Using Software Agents to Generate VLSI Layouts

Dilvan de Abreu Moreira, University of São Paulo
Les T. Walczowski, University of Kent at Canterbury

SOFTWARE AGENTS CAN PROVIDE FLEXIBILITY AND SCALABILITY IN A CLIENT-SERVER ARCHITECTURE. TO DEMONSTRATE HOW AGENTS CAN AID LAYOUT GENERATION, THE AUTHORS CREATED AGENTS, A SCALABLE DISTRIBUTED SYSTEM THAT GENERATES FULL-CUSTOM DIGITAL AND MIXED-SIGNAL VLSI LAYOUTS IMPLEMENTED IN CMOS OR BICMOS TECHNOLOGIES.

AGENTS ARE SOFTWARE COMPONENTS that communicate with their peers by exchanging messages in a communication language.[1] Working cooperatively, agents can generate a flexible VLSI layout and solve its placement-routing problem. We can divide this type of agent system, on one level, into a few big agents and distribute them over a computer network, taking advantage of the computer power to generate good layouts faster. At a lower level, small, relatively simple agents can work together in groups, inside the big agents, to accomplish complex tasks.

We've incorporated these concepts into our Agents system,[2] a set of programs that automatically generates full-custom digital and mixed-signal VLSI (Very Large Scale Integration) layouts implemented in CMOS (complementary metal-oxide semiconductor) or Bicmos (bipolar complementary metal-oxide semiconductor) technologies. Our design follows the philosophy that competence will emerge from collective agent behavior.

Structuring Agents

Nature has many examples of distributed-reasoning systems. Bees have a set of simple individual behaviors that add up to very complex group behaviors in the swarm. The behavior-system approach starts with the view of behaviors as the fundamental unit of analysis. A *behavior* is a regularity in the interaction dynamics between an agent and its environment.[3] For example, an agent might maintain a certain distance from a wall. As long as this regularity holds, observers can say that there is an obstacle-avoidance behavior.

An agent needs some sort of mechanism to realize a behavior. This mechanism's implementation should consist of different components and a control program. Observed behaviors are due to the interaction between the mechanism's operation and the environment the agent is experiencing. A *behavior system*, then, is the collection of components responsible for realizing a particular behavior.

Like ants or bees, behavior-based agents can work together to have more complex group behaviors. A small number of our agents will coordinate Agents' system actions. Other agents will search for problem solutions, without much awareness about the system's final goals (see the "Small behavior-based agents" sidebar). Together, they will supervise all the layout-generation tasks.

Agents comprises a broker server and three server program types: *Placer*, *Router*, and *Database*. The server programs communicate over the network using the EDIF (electronic design interchange format) language to exchange design information. We chose EDIF because developers commonly use it to represent electronic designs in commercial CAD tools.

Broker server. An important trend in the computer industry is toward system architectures

0885-9000/97/$10.00 © 1997 IEEE

Small behavior-based agents

A small agent's main task is to carry out reasoning. An agent's simple individual behaviors combine to form complex group behaviors. We implement our small agents as C++ objects, which we categorize by function. A base class holds the basic inference routines, and the derived classes add the particular knowledge needed for each application.

We embedded in these small agents a mechanism based on Newell's search and problem spaces theory.[1] He reviews cognitive-science foundation concepts and makes a case for unified theories by describing a candidate: a general-cognition architecture called Soar.

Search and problem spaces. A system displays intelligent behavior when it uses its knowledge to attain its goals. This process takes basically the form of a search. Search, in this case, is not another method or cognitive mechanism, but a fundamental process for intelligent behavior.[1]

If an intelligent agent wishes to attain a goal, it must formulate this task. It does this on the basis of available knowledge. The more problematical a situation, the less knowledge is available that says how to attain the goal. Formulating a task that makes the least demands on specific knowledge, involves creating and searching a space that contains the goal. A space, in this case, is the set of all the problem's possible solutions. As the available knowledge decreases, this problem space becomes larger and more difficult and expensive to search.

