POSE: A Parallel Object-Oriented Synthesis Environment

PAO-ANN HSIUNG
National Chung Cheng University

Design automation tools and methodologies always encounter a problem of how systems may be designed efficiently, including issues such as static modeling and dynamic manipulation of system parts. With the rapid progress of design technology, the continuously increasing number of different choices per system part and the growing complexity of today's systems, the efficiency of the design environment is not only a major concern now, but will also be a demanding problem in the near future. In contrast to heuristic methods, a novel environment called POSE is proposed that increases efficiency during design without losing optimality in the final design results. System parts are modeled using the popular object-oriented modeling technique and are dynamically manipulated using the parallel design technique. A complete integration of object-oriented and parallel techniques is one of the major features of POSE. Common problems related to parallel design such as *emptiness* and *deadlock* are also elegantly solved within POSE. Experimental results and formal analysis based on POSE all show its practical and theoretical usefulness. POSE can be used at any level of synthesis as long as off-the-shelf building-blocks manipulation is required. POSE can be applied especially to *system-level* synthesis, whose targets can be parallel computer architectures, systems-on-chip, or embedded systems. We will show how POSE has been applied to ICOS, a recently proposed synthesis methodology. Furthermore, POSE can be easily integrated with other heuristic design methodologies to allow increased design efficiency.

Categories and Subject Descriptors: J.6 [**Computer Applications**]: Computer-Aided Engineering—*Computer-aided design* (CAD); B.m [**Hardware**]: Miscellaneous—*Design management*

General Terms: Design

Additional Key Words and Phrases: Design-completion check, hardware synthesis, object-oriented technology, parallel design, synthesis rollback

1. INTRODUCTION

Current technology advances have produced an enormous amount of hardware system components including different types of ASICs, I/O devices,

Author's address: Department of Computer Science and Information Engineering, National Chung Cheng University, 160, San-Hsing, Min-Hsiung, Chiayi-621, Taiwan, ROC; email: hpa@computer.org.

communication and control interfaces, memory modules, processing elements, etc. This has led to an increased design effort when a system is to be synthesized (i.e., designed automatically) from such a large choice of system parts. It is a well-known fact that the size of the system design-space is generally exponential in the number of system parts or components used for design [Hsiung et al. 1998]. Many heuristic methods have been developed to explore the design space, partially in order to arrive at a heuristically optimal solution in an acceptable period of design time. Machine learning and fuzzy logic are some of the techniques proposed to reduce design time, but they all aim at deriving a heuristic solution. In the current work, we aim at attacking the problem by directly reducing the overall design time at no expense of losing optimality in the design alternatives produced. Our approach is two-fold: one is *parallelizing* the design procedures and the other is modeling both the static system parts and the dynamic manipulations of parts during synthesis using *object-oriented* technologies.

System-level synthesis is the process of automatic transformation from a set of system specifications including architectural, performance, and design-related requirements into a high-level architecture consisting of a description of the various components such as processors, memory, interconnections, and the number of each component used. Different target systems require different system-level design primitives and constraints. For example, on the one hand, SOC (Systems-On-Chip) designs are often designed using several IPs (Intellectual Property) and their specifications include power consumption, chip-area usage, etc. On the other hand, parallel architecture designs are often designed using PMU (Processor-Memory Units) and system interconnections, and their specifications include minimum throughput, maximum cost, etc. Some examples of system-level synthesis tools include the MICON system [Birmingham et al. 1989; Gupta et al. 1993] and the Megallan system [Gadient and Thomas 1993]. Recently proposed methodologies for system-level synthesis include PSM [Hsiung et al. 1996] and ICOS [Hsiung et al. 1998].

Traditionally, systems were synthesized in a sequential manner, that is, only one design was synthesized at a time. This often resulted in a very large overall design time and delayed detection of infeasible specifications. However, to remedy such a situation, if system parts could be synthesized simultaneously and design alternatives produced in parallel, then the overall design time could be reduced significantly, thus allowing early detection of nonsynthesizable specifications. When system parts are synthesized in parallel, the dynamic concurrent manipulation of design parts and partially synthesized design alternatives may cause some temporal and spatial overhead. This is because a record of the current status of all design parts under synthesis has to be maintained, design consistencies among the design parts have to be ensured, and one or more design parts may have to be resynthesized (also called *synthesis rollback* as described later) when some unsynthesizable specifications are encountered. In such a parallel design environment, the management overhead could be reduced

significantly when the synthesis control is *distributed*, rather than centralized. Under distributed synthesis control, parts under design will have to be able to send and receive messages. Further overhead reduction could be achieved when the entire parallel design environment is *object-oriented*, thus allowing part encapsulation as required for distributed control. Object-oriented (OO) technology is used not only for the static modeling of system parts, but also in the dynamic parts manipulations during synthesis, thus allowing a compact integration of OO with parallel design. Some common problems encountered in a parallel design environment such as *emptiness* and *deadlock* can also be elegantly solved using OO techniques.

Parallel Object-Oriented Synthesis Environment (POSE) is a general design environment that integrates the above-mentioned parallel and object-oriented design techniques for high-level design space exploration. Though POSE can be applied to any system-level synthesis methodology and to any target machines such as SOC, parallel architecture, or embedded systems, we will mainly illustrate the concepts in POSE through a recently proposed parallel architecture synthesis methodology called *Intelligent Concurrent Object-Oriented Synthesis* (ICOS) methodology [Hsiung et al. 1998].

The paper is organized as follows. Section 2 describes some previous and related work on synthesis, heuristic design, and OO techniques. Section 3 describes the system model and the synthesis model in POSE. Section 4 shows how some parallel design related problems such as *emptiness* and *deadlock* are solved in POSE. Section 5 describes the application of POSE. Section 6 concludes with some future work.

2. PREVIOUS RELATED WORK

As far as hardware system synthesis is concerned, there have been several methodologies and tools developed to either fully or partially automate the design process at various levels of design such as the MICON system [Gupta et al. 1993; Birmingham et al. 1989], the *Megallan* system [Gadient and Thomas 1993] including the *System Architect's Workbench* [Thomas et al. 1988], WOLFIE, and LASSIE [Trick and Director 1989], *Performance Synthesis Methodology* (PSM) [Hsiung et al. 1996] and *Intelligent Concurrent Object-oriented Synthesis* (ICOS) methodology [Hsiung et al. 1998], to mention a few.

The problem of synthesis efficiency has been tackled by introducing heuristics into the design methodology. Rather than searching the entire design-space exhaustively, several techniques have been proposed in the past to partially explore the region that most likely contains the optimal design solution. The techniques include fuzzy logic [Rezaz and Gau 1990; Lin and Shragowitz 1992; Kang et al. 1994], learning [Mitchell et al. 1985], object-oriented design [Kumar et al. 1994], object-oriented language [Chung and Kim 1990], specification reuse [Antonellis and Pernice 1995], distributed exploration [Dutta et al. 1992], and formal approaches [Hsiung et al. 1997a; Lee and Park 1993]. Although the proposed techniques help increase synthesis efficiency, they do not guarantee optimal solutions.

With the increasing wide-spread use of object-oriented technology in software modeling and development, there has been a technology transfer from software to hardware [Brooks et al. 1984; Gross 1985; Hsiung et al. 1997b]. Application of OO to hardware synthesis has matured from a simple modeling [Kumar et al. 1994] as in PSM [Hsiung et al. 1996] to a complex design methodology such as ICOS [Hsiung et al. 1998]. OO has also been applied in the formal analysis of systems [Lee and Park 1993] and synthesis [Hsiung et al. 1997b]. Though OO has been applied, but the actual method of application is not very clear from the previous work. Formerly, distributed design techniques have been used to reduce the design space exploration time [Dutta et al. 1992]. With the increased use of parallel computers, it is naturally desirable to parallelize the synthesis process in order to design more complex and larger systems. In contrast to the previous work, the current work does not lose optimality in the solutions for efficiency in the design process. Object-oriented modeling and design techniques coupled with the parallel design process illustrate an efficient design environment as evidenced by practical implementation [Hsiung et al. 1998] and formal analysis [Hsiung et al. 1997a]. Solutions to the emptiness and deadlock problems found in a parallel design environment are also proposed.

