathrm{C}\ (s_i)}{MaxC}, \mu_{G_1}(s_i) = \frac{MinP}{\mathrm{P}\ (s_i)}, \mu_{G_2}(s_i) = \frac{MinR}{\mathrm{R}\ (s_i)}, \mu_{G_3}(s_i) = \frac{MinF}{\mathrm{F}\ (s_i)}, \mu_{G_4}(s_i) = \frac{MinS}{\mathrm{S}\ (s_i)}. \qquad (8)$$

where $MaxC$, $MinP$, $MinR$, $MinF$, and $MinS$ are the respective constraints and C, P, R, F, and S give the cost, power, reliability, fault-

Table IV. Cost and Performance Assumptions for Small Illustrative Example

Characteristics capacity	Bus 8	MIN_1 8x8	MIN_2 8x8	MIN_3 8x8	3-cube 8
Cost ($)	50	100	110	110	120
Power (bytes/s)	100	400	600	700	800
Reliability	0.9	0.9	0.9	0.9	0.9
Fault-Tolerance	0	0	0	0	0
Scalability	0	0.5	0.5	0.5	0.4

Table V. Penalty Factors for Fuzzy DSE at SI Class

Penalty Factors	Bus (s_1)	$MIN_1(s_2)$	$MIN_1(s_3)$	$MIN_1(s_4)$	3-Cube(s_5)
$\mu_{C_1}(s_i)$	0.005	0.01	0.011	0.011	0.012
$\mu_{G_1}(s_i)$	1	1	1	1	0.5
$\mu_{G_2}(s_i)$	0	1	0.667	0.571	1
$\mu_D(s_i)$	0.335	0.67	0.592	0.527	0.504

tolerance, and scalability of the specializations. These terms are defined in PSM [Hsiung et al. 1996]. Some cost and performance assumptions are given in Table IV.

Assuming $MaxC(SI)$ = \$120, $MinP(SI)$ = 400 bytes/s, and $MinR(SI)$SI) = 0.9, Table V shows the penalty factors calculated for each SI specialization using Equations (6) and (8).

Since s_1 does not satisfy the power requirement, $S_{SI} = \{s_2, s_3, s_4, s_5\}$, using Equation (6) we get, $\Sigma_{i=2}^{5}\mu_D(s_i)/4 = 2.293/4 = 0.57325$. Thus, $MaxS = |\{s_4, s_5\}| = 2$. Therefore, only two acceptable specializations $\{s_4, s_5\}$ are considered for further synthesis.

4.3.4 *Step 4. Design Storing and Retrieval.* ICOS uses a *Learning Hierarchy* (LH) for design storing. LH is a structure similar to CH, but has the capability of storing multiple design versions of the same component class. If a component has been synthesized in a component synthesis step instead of having been reused by learning from past experiences, then all its design information including the component name, the specification values, and the design details are stored in LH for future reference and possible reuse. For example, Cache synthesized in step (f) of Table I will be stored in LH for future reuse.

4.4 Phase III. System Integration Phase

In this phase, the full system under design is integrated, simulated, and its performance evaluated. Since ICOS uses a concurrent synthesis approach, a final checking for design completion is necessary; this is accomplished using the recently proposed *Multi-token Object-oriented Bi-directional net* (MOBnet) model [Hsiung et al. 1997]. If the design cannot be completed, synthesis rollback occurs with the aid of the MOBnet model to find other possible design alternatives. Due to space consideration, design completion

```
Class generic{
  protected:
     specifications:                      // specifications to be updated
         spec1 = value1;                  // before synthesis starts
         spec2 = value2; ...
     pre-design characteristics:          // characteristics with values
         prechar1 = value1;               // known before design
         prechar2 = value2; ...
     post-design characteristics:         // characteristics with values
         postchar1 = value1;              // known only after design
         postchar2 = value2; ...
     type = {A-node | G-node | P-node};   // type of node
     synthesized = {TRUE | FALSE};        // if it was ever synthesized before
  public:
     generic();                           // constructor function
     update_spec();                       // update specifications
     reuse_by_learning();                 // reuse by learning
     synthesize();                        // synthesize the generic component
     store_design();                      // store synthesized design
     rollback();                          // rollback the synthesis process
     update_postchars();                  // update post-design characteristics
}
```

check and synthesis rollback are not described here. Interested readers are requested to refer to Hsiung et al. [1997].