Newell divides the knowledge an agent uses to search a problem space in two types: *task knowledge* and *search-control knowledge*.[2] Task knowledge comprises the initial state, the desired state (or any means to detect it), and the operators. Using just this knowledge, an agent can find a solution by exhaustively searching the problem space until it finds the goal state. This can be very inefficient. Search-control knowledge specifies which operator to take from a given state, directing the search to the desired goal. If a system doesn't have enough search-control knowledge, task knowledge acquires additional knowledge through searching.

An agent in the problem space must have knowledge to pursue its goals. For example, an agent should know how to propose and chose operators. When an agent does not have the knowledge to implement one function, an impasse occurs, and no further problem solving can be undertaken until knowledge is generated to solve this impasse. Impasses are solved by formulating a subgoal to acquire missing knowledge. The agent sets up the subgoal as a task to be solved within another problem space or can delegate it to other agents. Impasses can occur in any problem space, forming a goal and subgoal hierarchy with spaces and subspaces in one or multiple agents.

The knowledge to implement the problem-space computational-model functions is expressed in production rules. To create or change problem spaces, states, or operators, these rules propose values or express preferences for selecting values. To make a choice based on preference, the system applies knowledge to propose choices and then applies knowledge to order the choices by preference. Once the system has applied all available knowledge, it chooses the highest-ranked value.

Figure A shows the architecture of a C++ agent object; in this case, the object is controlling a robot. The *goals list* contains the current problem-space hierarchy, organized in a goal-context stack. Each goal context contains a goal, the problem space to search for that goal, the state slot of the program space, and the current applied operator. The *preference* list contains values proposed by the rules with their respective preferences. *Internal variables* are any kind of variables or objects held by a particular agent. The *in* and *out* triangles represent access to other objects or variables outside the agent object.

Task and search-control knowledge are encoded as production rules in permanent memory. These rules test the state of the goals list, internal variables, and the outside world. When fired, the rules can produce preferences for changing the goals-list elements or act on the internal variables and the outside world. Each element in the goals list is a list representing one goal and a problem space. When there is an impasse, the system automatically creates a new goal and problem space in the goals list, with data about the impasse.

Agent objects (and Soar) are different from other common cognitive architectures or AI shells in that they do not make arbitrary decisions about what to do next. There are no built-in conflict-resolution mechanisms. Instead, the application of task and search knowledge makes the decisions. The knowledge stored in the agent's rules entirely controls the system's behavior.

References

1. A. Newell, *Unified Theories of Cognition*, Harvard Univ. Press, Cambridge, Mass., 1990.
2. J.W. Smith and T.R. Johnson, "A Stratified Approach to Specifying, Designing, and Building Knowledge Systems," *IEEE Expert*, Vol. 8, No. 3, June 1993, pp. 15–25.

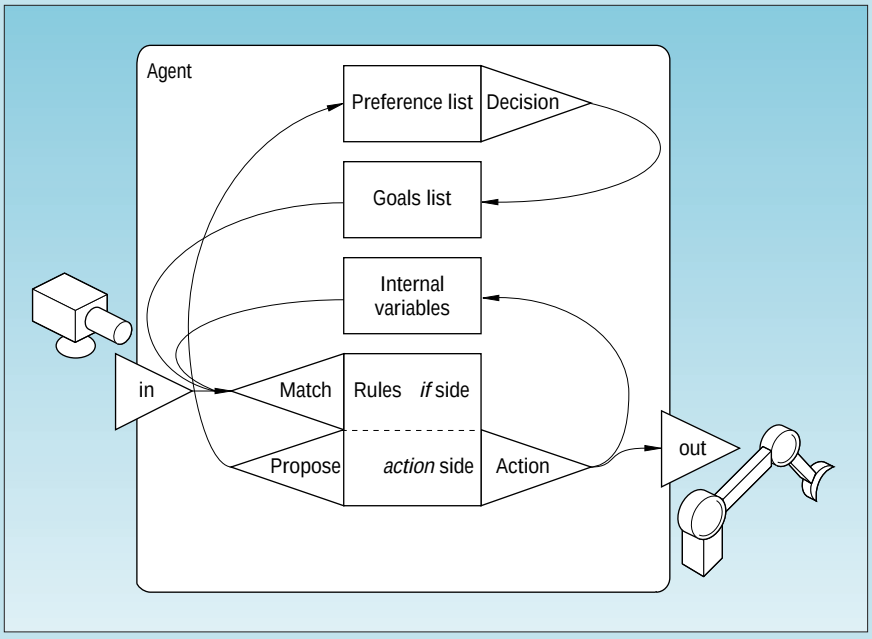

Figure A. An agent's structure.