3. PARALLEL OBJECT-ORIENTED SYNTHESIS ENVIRONMENT

The proposed design environment called *Parallel Object-Oriented Synthesis Environment* (POSE) mainly consists of two models: a static *system model* and a dynamic *synthesis model*. While the system model is based fully on object-oriented techniques, the synthesis model demonstrates an elegant combination of two kinds of design techniques, namely, *parallel* and *object-oriented*. POSE integrates the two models in such a way that, though they complement each other in functionalities and domain knowledge, they are still modularized, that is, one model can be modified or enhanced without any change to the other. POSE can be used to design any system at any level of synthesis as long as off-the-shelf building blocks or library of system parts [Tobias 1981] are utilized for synthesizing a system. As detailed in the next section, wide applicability and generality of purpose are some of the features of POSE.

3.1 System Model

Since POSE is a general design environment, the concepts within POSE can be applied to different target systems. For example, some typical target systems are *Parallel Computer Architectures* (PCA), *System-On-Chip* (SOC), and *Embedded Reactive Systems* (ERS). Different target systems require different design *primitives* and *constraints*.

Design primitives are basic components or building-blocks of a target system. PCA synthesis requires architectural components such as processing elements, RAM modules, cache modules, control units, processor-memory interconnections such as bus, mesh, etc., and communication

protocols or interfaces. SOC synthesis requires compatible IPs, interconnection modules, power supplying units, and communication interfaces. ERS synthesis requires hard cores, soft cores, firm cores, interfaces, I/O mechanisms, ASICs, and processors. Though, as described above, different target systems require different primitives, yet the concepts in statically modeling these primitives, dynamically manipulating them for synthesis, and maintaining a library of the primitives could be generalized. POSE provides such a generalization in the form an object-oriented class hierarchy as described later in this section.

Design constraints are the specifications that a synthesized system must satisfy. Corresponding to different target systems, we also have different design constraints. PCA designs must satisfy constraints such as maximum cost, minimum throughput, system utilization bounds, reliability, scalability, and fault-tolerance. SOC designs must satisfy constraints such as chip-area usage, maximum power consumption, and maximum overall cost. ERS designs must satisfy constraints such as real-time conditions, environment stimuli response, hardware-software trade-off, and maximum cost. Though, as described, different target systems must satisfy different constraints, yet these constraints can always be modeled into the design libraries and methodologies. POSE models design constraints into the objects in a class hierarchy as their data characteristics.

Physically, a hardware system can be perceived as a collection of parts (objects) interconnected by links (relationships). Logically, hardware system parts can be classified into a hierarchy based on structure or function. Hence, both physically and logically, a hardware system readily fits into the object-oriented model. The rest of this section will go into the details of how a hardware system can be actually modeled using OO techniques.

3.1.1 *Object-Oriented Structure*. As far as notations and terminologies are concerned, we basically follow Rumbaugh's *Object Modeling Technique* (OMT) [Rumbaugh et al. 1991] 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 [Parnas 1985] and Rumbaugh's Object Model in OMT. CH can be constructed either by a top-down system decomposition or a bottom-up system integration. In the top-down method, a desired system is gradually decomposed into subsystems, then into smaller parts, and finally into physically available components or primitives. In the bottom-up method, physical components are first classified into intercompatible and noncompatible ones, then a clustering process (usually the same as object-oriented grouping) results in higher level components such as a processor subsystem consisting of one or more processors, local memory, and an interconnection.

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 the *whole* in a "*whole-part*" relationship. An aggregate node is said to be *composed* of its child component nodes. A generalized node is a superclass which is the parent in an "*is-a*" relationship and represents the abstraction of more specialized subclasses. 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 Section 3.2.

A generic class has attributes including data members and function members. We classify the data members of a class into *specifications*, *predesign characteristics*, and *postdesign characteristics*, where a specification is a requirement, it may be a relation between several characteristics, a predesign characteristic is one whose value is known before design and a postdesign characteristic is one whose value is known only after design.

3.1.2 *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. The class being refined is called the *superclass* and its refined versions are called *subclasses*. 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 in ICOS 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 *postdesign 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. A component may have one or more of its specifications expressed in terms of the specifications of another component. Such specifications are called *dependent specifications*. For the synthesis of a component to be possible, all its dependent specifications must be updated by querying the component it is relatively dependent on. For example, if a component's cost is specified as a partial cost of another component, then the former must query the latter for the latter's cost in order to update its own cost. An absolute dependence between the components restricts the order in which the components are to be synthesized, whereas a relative dependence places no such restriction. 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 follows:

$$m = k \times p \tag{1}$$

where k is a constant and it is assumed that m is a specification and p a postdesign characteristic. Further, a `Cluster Control Unit` (`CCU`) and a `System Interconnect` (`SI`) are relatively dependent because the `CCU` *data transfer rate* (d_{CCU}) and the `SI` *data transfer rate* (d_{SI}) are related as follows:

$$d_{\mathrm{SI}} = c \times d_{\mathrm{CCU}} \tag{2}$$

where c is a constant and both d_{SI} and d_{CCU} are required *specifications*.

Each type of relationship allows us to perform different synthesis actions, thus the synthesis process is guided by the relationships. When an aggregation relationship is encountered, 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 subclasses 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 and before any synthesis actions are taken, a class must *first* update all of its dependent specifications by querying for specification values (such as cost, throughput, etc.) from classes that have relative dependence relationships with it. Thus, the dependence relationship has the *highest priority* among all relationships that a class may have with other classes. When it is an absolute dependence, that is, a class A is absolutely dependent on a class B, then class A must wait for the design completion of class B before class A can begin synthesis. When it is a relative dependence, that is, class A is relatively dependent on a class B, then class A need not wait for class B, but it must have all of its specifications assigned values by querying class B before class A can begin synthesis. It is this kind of relationship that allows an object-oriented synthesis system to design more than one component at the same time, i.e., *parallel synthesis*.

3.1.3 *Object-Oriented Operators*. As mentioned in the previous sections, 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. Figure 1 shows the correspondence between object-oriented structure, relationships, operators, and synthesis actions.

"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. For example, a processing subsystem is an aggregate of some processors, a local interconnection network, some memory modules, and perhaps a controller. The specifications of the processing subsystem may include a maximum

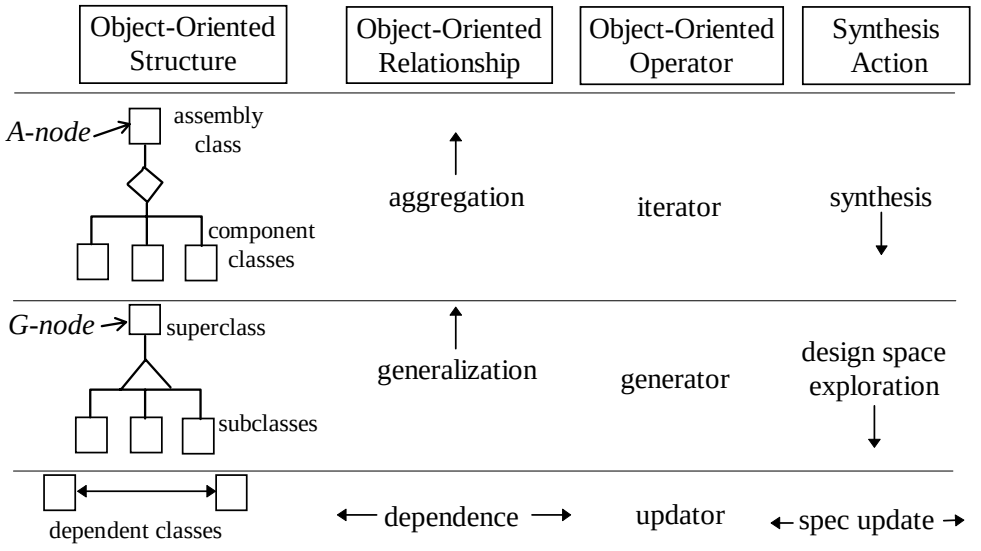


Fig. 1. Object-oriented structure, relationships, operators, and synthesis actions.

cost of $10,000, a minimum throughput of 2 MFlops, which may be satisfied if we use 4 processors, a shared bus, and 100 MB memory.