Simulation and performance evaluation of the design alternatives are basically the same as those in PSM [Hsiung et al. 1996]. As in PSM, executable component models were created using the SES/Workbench* simulation tool [Scientific and Engineering Software 1992]. This final evaluation of the design alternatives has been extensively covered in PSM; hence, it is not elaborated upon in this article. A design with the best performance is the final architecture output.

5. IMPLEMENTATION AND DESIGN EXAMPLES

As shown in Figure 7, the implementation of ICOS consists of four parts: a CH Constructor, a Synthesizer, a System Simulator, and an LH Maintainer. We implemented this methodology on a Sun SPARC Station-20 machine. The two hierarchies, CH and LH, were implemented as object-oriented databases. Ease of object access and quick relationship traversal were chief concerns during the implementation of the hierarchies. A generic component class is specified above.

Some of the functions are shown in Figure 8. The System Simulator constitutes executable SES/Workbench models. The performance of the

*SES/Workbench is a registered trademark of Scientific and Engineering Software, Inc.

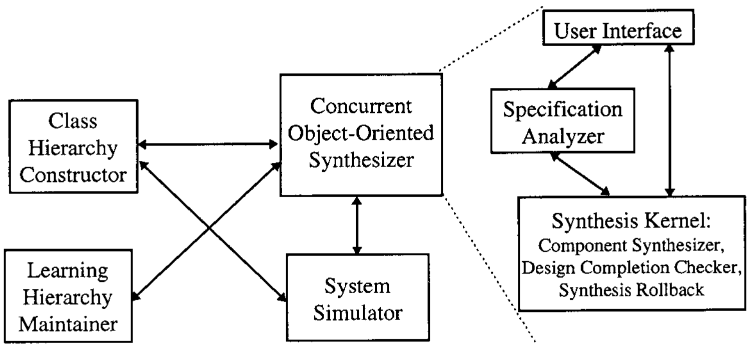
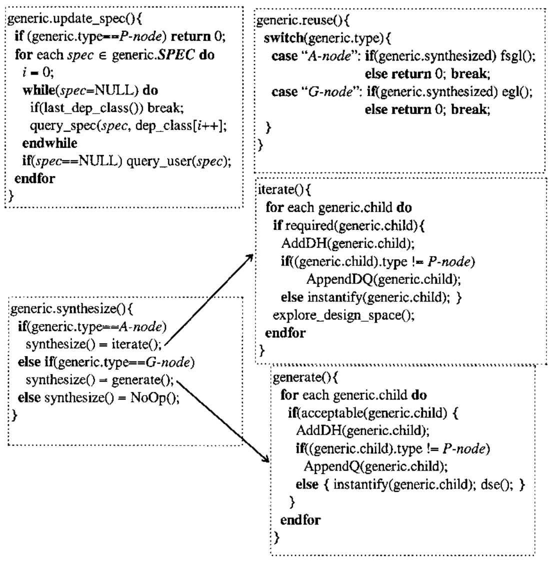

User Interface
Specification Analyzer
Class Hierarchy Constructor
Concurrent Object-Oriented Synthesizer
Synthesis Kernel: Component Synthesizer, Design Completion Checker, Synthesis Rollback
Learning Hierarchy Maintainer
System Simulator

Fig. 7. ICOS implementation.



```
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;
  }
}
```

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

```
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
}
```

```
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
}
```

Fig. 8. Some generic class functions in ICOS.

design alternatives were evaluated using the PSM *Performance Estimation Formula*, $D = (P \times S \times R \times F) / C$, where D is the distance metric, P the power, S the scalability, R the reliability, F the fault-tolerance, and C the total cost [Hsiung et al. 1996].