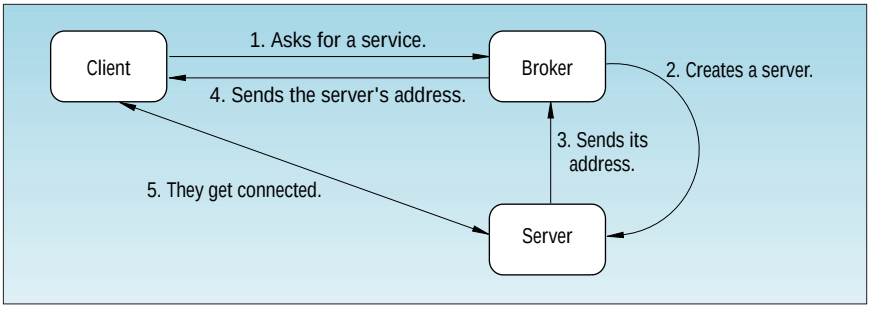

Figure 1. A negotiation between a client and the Broker agent server.

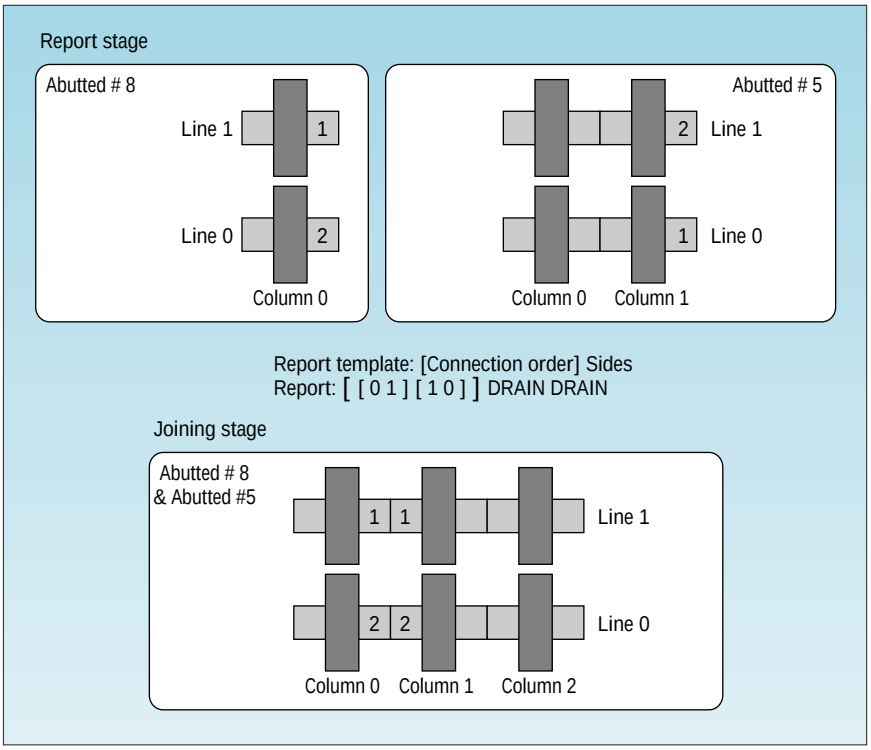

Figure 2. The agent-grouping process.

that are fully scalable; that is, when the number of a system's processors increases, system performance scales up proportionally. For software, scalability basically means that a program should adapt to exploit the available computational resources. This can be achieved through the client-server model, where a server program provides a service that a client program requests.