"Generator" is the design-space exploration operator, 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 [Hsiung et al. 1996] or a fuzzy comparison of specifications as in ICOS [Hsiung et al. 1998]. An example is the implementation of a system interconnection which is represented as a G-node having `Shared Bus`, `Multistage Interconnection Network` (`MIN`), and `Hypercube` as its specialized classes. If the specification sets a maximum cost of $2,000 and if `Shared Bus` and `MIN` can satisfy the cost specification, then one order of preference might be based on the throughput of the interconnection systems.

"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 because through this operator a sort of communication or information transfer is setup between the hardware components under synthesis, which is necessary if the components are to be integrated into a feasible working design. For example, the memory access time should be a multiple of the processor cycle time, hence the `Memory` class and the `CPU` class should communicate the values of their respective specifications so that they can be compatible when integrated or synthesized into the same system. The updator operator is used by a class whenever it has a dependence relationship with another class.

3.1.4 *Class Hierarchy*. In the above descriptions: *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 can 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 interrelated by the object-oriented relationships. This hierarchy is usually constructed a priori just as software library is constructed for future (re)use. CH must be constructed by a tool-vendor, which implements the hierarchy as a library or database of design primitives corresponding to the target system. For SOC target systems, CH would have a hierarchy of IPs, power units, and interconnections. For ERS target systems, CH would consists of ASICs, CPUs, environment interfaces, I/O mechanisms, etc. Tool vendors must create all these Class Hierarchies as reusable libraries for system designers. An example of Class Hierarchy for a hierarchical parallel computer system is given in Figure 2. The purpose of this hierarchy is to serve as a framework in which synthesis proceeds. The main concept of object-oriented synthesis can be defined as:

Definition 1. 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.

3.2 Synthesis Model

Before going into the details of how OO design techniques are used in the synthesis process, we will first discuss the design flow in a typical synthesis methodology. We mainly refer to ICOS, and show how POSE has been applied in ICOS.

As shown in Figure 3, the ICOS methodology is divided into three design phases called *Specification Analysis*, *Concurrent Design*, and *System Integration*. ICOS provides a specification language for architecture, performance, and synthesis related specification input. Details can be found in Hsiung et al. [1998].

In the specification analysis phase, a user's system specifications are first analyzed to check if there are any errors such as architecture related or obvious ones such as constraints that are not feasible under current technology. In the concurrent design phase, the main system synthesis is performed. Here, components (including complete subsystems) are designed concurrently. POSE is a generalization of the concurrent design techniques used in this phase of ICOS. Generalizations include the different messages that objects pass among themselves, the design completion check (see Section 4.1), and the synthesis rollback mechanisms (see Section 4.2). A design hierarchy and a design queue are utilized in ICOS in this design phase for recording the current design status. The final phase of system

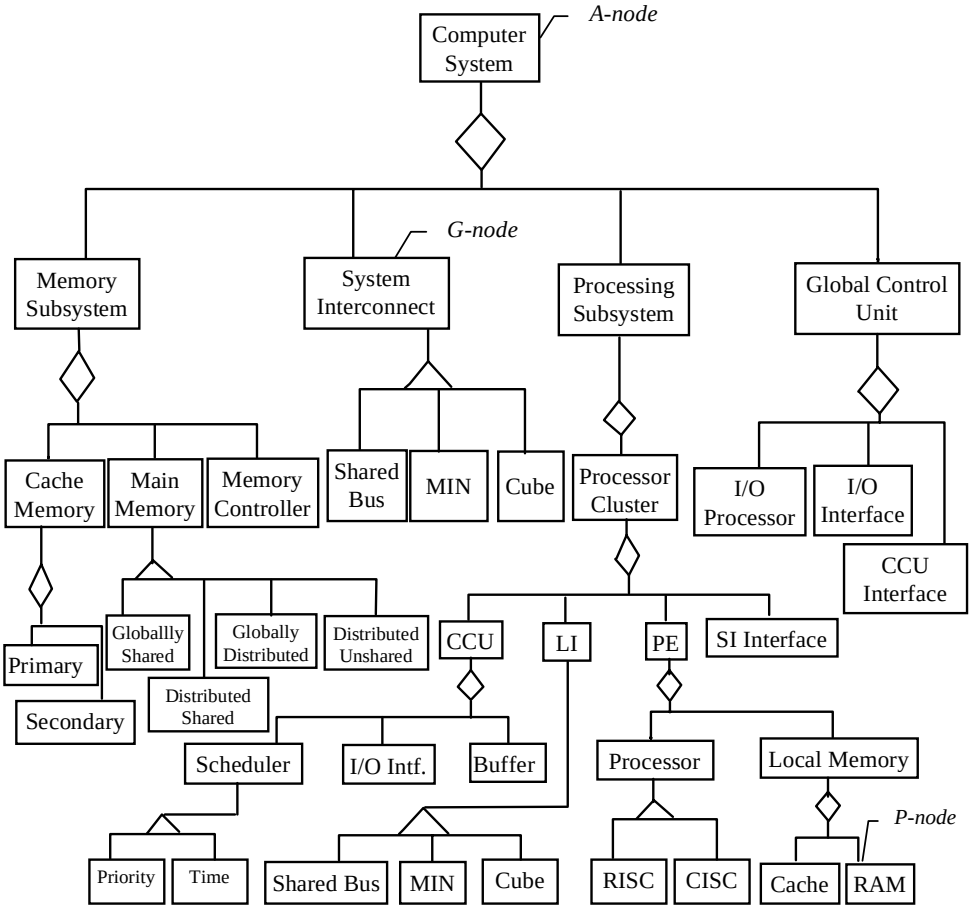


Fig. 2. Class hierarchy.

integration mainly evaluates the performance of a completed design and outputs the system design that best meets the design constraints.

POSE is applied mainly in the concurrent component design phase of ICOS. Here, components are designed *concurrently* and in an *active* manner, as described in this section. The work presented in ICOS [Hsiung et al. 1998] focussed mainly on what is the methodology for designing a parallel system, whereas here in POSE we show how the design environment can be generalized, formalized, and applied to other design methodologies. Thus, technical details of ICOS are left out and a more general design environment is presented. POSE shows how high-level design space explorations may be carried out, how existing sequential design methodologies may be enhanced by parallelism, and how OO can actually be applied to hardware design.

The model used for synthesis is based mainly on object-oriented and parallel design techniques. In the system model, as described in the previous section, each system part is modeled as an object class. In the