The *Synthesizer* is the main synthesis part of ICOS. It consists of a *User Interface*, a *Specification Analyzer*, and a *Synthesis Kernel*. The User Interface provides a means for the input of user specifications and the performance constraints and the output of the final architecture. The Specification Analyzer tries to detect all contradictions among user specifications and infeasible or false statements. The Synthesis Kernel is responsible for DH and DQ maintenance, the creation of *Component Synthesis Processes* (CSP), the concurrent process management, and the system

Table VI. Synthesizing the Small Illustrative Example with and without Learning

	#A	#G	#P	Total Nodes	Design Space Size	Synthesis Time (s)
With Learning	6	2	8	16	384	558
Without Learning	9	5	16	30	1052	1200

#A, #G, #P are the number of A-nodes, G-nodes, and P-nodes, respectively.

integration, which includes design completion checking and synthesis roll-back. The synthesis kernel was implemented using object-oriented language C++ and the concurrency of component synthesis processes were realized using processes in a multitasking environment such as the UNIX Operating System. The process of activation of a component class after removal from DQ is implemented in the synthesis kernel as the creation of a CSP that is an individual process for the synthesis of a component. Passing of synthesis parameters such as dependent specifications, implementation of tokens, and the traversal of relationships are all implemented as *Interprocess Communications*. A CSP is killed as soon as the self-synthesis of that component is complete.

The first illustrative example using the ICOS methodology has just been depicted along with the presentation of ICOS in Section 4 and is concluded Section 5.1. Another synthesis example is given in Section 5.2. A list of other application examples is given in Section 5.3. Some observations are presented in the final section.

5.1 Example 1. Synthesis of the Small Illustrative Example

The small illustrative example has been successfully synthesized through the three phases of ICOS as shown in Section 4. Table VI shows how the use of machine learning in ICOS reduces the total number of nodes to be synthesized and how the use of concurrent design techniques reduces the total synthesis time to half of what they would be if machine learning and concurrent design were not used.

5.2 Example 2. Synthesis of an MIMD Architecture

This example shows how the total effect of machine learning at different nodes can increase synthesis efficiency and performance. The designated system is an asynchronous MIMD hybrid (shared-memory and message-passing) architecture with globally shared memory and `Shared Bus` as the System Interconnection. All abbreviated symbols in this example were explained in the specification language (Section 4.1) and the specification analysis phase (Figure 5).

Design Specification:
Architecture:

System: AT = Hybrid, CT = a-MIMD, MT = GS, SI = Bus, $SP = 1024, NC = 64$

Cluster: PU = RISC, CI = MIN, $CP = 16$

Performance: $MaxC$ = \$700,000, $MinP$= 500Mflops, $MinR$=0.9, $MinF$ = 0.5, $MinS$=0.5.

Synthesis: ML =Yes

Design Synthesis:

Specification Analysis:

Analysis: $SP = NC \times CP$, $C > USC$, $MaxC > SP \times LPC + LC(SI) + LC(MT) = 512{,}300$.

Assumptions: USC = \$1,000, LPC = \$500, LC(SI) = \$150, LC(MT) = \$150, C(RAM) = \$140/4MB, C(Cache) = \$100/1MB.

Analysis Result: No error

Design Reuse by Learning: Only partial synthesis is shown in order to emphasize the reuse by learning capabilities of ICOS.

Fuzzy SGL at the Processing Subsystem(PSS):

PSS Specification: PU = RISC, LI = MIN, $CP \geq 16$, C(PSS) ≤ \$10,000, P(PSS) ≥ 8Mflops, LM RAM Size ≥ 1MB, LM Cache Size ≥ 0.5MB, LM RAM Access Time ≤ 8ns, CCU Buffer Size ≈ 1MB.