To organize all of Agents' clients and servers, we adopted a solution based on a *broker*. A broker is a server that acts as an agent in negotiating contracts. In terms of distributed computing, the broker is an intermediary between the client making a request and the server that fulfills the request.

We created our Broker agent server to coordinate the access between our applications and other servers. Written in Lisp (Squeme), Broker interprets Lisp commands sent in by a stream-socket connection with a client. On top of Lisp, Broker implements a subset of KQML (knowledge query and manipulation language).[4]

Each agent appears to KQML as if it manages a knowledge base. That means, communication between agents are with regard to their knowledge bases. Broker retains information about its clients and about servers it manages. This information forms the Broker's knowledge base. Broker identifies each client or server by a symbol, which has an entry in the knowledge base. Broker searches using the identity symbol of a client or server as a key.

When an application wants to use any of the services provided by servers that Broker manages, it starts a negotiation with Broker. Figure 1 illustrates an example. The client asks for a service. Because the server in charge of this service is not running, Broker creates one, obtains its address, and sends it to the client. The client then begins to communicate directly with the server. Broker decouples its clients from the implementation details of the services it manages.

Placer agent server. Through a network, this server receives from its clients a description of the circuits to be generated, using the EDIF language. After reading the design information, Placer positions components by

- forming columns of related components,
- forming groups (joining columns of FETs, field-effect transistors, that share drain or source connections or form pass pairs), and
- using genetic-algorithm techniques to place the groups and route them with Router.

The best placement successfully routed becomes the final circuit. Placer sends this circuit back to the client application. The design comprises a list of EDIF cells consisting of physical-mask layout views and of one main cell. The main cell's symbolic view reports on how to connect the other cells.

Apart from the main cell, Placer has five types of cells:

- *PAD*, for I/O PADs;
- *FET*, for MOS transistors;
- *Bipolar*, for bipolar transistors;
- *General*, for any other kind of cells; and
- *ElectricNode*, for the wiring of the nodes.

Placer distributes component cells over a design containing the PAD cells and any preplaced cells. Placer uses three kinds of agent objects for placement:

- *Cont* controls all the operations;
- *Abutted* builds columns of related transistors and then forms groups; and
- *Eval* uses the genetic algorithm to find a good placement for the groups.

Column formation. A Cont agent coordinates all the actions. It receives the new circuit as a list. It separates the NMOS (*n*-channel MOS) FETs, PMOS (*p*-channel MOS) FETs, bipolar transistors, and general cells in different lists and creates the first Abutted agent.

During this phase, Abutted agents try to form columns of components. Each column contains only one kind of component. When

created, an Abutted agent performs the first of its two basic behaviors: it goes to a Cont agent's lists of available cells and tries to grab as many components as possible to form its column. First, the agent tries to obtain MOS FETs that have their gate interconnected or to form a pass pair (two FETs that have their source and drain interconnected). If all MOS transistors have already been used, the Abutted agent constructs its column with pairs of interconnected bipolar transistors. If it cannot form a pair, it settles for only one transistor. If no transistors remain, the agent fetches a general cell.

After the Abutted agent has formed its column, it tries to reproduce. If cells are available in a Cont agent, it creates a new Abutted agent, which tries to obtain cells. Abutted agents behave like a culture of bacteria—they keep reproducing until there is no more food.

Group formation. After column formation, a Cont agent switches the Abutted agent to its second behavior. The Abutted agent will now try to pair up with other agents joining their cells. To pair up, two agents must share a number of source or drain interconnections. The idea is to join columns of FETs, by either the source or the drain sides. In this way, the Abutted agent can lay out FETs in parallel strips of diffusion—that is, they can be abutted.

In Figure 2, a Cont agent controls group formation, performed in cycles. In each cycle, the agent goes through the list of Abutted agents, exposes an agent, and asks the other agents for reports on how well they would connect to the exposed agent. A report states if a match is possible, how good it is, and details how it should be implemented. At the end of a cycle, a Cont agent joins the two agents that have the best connections. These cycles continue until no agents can be joined or the quality of the possible connections is too poor.