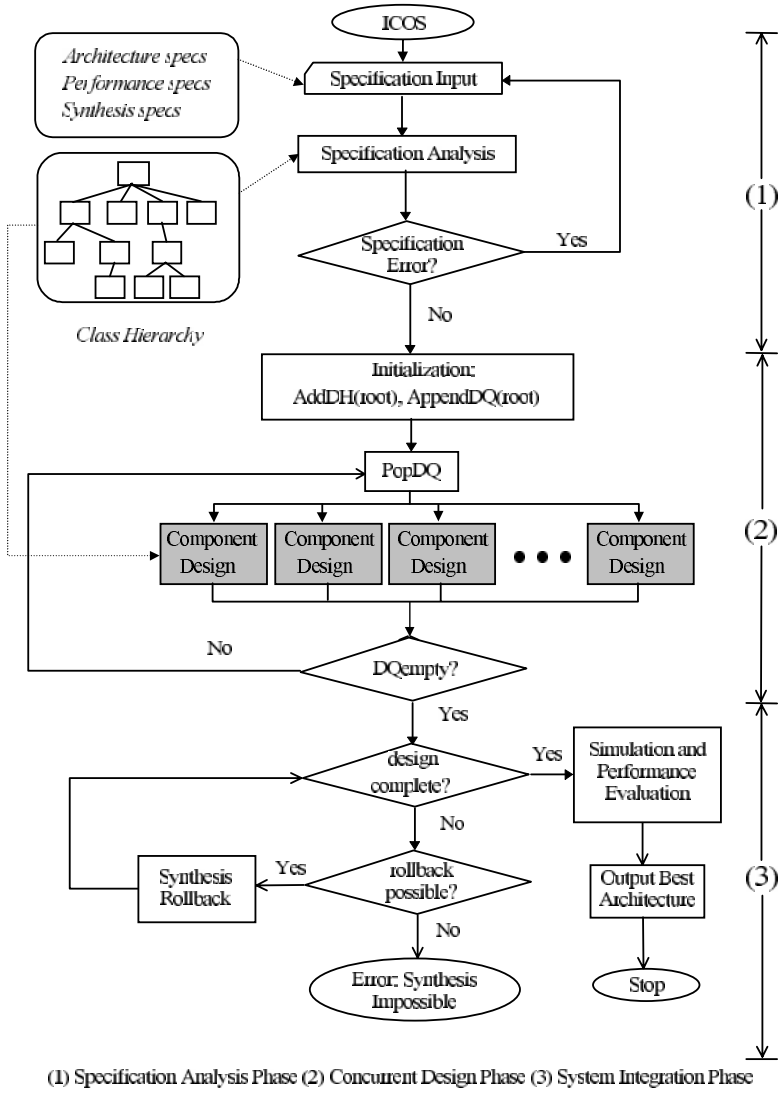


Fig. 3. ICOS design flow.

synthesis model, these object classes *actively* synthesize themselves using a *message-passing communication scheme*. Totally, eight types of messages are used for different types of communication. Classified into three groups, there are *synthesis-related*, *update-related*, and *rollback-related* messages. There are three synthesis-related messages, namely, `synthesize` (m_s), `synthesis-complete` (m_{sc}), and `synthesis-incomplete` (m_{si}) messages; two update-related messages, namely, `update` (m_u) and `update-complete` (m_{uc}) messages; and three rollback-related messages, namely, `rollback` (m_r), `rollback-complete` (m_{rc}), and `rollback-incomplete` (m_{ri}) messages. These messages are all implemented as method invocations in each

individual object. The roles played by these messages will be discussed in this section and Section 4.

Active synthesis is an important feature of the synthesis model in POSE. Each system part, modeled as an object class, does not get *passively* synthesized, it instead *actively* seeks to synthesize itself by sending and receiving messages. This kind of synthesis control reduces the overhead of a synthesis kernel, whose main job now becomes the maintenance of the current design status and consistency. A reduced synthesis kernel allows greater scalability in terms of the complexity of systems synthesized. Within the underlying object-oriented system model, the proposed active synthesis approach further strengthens the encapsulation of objects such that not only are the static features (the data and the function members) encapsulated as an object, but also are the dynamic states (the synthesis status) of an object encapsulated. Parallel synthesis, thus becomes a product of the object-oriented system model and the active synthesis approach.

The rest of this section goes into the details of how the object actively synthesize themselves by sending and receiving messages within the OO design environment.

3.2.1 *Synthesis Process*. Whenever an object receives a `synthesize` message in the form of a method invocation, it first checks if it is *associated* with any other object. Here, association is the dependence relationship as described in Section 3.1.3. In case of no association, synthesis actions are performed depending on its node-type; whereas if there is one or more associations, the object must actively send messages to all of the objects associated with it. Three types of messages are used in this process, namely, `synthesize`, `update`, and `update-complete` messages. In the following, cases (a), (b), and (c) describe objects with no association, while case (d) describes objects with at least one association as shown in Figure 4.

Case (a) *A-node with no association with other objects*: On receiving a `synthesize` message, an A-node with no association with other objects uses the *iterator* operator to synthesize itself. It iterates through its child objects (which are sub-classes of the aggregate class representing the A-node under synthesis), checking which objects are required to synthesize itself. It then simultaneously sends a `synthesize` message to *each* of the selected objects. The selection policy may be based on a simple functional composition as in PSM or a complex fuzzy decision as in ICOS.

Case (b) *G-node with no association with other objects*: On receiving a `synthesize` message, a G-node with no association with other objects uses the *generator* operator to implement itself by sending a `synthesize` message to a child object. The child objects (which are specialized classes of the generalized class representing the G-node) are selected based on a straightforward cost-based heuristic approach as in PSM or a fuzzy design-space exploration method as in ICOS. The number of specializations used to implement each G-node may be a complete set considering

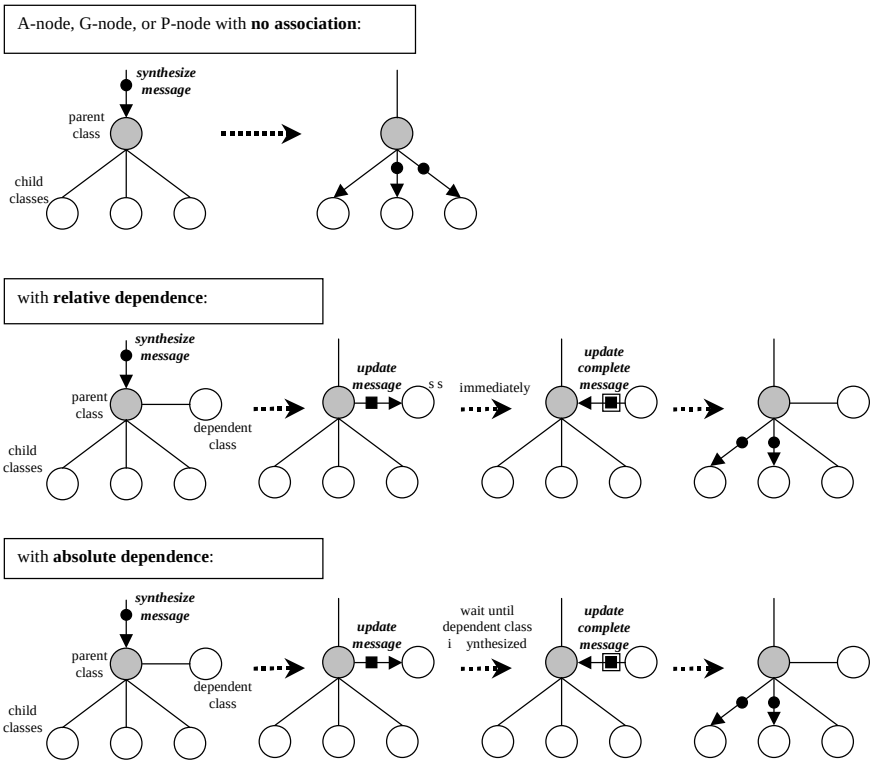


Fig. 4. Synthesis process.

all available design options for synthesizing an optimal solution or a partial set of only the most preferable ones, where preference can be defined as the heuristic proximity of design points near the optimal solution as in PSM or defined in terms of the overall degree of specification satisfaction as in ICOS.

Case (c) *P-node with no association with other objects*: On receiving a `synthesize` message, a P-node with no association with other objects instantiates itself by selecting one or more physical instances from the physical object library. Instantiation of the physical objects may either consider all the instances available as in PSM or a subset of more preferable ones in terms of specification satisfaction as in ICOS. Synthesizing a P-node is almost like a G-node except that we can consider more instances at a P-node without increasing the design space by an exponential factor.

Case (d) *Any node with at least one association with other objects*: On receiving a `synthesize` message, an object with at least one association with other objects must update all of its specifications by sending an `update` message to each associated object. When an object receives an `update` message, if the association is only a *relative dependence*, an

acknowledging `update-complete` message is returned immediately along with the values of the predesign characteristics that were requested. If it is an *absolute dependence*, then the object on which the current object under synthesis is dependent should have *already completed* synthesis for the current object to begin synthesis immediately, otherwise, the current object must wait for that object to finish synthesis. When the object receiving an update message is already synthesized it will send an acknowledging `update-complete` message notifying the update-requesting object about its completion of synthesis along with the values of the postdesign characteristics and/or specifications that were requested. After the update process, when all the `update` messages are acknowledged, the behavior of an object under synthesis is the same as in the above three cases.