Assumptions: *Cost*(8 ns RAM) = \$30/MB, *Cost*(7 ns RAM) = \$35/MB, *Cost* (6 ns RAM) = \$38/MB.

As shown in Table VII, using Equation (4) and Table II, the proximity values of the six previously stored designs are calculated as 0.3889, −0.3889, 0.6667, 0.6556, −0.5556, and −1.3333, respectively, where the associated weights are all equal ($w_j = w_i$, $\forall i \neq j$). The similarity set, Δ_{PSS}, is $\{A, C, D\}$ with $\delta = 0.38$. Hence, the best choice is design C. If the LM RAM size, RAM access time, and LM Cache size are given greater importance than the cost of PSS (i.e., $w_j = 1/9$ for $j = 1,2,3,5,9$, $w_4 = 1/18$, and $w_i = 7/54$ for $i = 6,7,8$), then the proximity values are recalculated for A, C, and D as 0.3889, 0.6389, and 0.6704, respectively. In this case, D becomes the best design choice.

Similarly, fuzzy SGL is performed at the MSS class. The savings of design time and cost are as shown in Table VIII.

5.3 Other Examples

The sample designs synthesized by PSM in Hsiung et al. [1996] were resynthesized using ICOS. Table IX compares the performance of PSM and ICOS in synthesizing similar designs.

Table VII. Fuzzy Specification: Guided Learning at PSS Class

#	PU	CI	CP	PSS Cost ($)	PSS Power (MFlops)	LM RAM Size (MB)	LM Cache Size (MB)	LM MAT (ns)	CCU Buffer Size (MB)	μ_P
A	SPARC	MIN	16	10,000	9	1.0	0.5	8	2	0.3889
B	MIPS-R4400SC	MIN	18	13,500	10	1.0	0.5	7	2	− 0.3889
C	ALPHA-21064	MIN	16	9,500	9	1.0	0.6	7	2	0.6667
D	Power PC-601	MIN	16	9,800	9	1.2	0.5	6	2	0.6556
E	Intel Pentium	MIN	18	10,000	8	1.0	0.5	6	2	− 0.5556
F	PA-7100	Mesh	18	13,000	10	1.2	0.5	6	2	− 1.3330
*	{RISC}	MIN	16+	10,000-	8+	1+	0.5+	8-	1 ±	+ 1.0000

* = Current Design, LM = Local Memory, MAT = Memory Access Time

Table VIII. Synthesizing Example 2 With and Without Learning

	#A	#G	#P	Total Nodes	Design Space Size	Synthesis Time (s)
With learning	4	1	4	9	128	392
Without learning	9	4	15	28	512	1150

#A, #G, #P are the number of A-nodes, G-nodes, and P-nodes

From Table IX, we observe that intelligent reuse by learning in ICOS has helped to considerably reduce the total number of nodes synthesized, thus reducing the overall design time by an appreciable amount. The number of nodes synthesized by PSM was two to three times larger than that required by ICOS. Due to concurrent design and intelligent reuse, the time required by ICOS to synthesize a complete multiprocessor system is approximately one-half to one-third of that required by PSM. This shows the efficiency of ICOS over PSM in designing MP systems, when intelligent reuse by learning and concurrent synthesis is used along with object-oriented design.

5.4 Observations

Some observations are made from the examples given in this section.

Learning consistency: The similarity set $\Delta_{\mathtt{cls}}$ does not depend on the weights (w_j) associated with each specification of *cls*. This shows that irrespective of the degree of importance given to the different specifications, the acceptable previous designs to be considered for reuse by learning always remain the same.