After the cell grouping has finished, a Cont agent contains a list of Abutted agents holding groups of cells. A Cont agent takes these cells from the Abutted agents and puts them in a list. Then, the Abutted agents are killed.

Genetic-algorithm placement. Up to now, we have not determined component grid or position coordinates. We will now place the groups using the genetic algorithm encapsulated in an Eval agent. (Genetic algorithms are a class of computational model that mimic natural evolution to solve problems in a wide variety of domains.[5])

The algorithm follows a simple cycle:

- creation of a population,
- evaluation of each individual,
- selection of the best ones for reproduction, and
- genetic manipulation to create a new population.

The cycle repeats until the algorithm finds a good result. Placer sends the best result (placement) to a Router for wiring and thus restarts the reproduction cycle (for another number of generations) to produce another best placement. The procedure repeats until one of the Router servers produces a fully routed design (or an error condition occurs), which it sends to the Placer's client (see Figure 3).

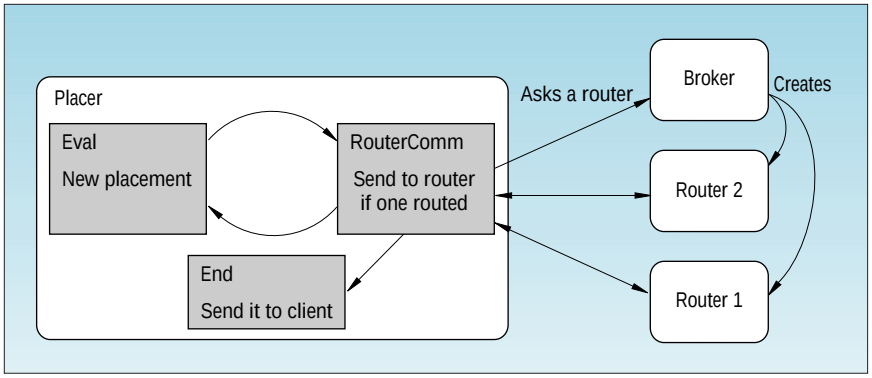

Figure 3. The placement and routing cycle. The RouterComm object undertakes all the communication and resource management of the Router servers.

Router agent server. A Router server performs all the circuit wiring. It receives circuits with all the components already placed. It routes these circuits and returns them to the client. Router basically mimics the way human designers use a CAD system to route circuits: The designer makes all the important decisions about design, such as where the wires go and the quality of the routing. The designer uses a CAD system as a tool to represent and manipulate the design. A major drawback of this methodology lies in the design and maintenance of the cell libraries for every upgrade of the manufacturing process. Our Agents system, however, can generate layouts for a great range of SSI circuits, which take the place of cell libraries, and can be process independent.

In a Router server, *design objects* play the CAD role. They contain the design medium and the facilities to analyze or change it. They use retrieve methods (to search for design-component information) and building methods (to add design elements and conduct simple construction tasks).

RouterExpert and *Connect* agent objects carry out the designer's role. They control the decision making. Using the facilities provided by the design object, they direct the routing.

In combination, RouterExpert and Connect agents perform an augmented *maze-route* algorithm. Maze routing[6] can be performed over a rectangular grid of cells with some cells free and others blocked. Basically, the algorithm finds the shortest path between two predefined cells that does not pass through any blocked cell.

We can easily generalize maze routing—the notion of expanding to neighboring points works with any graph, not only with rectangular arrays. Instead of expanding uniformly in all directions, the algorithm can expand directly to the next *interesting* point,[6] where a change in direction or layer is more likely. This saves time by skipping over less interesting parts of the layout. More important, by eliminating the need to process data at the costly level of pixels, it lets us perform processing directly on the circuit description.

A point is interesting when it aligns with the goal point or obstacle's edge, or crosses the obstacle's side (Figure 4). For an obstacle, we consider the edge and side to be the area that overlaps the obstacle by a security margin.