The above synthesis process assures that each object will know how to synthesize itself and at the same time maintain the dependence relationships amongst themselves.

3.2.2 *Synthesis Kernel*. The synthesis kernel is responsible for design consistency and status maintenance. Some data structures used in the POSE synthesis kernel and the maintenance of the design state for an object are described in this section.

3.2.2.1 *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, Figure 5 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 referring 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 redesign 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 Class Hierarchy. Here, Design Hierarchy comes into use since it is possible to know exactly what components were used and what were their specifications that necessitated rolling back the synthesis steps.

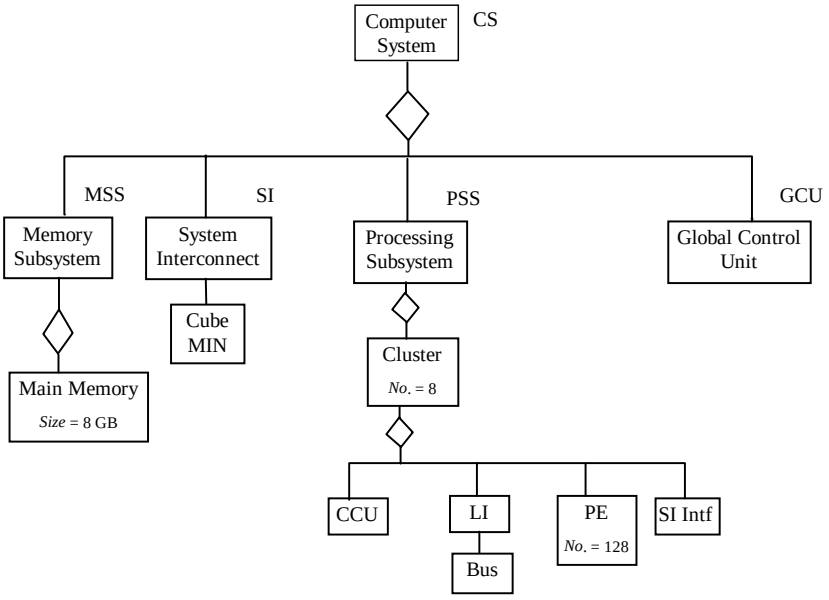


Fig. 5. Design hierarchy.

(3) *Design Completion Check*: Design Hierarchy can also indicate when a design alternative is complete for further processing such as simulation and performance evaluation.

3.2.2.2 *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 then have to be appended to DQ. Whenever a P-node results from some component synthesis process, it is not appended to the queue, but instead is instantiated into actual physical instances. For example, a RISC processor may be instantiated into an Alpha-21064 CPU. Design Queue does not sequentialize synthesis, it just sequentializes the processing of components for synthesis since more than one component may be undergoing synthesis after removal from the queue, that is, a component once removed from the queue need not be completely synthesized before the next component is removed from the queue.

3.2.2.3 *Design State*. The above DH and DQ are global structures that are 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,

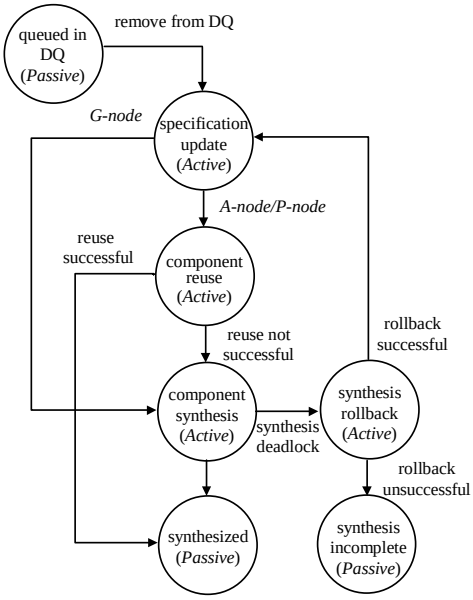


Fig. 6. State transition diagram of a class.

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 6 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 components that were designed before. 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 rollback is successful. If a component is reused or synthesized successfully, it enters DH and remains passive. Otherwise, it enters a passive synthesis incomplete state.

Using the above described data structures, a synthesis kernel keeps track of the design states and consistency, while each system part actively synthesizes itself. But, there still might be some problems related to the systems specifications. The common problems are described and solutions presented in the next section.

4. SOME RELATED PROBLEMS

Parallel Object-Oriented Synthesis Environment, just like any other parallel environment, must solve problems inherent to parallelism. This section defines and solves two related problems, namely, *emptiness* and *deadlock*. A designer must know as early as possible if the given specifications are

feasible under the current technology. This is the emptiness problem. At the same time, a top-down design environment must be able to ensure that all specifications propagated from the top of the design hierarchy are satisfied by each designed part. In case of infeasible specifications, a deadlock occurs.

4.1 Design Completion Check

A system designer stipulates his or her requirements by giving system-level specifications including performance-related constraints such as the minimum throughput, the maximum cost, the utilization factor, and architecture-related constraints such as the system interconnection, the amount of main memory, the memory organization, etc. In terms of the current technology, the given specifications may either be feasible or infeasible. Feasibility is defined in terms of satisfying all the system constraints under the current technology. Current technology is generally encoded within each object, for example the maximum possible power allowed, the heat dissipation rate, the response time, maximum network bandwidth, etc.

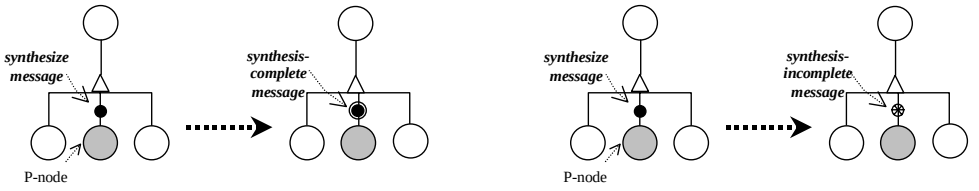
Sometimes, infeasibility due to obvious contradictions and errors is detected easily by a predesign specification analysis. But at times, infeasibility may go undetected until the design process has well advanced into some intermediate stage. Hence, it is desirable to detect infeasible specifications at the earliest-possible stage of the design process. We call this the *emptiness* problem. A practical solution called the *design-completion check* for the emptiness problem is presented within POSE.

Definition 2 (The Emptiness Problem). Given a set of performance and architecture specifications, is there a system design that can satisfy all the stipulated specifications and is feasible under the current hardware technology.

Since the synthesis process is a distributed-control parallel-design process, a mechanism is needed to ensure that a particular design is either feasible and complete or infeasible and incomplete. This design completion check (DCC) process is accomplished using two types of messages or method invocation calls, namely, `synthesis-complete` and `synthesis-incomplete` messages.

As the `synthesize` message is gradually propagated in a top-down direction and broadcast in a breadth-first-search hierarchy traversal, a `synthesize` message eventually reaches a P-node at the hierarchy leaf. The three types of nodes behave in the following ways:

(a) Whenever a P-node receives a `synthesize` message, it starts to instantiate itself. If this instantiation process is feasible and complete, the P-node sends a `synthesis-complete` message upwards to its parent node in the Class Hierarchy; otherwise, the P-node performs a *synthesis rollback* action as described in the next section. If the rollback process also fails, the P-node then sends a `synthesis-incomplete` message to its parent node. This process is depicted in Figure 7.



Case (a): P-node is feasible Case (b): P-node is infeasible

Fig. 7. Design completion check process for a P-node.

(b) Whenever a G-node receives at least one `synthesis-complete` message from one of its child objects and `synthesis-incomplete` messages from the other child objects, the G-node sends a `synthesis-complete` message to its parent object. If no `synthesis-complete` message is received from its child nodes, then the G-node performs a *synthesis rollback* action (as described in the next section). If the rollback process also fails, the G-node sends a `synthesis-incomplete` message to its parent object. This process is illustrated in Figure 8.