Table IX. Comparison between PSM and ICOS

Design	AT	CT	MT	SI	SP	NC	MaxC(10^4 \$)	MinP
A	Hybrid	SIMD	GD	HC	10,240	2,560	1,150	10.5
B	Hybrid	SIMD	GD	MIN	1,024	256	110	5.4
C	SM	MIMD	GS	Bus	1,024	256	175	128
D	MP	MIMD	DU	HC	512	218	60	2

AT, CT, . . . are symbols from the specification language of ICOS

Design	C_{PSM}	C_{ICOS}	S_{PSM}	S_{ICOS}	T_{PSM}	T_{ICOS}
A	32	15	480	120	605	300
B	26	11	440	102	519	242
C	29	11	400	100	580	250
D	20	8	388	64	472	168

C_{PSM}, C_{ICOS} are the no. of components synthesized by PSM and ICOS, respectively; S_{PSM}, S_{ICOS} are the design space sizes explored by PSM and ICOS, respectively; T_{PSM}, T_{ICOS} are the design time in seconds for PSM and ICOS, respectively.

Specification trade-off: By varying the weights associated with each specification, the fitness of a final previous design to be reused for the current application may vary. This shows the flexibility of ICOS learning which allows the designer to trade-off among various specifications.

Fuzzy ordering: Given numerous specifications of a design to be synthesized, it becomes very difficult to associate an ordering among the designs in the Learning Hierarchy. This ordering is necessary for selecting *the most similar designs* to be reused. Learning in ICOS accomplishes this by using a fuzzy proximity set.

Saving in design time: The number of nodes of each type (A, G, and P) to be synthesized with and without learning varies greatly. As shown in the Small Illustrative Example and Example 2, learning during synthesis reduces the total number of nodes to be synthesized to approximately one-half (Table VI) or even one-third (Table VIII) of that which would be required if no learning was used, respectively. Considerable time and effort are thus saved.

6. CONCLUSION AND FUTURE WORK

The design methodology, *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) was presented and implemented. OO-based design representation and fuzzy searching were used in ICOS to successfully synthesize multiprocessor systems by considering all the features of MP systems. Several representative design examples were synthesized using ICOS and compared with those synthesized by PSM [Hsiung et al. 1996]. The experimental results were in adherence to our initial motives.

We have shown how a complete design methodology integrated the techniques of OO, fuzzy logic, concurrent design, and machine learning in modeling and design, design-space exploration, synthesis process, and intelligent reuse, respectively. Each of the four techniques contributes towards synthesis efficiency. Consider the total design time $T(n) = \tau(m)^n$ as given in Equation (1). Object-oriented and concurrent design reduces the average design time τ of a single component by a factor of approximately 3. Fuzzy DSE, without trading off the design quality, reduces the number of specializations (m) needed to be considered for further synthesis to $MaxS$, which is only half of the total number of specializations. Intelligent learning drastically reduces the total number of nodes (n) synthesized to approximately $n/2$ or even $n/3$ since reusing an A-node means the whole subtree rooted at the A-node need not be synthesized again. Each of the four techniques helps to reduce some part of the total design time. The upper bound of the total design time is now $T_{\text{ICOS}}(n) \leq \tau_{\text{OO-CS}}(MaxS)^{n_L}$, where $\tau_{\text{OO-CS}}$ is the average design time of a single component when the system is designed using OO and concurrent synthesis, $MaxS$ is the number of specializations considered for fuzzy DSE (Equation (7)), and n_L is the total number of nodes needed to be synthesized when learning is used. $T_{\text{ICOS}}(n)$ is significantly smaller then $T(n)$ even for a small system, with a small n.

The excellent blend or integration of object-oriented techniques, concurrent synthesis, fuzzy logic, and machine learning has resulted in an efficient and intelligent synthesis approach to multiprocessor system design. Future research directions in this field of multiprocessor system design automation will involve the exploration of the possibility of a hardware-software cosynthesis approach and the formulation of a formal theoretical base for system-level synthesis.

References

De Antonellis, V. and Pernici, B. 1995. Reusing specifications through refinement levels. *Data Knowl. Eng. 15*, 2 (Apr.), 109–133.

Birmingham, W. P., Gupta, A. P., and Siewiorek, D. P. 1989. The MICON system for computer design. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation* (DAC '89, Las Vegas, NV, June 25–29, 1989). ACM Press, New York, NY, 135–140.