Making connections. When a RouterExpert agent receives a circuit, it first performs some simple routing. It connects straight-diffusion and polysilicon lines and connects all the unconnected diffusion lines to metal1 (a layout mask used to undertake the routing).

After completing the basic connections, a RouterExpert wires all remaining connections. It lists all unwired nodes, ordered by importance and size. This will help if any rerouting is necessary later. The RouterExpert agent then connects all the list members.

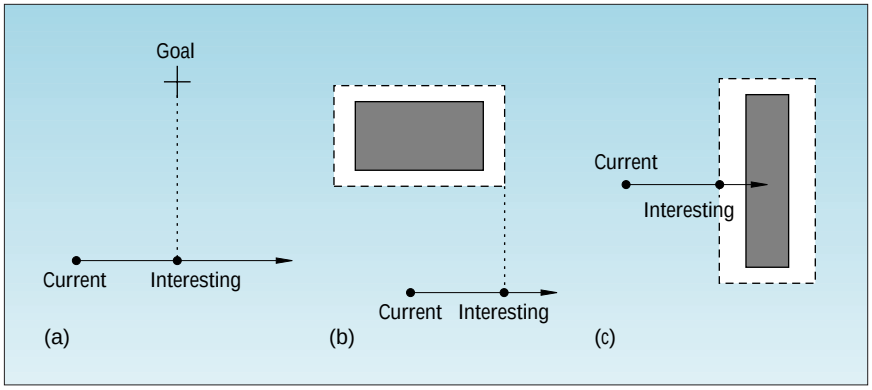

Figure 4. Interesting points: (a) alignment with the goal point; (b) alignment with the obstacle's edge; and (c) crossing an obstacle's side.

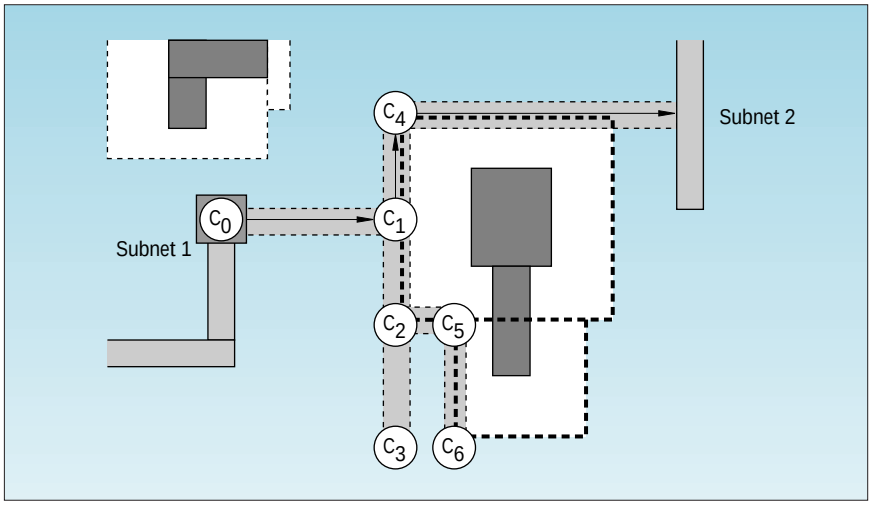

Figure 5. Connect agents probing interesting points: RouterExpert creates a Connect agent C_0 at the wire origin in Subnet 1. C_0 creates C_1 to analyze the next interesting points (crossing an obstacle's side). C_1 creates C_2, C_3, and C_4 to analyze further points. The process continues until C_4 finds a suitable connection to Subnet 2.

Connect agents are at the core of the routing process, because they carry out the actual subnet routing. A Connect agent's goal is to reach a point in the destination subnet (see Figure 5). Upon creation, a Connect agent receives data from its parent agent, which includes where the Connect agent is in the design, where it should go, and the wire segment it already holds. The agent then remains dormant until the RouterExpert agent activates it.