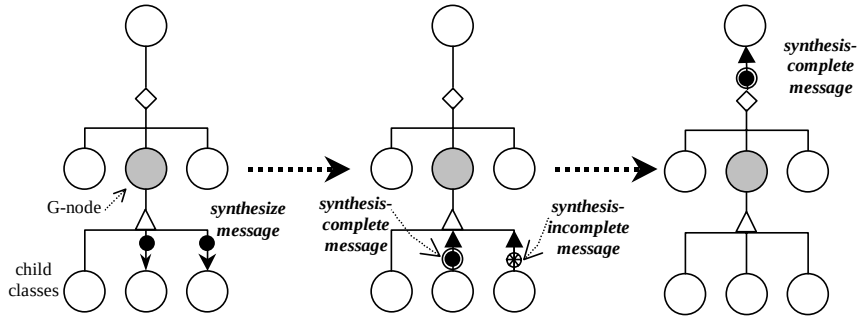
(c) Whenever an A-node receives a `synthesis-complete` message from each of its child objects that has been sent a `synthesize` message, the A-node sends a `synthesis-complete` message to its parent object. If the A-node receives a `synthesis-incomplete` message from any one or more of its child objects, it first performs a *synthesis rollback* action (as described in the next section). If this rollback process also fails then a `synthesis-incomplete` message is sent upwards to its parent class. This process is shown in Figure 9.

The above design completion check process will finally result in either a `synthesis-complete` or a `synthesis-incomplete` message being received at the root node that represents the whole computer system. In the former case, the design is feasible under the current constraints and specifications, while in the latter case, it is infeasible and incomplete.

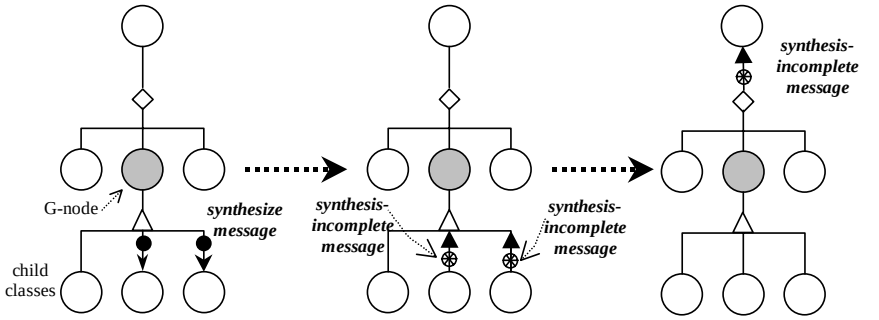
4.2 Synthesis Rollback

Since POSE is a top-down parallel design environment, specifications are propagated from the top of the class hierarchy towards the leaf physical classes. In the course of this propagation, it may happen that some component cannot be synthesized under the derived specifications as propagated by its parent class. This is called the *deadlock* problem since the parent requires its child to satisfy certain specifications while the child cannot do so. A *rollback mechanism* is proposed to solve this problem in POSE.

Definition 3 (The Deadlock Problem). When a child class (or component) cannot be synthesized under the derived specifications as propagated by its parent class (or component), a *deadlock* occurs.



Case (a): synthesis complete for at least one child class of G-node



Case (b): synthesis incomplete for all child classes of G-node

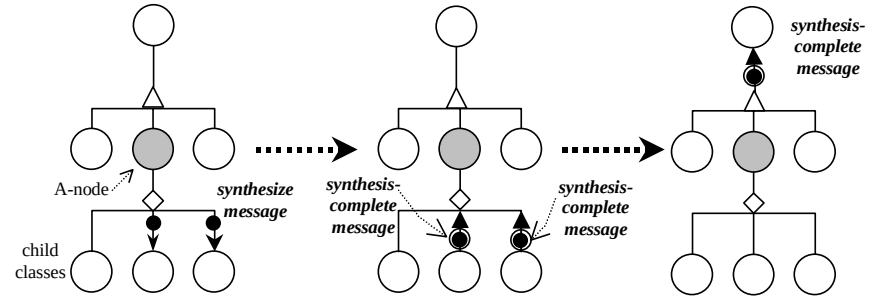
Fig. 8. Design completion check process for a G-node.

As mentioned in the previous section, the synthesis rollback process is interleaved with the design completion check process. Three types of messages, namely, `rollback`, `rollback-complete`, and `rollback-incomplete` messages, are used in this process which is illustrated in Figure 10.

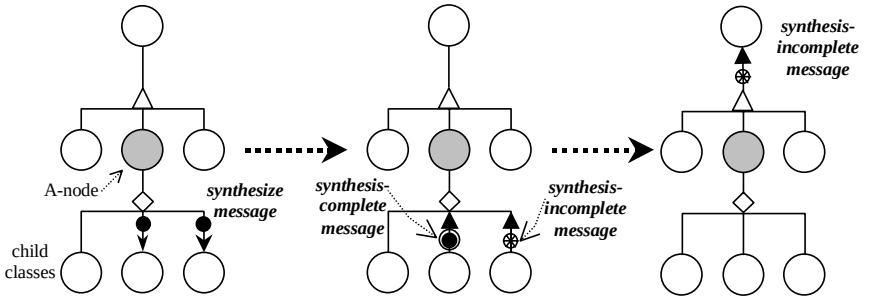
Whenever an object, either an A-node or a G-node, receives a `rollback` message along with the object characteristics that have triggered the rollback and the range of values acceptable for each of the object characteristics, the object then behaves as follows:

Case (a) If by resynthesizing itself, the object can relax the infeasible specifications and characteristics to satisfy all the constraints of the parent object, the associated object(s), and the child object(s), then it will resynthesize itself and relax the specifications and characteristics. After synthesis, the object will send a confirmation in the form of a `rollback-complete` message to the sender of the `rollback` message. This is shown graphically in Figure 10-Case (a).

Case (b) If the object receiving a rollback message cannot relax the infeasible specifications and the specifications are not related to either the parent class or any associated classes, the object sends a `rollback-`



Case (a): synthesis complete for all under-synthesis child classes of A-node



Case (b): synthesis incomplete for at least one child class of A-node

Fig. 9. Design completion check process for an A-node.

`incomplete` message to the sender of the `rollback` message. The rollback is unsuccessful in this case. This is shown graphically in Figure 10-Case (b).

Case (c) If the object cannot relax the infeasible specifications but they are related to other objects, then it will propagate the `rollback` message along with the related information to its parent object and/or the associated objects. One or more `rollback` messages are sent simultaneously. If at least one rollback completes, then the rollback is successful, otherwise the rollback is unsuccessful and a `rollback-incomplete` message is sent to the rollback message sender. On receiving a `rollback-complete` message, if the object itself had been the receiver of a `rollback` message, it will pass on the `rollback-complete` message to the rollback message sender. In this way, all rollbacks are confirmed and finally the original message sender will receive a confirmation. This process is shown graphically in Figure 10-Case (c).