Chiang, M.-C. and Sohi, G. S. 1992. Evaluating design choices for shared bus multiprocessors in a throughput-oriented environment. *IEEE Trans. Comput. 41*, 3 (Mar.), 297–317.

Chung, M. J. and Kim, S. 1990. An object-oriented VHDL design environment. In *Proceedings of the ACM/IEEE Conference on Design Automation* (DAC '90, Orlando, FL, June 24-28). ACM Press, New York, NY, 431–436.

Dutta, R., Roy, J., and Vemuri, R. 1992. Distributed design-space exploration for high-level synthesis systems. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation* (DAC '92, Anaheim, CA, June 8-12). IEEE Computer Society Press, Los Alamitos, CA, 644–650.

Gadient, A. J. and Thomas, D. E. 1993. A dynamic approach to controlling high-level synthesis CAD tools. *IEEE Trans. Very Large Scale Integr. Syst. 1*, 3 (Sept.), 328–341.

Gupta, A. P., Birmingham, W. P., and Siewiorek, D. P. 1993. Automating the design of computer systems. *IEEE Trans. Comput.-Aided Des. Integr. Circuits 12*, 4 (Apr.), 473–487.

HSIUNG, P.-A. 1996. System level synthesis for parallel computers. Ph.D. Dissertation. Graduate Institute of Electrical Engineering, National Taiwan University, Taipei, Taiwan.

HSIUNG, P.-A., CHEN S.-J., , HU, T.-C., AND WANG, S.-C. 1996. PSM: An object-oriented synthesis approach to multiprocessor system design. *IEEE Trans. Very Large Scale Integr. Syst. 4*, 1 (Mar.), 83–97.

HSIUNG, P.-A., LEE, T.-Y., AND CHEN, S.-J. 1997. MOBnet: An extended Petri net model for the concurrent object-oriented system-level synthesis of multiprocessor systems. *IEICE Trans. Inf. Syst. E80-D*, 2 (Feb.), 232–242.

KANG, E. Q., LIN, R.-B., AND SHRAGOWITZ, E. 1994. Fuzzy logic approach to VLSI placement. *IEEE Trans. Very Large Scale Integr. Syst. 2*, 4 (Dec.), 489–501.

KODRATOFF, Y. 1988. *Introduction to Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

KUMAR, S., AYLOR, J. H., JOHNSON, B. W., AND WULF, WM. A. 1994. Object-oriented techniques in hardware design. *Computer 27*, 6 (June), 64–70.

LEE, Y. K. AND PARK, S. J. 1993. OPNets: an object-oriented high-level Petri net model for real-time system modeling. *J. Syst. Softw. 20*, 1 (Jan.), 69–86.

LIN, R.-B. AND SHRAGOWITZ, E. 1992. Fuzzy logic approach to placement problem. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation* (DAC '92, Anaheim, CA, June 8-12). IEEE Computer Society Press, Los Alamitos, CA, 153–158.

MABBS, S. A. AND FORWARD, K. E. 1994. Performance analysis of MR-1, a clustered shared-memory multiprocessor. *J. Parallel Distrib. Comput. 20*, 2 (Feb.), 158–175.

MITCHELL, T. M., MAHADEVAN, S., AND STEINBERG, L. I. 1985. LEAP: A learning apprentice for VLSI design. In *Proceedings of the 9th Conference on IJCAI* (IJCAI), 573–580.

REZAZ, M. AND GAU, J. 1990. Fuzzy set based initial placement for ic layouts. In *Proceedings of the European Conference on Design Automation*, 655–659.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ.

SCIENTIFIC AND ENGINEERING SOFTWARE, INC, 1992. *SES/Workbench User's Manual Release 2.1*.

SHAW, M., WULF, W., AND LONDON, R., Eds. 1981. *Abstraction and Verification in Alphard: Iteration and Generators*. Springer-Verlag, New York, NY.

Received: February 1997; revised: April 1997; accepted: August 1997