5. APPLICATION

The proposed environment, POSE, has been implemented as a working design framework in which the recently proposed *Intelligent Concurrent Object-oriented Synthesis* (ICOS) methodology [Hsiung et al. 1998] for

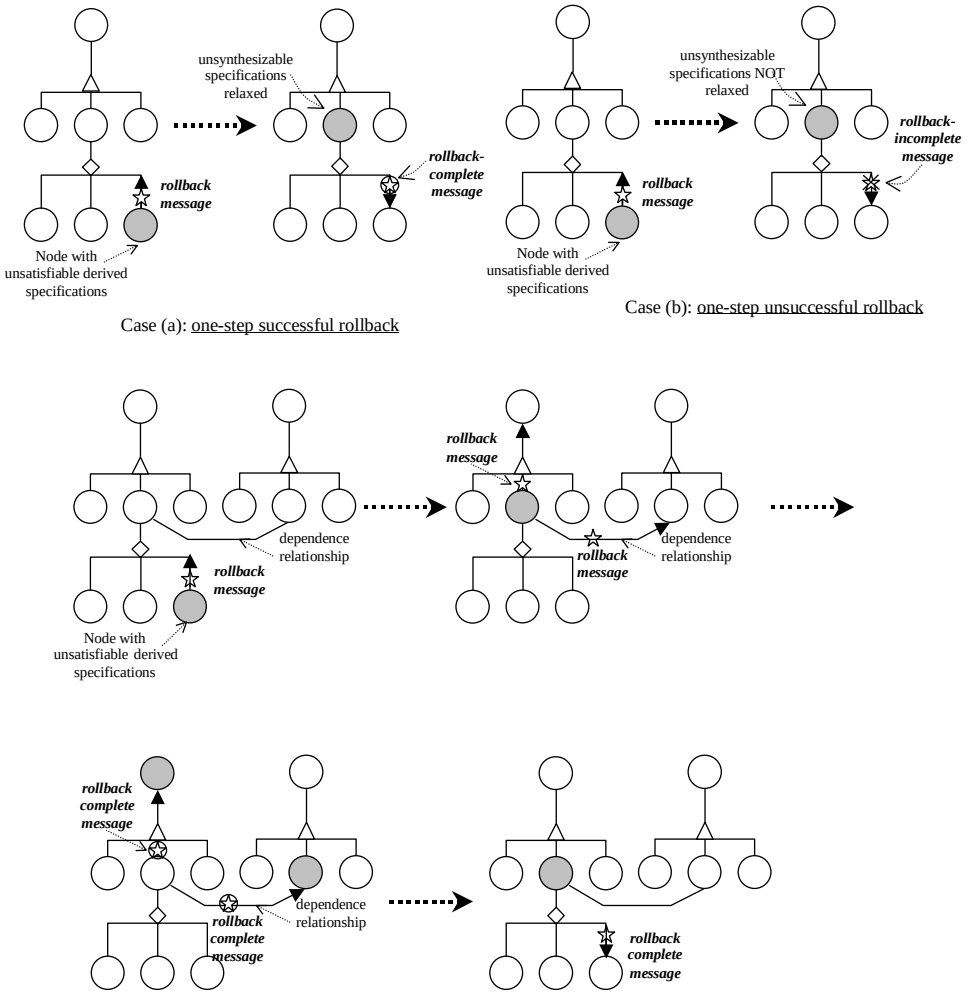


Fig. 10. Rollback process in POSE.

system-level synthesis of parallel systems was developed. Besides practical implementation, POSE has also been theoretically modeled, validated, and analyzed by the authors using high-level Petri nets called *Multi-token Object-oriented Bi-directional net* (MOBnet) [Hsiung et al. 1997a]. ICOS and MOBnet were respectively concerned with the synthesis methodology and the model analysis. The general design environment incorporating parallel and OO techniques was not presented before. Two examples are given in the rest of this section to understand the step-wise design evolution in POSE and the two processes of the design completion check and synthesis rollback.

Table I. Design Steps of an Illustrative Example

Step	Node	Class Type	Relation	Operator	Action	DQ Status
(a)	CS	*A*	aggregation	iterator	reuse	{MSS,SI,PSS,GCU}
(b)	MSS	*A*	aggregation	iterator	synthesis	{SI,PSS,GCU}
(c)	SI	*G*	generalization	generator	DSE	{PSS,GCU}
(d)	PSS	*A*	aggregation	iterator	synthesis	{GCU,Cluster}
(e)	GCU	*A*	aggregation	iterator	reuse	{Cluster}
(f)	Cluster	*A*	aggregation	iterator	synthesis	{CCU,LI,PE}
(g)	CCU	*A*	aggregation	iterator	reuse	{LI,PE}
(h)	LI	*G*	generalization	generator	DSE	{PE}
(i)	PE	*A*	aggregation	iterator	reuse	{}

5.1 Example 1

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 I and the intermediate status of DH are given in Figure 11, where CH and LH are referenced during synthesis. LH is the Learning Hierarchy used in ICOS.

Starting from the root node, the target system is iteratively synthesized by traversing down the Class Hierarchy towards the leaf nodes (P-nodes). Depending on the type of node (A, G, or P), each component is either synthesized or implemented using relationships as guidelines and operators to perform corresponding actions. 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 I. 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.

The above is a description of how objects get synthesized in POSE. Parallelism is achieved in POSE by simultaneously starting steps (b), (c),

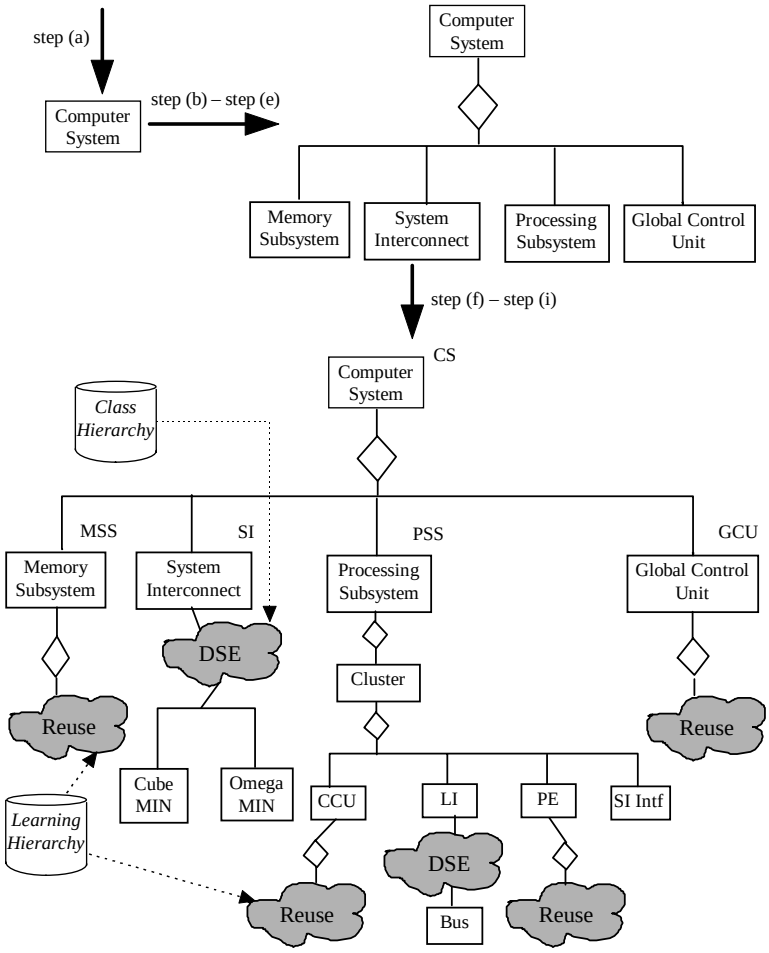


Fig. 11. Example 1 (intermediate DH status).

(d), (e) after step (a) completes. All the four objects can be dequeued and start synthesizing themselves in parallel. Next, steps (f), (g), (h), (i) can also start in parallel. Thus, there is actually only three parallel steps compared to the nine sequential steps. Readers interested in a detailed account of the actual synthesis methodology may refer to ICOS [Hsiung et al. 1998].

As shown in ICOS, when system parts are modeled as objects and synthesized in parallel, the overall design time is between one-half to one-third the time required when components are synthesized sequentially. The experimental results obtained show the practical usefulness of POSE in a real design methodology. We believe that POSE can also be incorporated with any other design methodology as long as off-the-shelf building blocks are used and modeled using OO technology.

5.2 Example 2

This example illustrates how the three processes: *synthesis*, *design completion check*, and *rollback*, work together in POSE to design a system. The target multiprocessor system is an SIMD machine with a `Multistage Interconnection Network` (`MIN`) as the system interconnection. The system specifications include a *data transfer rate* of at least 8 MB/s for each block of `MIN`, a total throughput of at least 64 MB/s, and a maximum total cost of $7,000. These specifications are input to ICOS. We will mainly concentrate on how POSE handles situations with unsynthesizable specifications. As shown in Figure 12, (a) and (b) illustrate the synthesis process, (c) and (d) illustrate design completion check, (d) and (e) illustrate the specification update process, (f)–(j) illustrate synthesis rollback. Since parts (a)–(e) of Figure 12 are straightforward and self-explanatory, we only explain in detail the parts (f)–(j).

Here, the derived specifications from `Processing Cluster` (`PC`) for `System Interconnect Interface` (`SI-Intf`) include a maximum *cost* of $200, which could only result in an interface that has a maximum *data transfer rate* of only 8.0 MB/s. This is in contradiction to the *data transfer rate* specification *updated* from `System Interconnect` (`SI`) in Figure 12(e) (which is 8.5 MB/s). Thus, `SI-Intf` is unsynthesizable in such a situation. It makes requests for synthesis rollback to both `PC` and `SI` (Figure 12(f)). `PC` cannot increase the cost of `SI-Intf` due to other cost constraints (Figure 12(g)). Meanwhile, `SI` propagates the rollback message to `MIN` (Figure 12(g)). `MIN` decreases the *data transfer rate* specification to 8 MB/s as required (Figure 12(h)). Finally, rollback completes informing `SI-Intf` of the change in the *data transfer rate* specification value (Figure 12(i)). Thus, `SI-Intf` can now synthesize itself under the derived *cost* specification of $200 and updated *data transfer rate* specification of 8 MB/s (Figure 12(j)).

6. CONCLUSION AND FUTURE WORK

A general design environment for the synthesis of computer hardware systems was proposed. *Parallel Object-Oriented Synthesis Environment* (POSE) used parallel design techniques to synthesize system parts which were modeled using object-oriented techniques. Being a general environment, one of its advantages is that it can be easily incorporated into any system design automation methodology such as PSM [Hsiung et al. 1996] and ICOS [Hsiung et al. 1998]. Experimental results of using POSE in ICOS has shown how the overall design time can be sped up by a factor of two to three as compared to PSM. Besides the practical efficiency, it was also shown how POSE can be used to solve two parallel design related problems, namely, the emptiness and the deadlock problems. POSE was also formally validated and analyzed in another related work [Hsiung et al. 1997a]. Both experimentally and theoretically, POSE was shown to be a useful working design environment.

Future work will be the application of POSE to hardware-software system codesign and its incorporation with other synthesis methodologies.

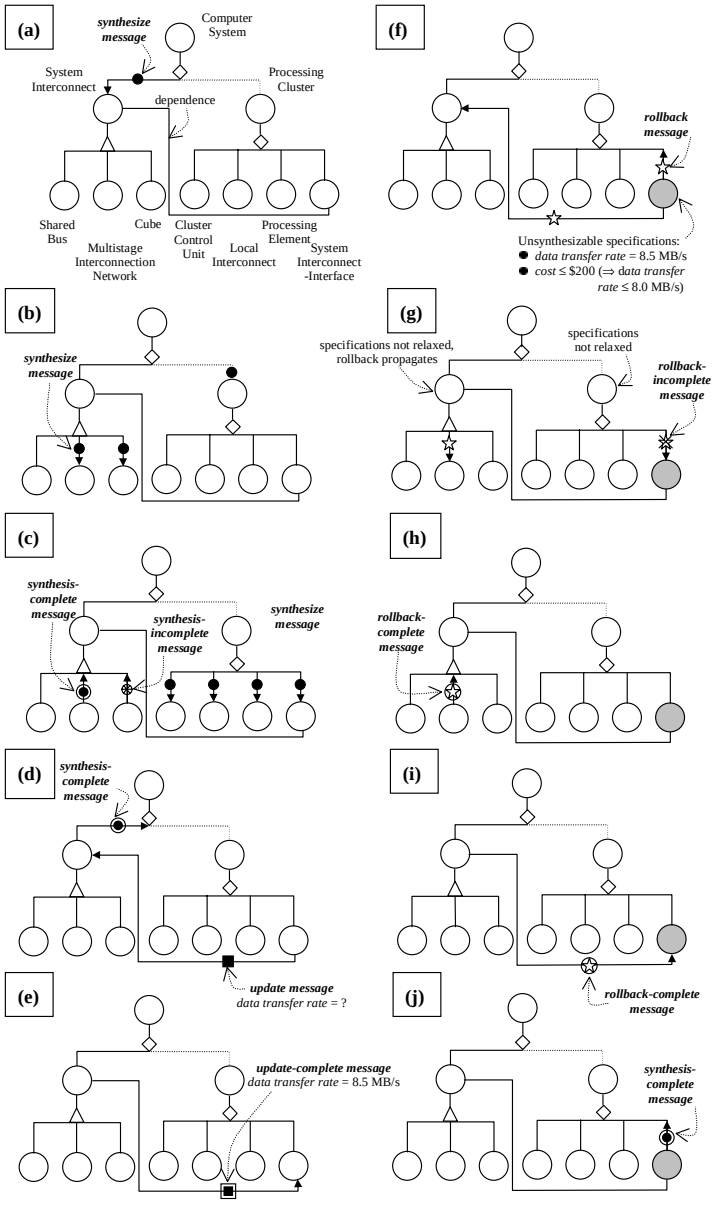


Fig. 12. Example 2.

REFERENCES

DE ANTONELLIS, V. AND PERNICE, 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), D. E. Thomas, Ed. ACM Press, New York, NY, 135–140.

BROOKS, F., GROSS, R. R., AND HEATH, L. S. 1984. Transfer of software methodology to VLSI design. TR 84-007. University of North Carolina at Chapel Hill, Chapel Hill, NC.

CHUNG, M. J. AND KIM, S. 1990. An object-oriented VHDL design environment. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation* (DAC '90, Orlando, FL, June 24-28), R. C. Smith, Chair. 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), D. G. Schweikert, Chair. 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.

GROSS, R. R. 1985. Using software technology to specify abstract interfaces in VLSI design. TR-85-017. University of North Carolina at Chapel Hill, Chapel Hill, NC.

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., CHEN, C.-H., LEE, T.-Y., AND CHEN, S.-J. 1998. ICOS: an intelligent concurrent object-oriented synthesis methodology for multiprocessor systems. *ACM Trans. Des. Autom. Electron. Syst. 3*, 2 (Apr.), 109–135.

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, 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.

HSIUNG, P. -A., LEE, T. -Y., AND CHEN, S. -J. 1997. Object-oriented technology transfer to multiprocessor system-level synthesis. In *Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems* (Sept.). 284–293.

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.

KUMAR, S., AYLOR, J. H., JOHNSON, B. W., AND WULF, WM. A. 1994. Object-oriented techniques in hardware design. *IEEE 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), D. G. Schweikert, Chair. IEEE Computer Society Press, Los Alamitos, CA, 153–158.

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.

PARNAS, D. L. 1985. The modular structure of complex systems. *IEEE Trans. Softw. Eng. SE-11*, 3 (Mar.), 259–266.

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., LORENSEN, B., AND LORENSON, W. 1991. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ.

THOMAS, D. E., DIRKES, E. M., WALKER, R. A., RAJAN, J. V., NESTOR, J. A., AND BLACKBURN, R. L. 1988. The system architect's workbench. In *Proceedings of the 25th ACM/IEEE Conference on Design Automation* (DAC '88, Atlantic City, NJ, June 12–15), D. W. Shaklee and A. R. Newton, Chairs. IEEE Computer Society Press, Los Alamitos, CA, 337–343.

TOBIAS, J. R. 1981. LSI/VLSI building blocks. *IEEE Computer 14*, 8 (Aug.), 83–101.

TRICK, M. T. AND DIRECTOR, S. W. 1989. Lassie: Structure to layout for behavioral synthesis tools. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation* (DAC '89, Las Vegas, NV, June 25–29), D. E. Thomas, Ed. ACM Press, New York, NY, 104–109.

Received: May 1998; accepted: August 1999