When a Connect agent begins to run, it first analyzes other nearby points of interest (beginning at the wire origin) and wire connections, looking for possible paths to its target net and for obstructions. With this information, it plans which of its operators to apply to modify the layout. These operators can add new sections to the agent's wire by extending the current wire for a specified distance, by going around an obstacle, by changing the wiring layer, or by connecting the wire to the target net. The successful operators add a new section and, if not yet connected to the target net, create a new Connect agent to work on the end of this new section.

In addition to any action, if there is an obstruction, an operator can create a new Connect agent that asks for the obstructions to be unwired and send it to a RouterExpert agent. If this new Connect agent ever runs, it will unwire the obstruction and try a connection. The RouterExpert agent will first run all Connect agents that do not require unwirings. Only if a suitable wire is not found will the other Connect agents be tried.

If a Connect agent finds a connection with the target subnet, the agent extends its wire to it and proposes it as the best wire to a RouterExpert agent. It can create (reproduce) other agents to analyze other nearby points of interest. These operations continue until two subnets are finally wired together by a reasonably short wire.

The rules in the Connect-agent knowledge base try to balance between the number of cases they consider with the likelihood of any of the cases leading to a perfect wire. In turn, RouterExpert agents must control the population of Connect agents so that the program finds a good solution in a reasonable amount of time. If a Connect agent finds a new interesting point, it reproduces. If it completes a wire, the Connect agent sends the wire to a RouterExpert agent. If it has exhausted all its options, it dies. The RouterExpert agent then takes care of its "farm," killing Connect agents with costly routing and giving more resources to agents with prospective wires. The interaction of these two types of agents creates the final routing. The Router server subsequently sends this routing back to the Router server's client.

Database. This server stores all the process-rule information for the Placer and Router servers. We did this to achieve process-independent placement and routing. Whenever a server needs information about the process, it queries Database. Database stores information in hash tables and accesses it using a very simple language. It reads data for each process from a process-description file.

A flexible and scalable solution

Using software agents gives the user more flexibility in the form of a richer set of layout options. Our Agents system can generate layouts for Bicmos and CMOS circuits and can mix small analog cells with a digital design. Figure 6 shows two different circuits generated by the system. To show the system's process independence, we generated them through two very different fabrication processes: Orbit's Bicmos 2-mm double-metal double-poly process and ES2's CMOS dual-metal layer 1.5-mm process.

Figure 6a shows a Bicmos two input nand gate. Figure 6b shows a CMOS D-latch circuit. They exemplify that the system can tackle unusual layouts, such as those with central power lines. Both layouts are of good quality and similar to ones developed manually—the nand gate is approximately 8% bigger than a manual layout, and the D latch is approximately 15% bigger.

We obtained performance flexibility using scalability, the quality of a program to adapt to the resources of the hardware on which it runs. To demonstrate the system's scalabil-

ity, we modified the program to generate ten layouts of these two circuits (normally, it stops when it generates the first layout) in computer configurations ranging from a single PC to eight workstations. All machines ran under Unix, and the workstations were connected by an Ethernet network with a 10 Mbps bandwidth.

Figure 7 shows the speed increases in relation to the PC configuration for the average time interval between the subsequent layout generations.

Even though the absolute values in Figure 7 depend on the network workload (at the time of the benchmark), to demonstrate scalability, we need to show the relative speed gains from one configuration to the other. So, we used the PC time values as a reference. Speed increases as the number of workstations increase, but scalability really shows in the big performance increase difference between one and three workstations.

OUR AGENTS SYSTEM SHOWS that layout generators based on software agents can exploit new trends in mainstream computer hardware, such as distributed processing. It also demonstrates that a distributed system based on cooperative software agents can generate flexible, rich layouts. Furthermore, because of our system's distributed basis, the use of software agents increases runtime speed, depending on the number of available processors. We are currently developing a framework for three-tier client-server VLSI design based on Corba-compliant technology as well as translating the Agents system to Java. Because Java can communicate with Corba objects, Agents 2 will be one of the tools available for this framework.

Acknowledgments

The authors gratefully acknowledge the CNPq—the National Council for Research, an agency of the Brazilian Federal Government—for financially supporting this work.

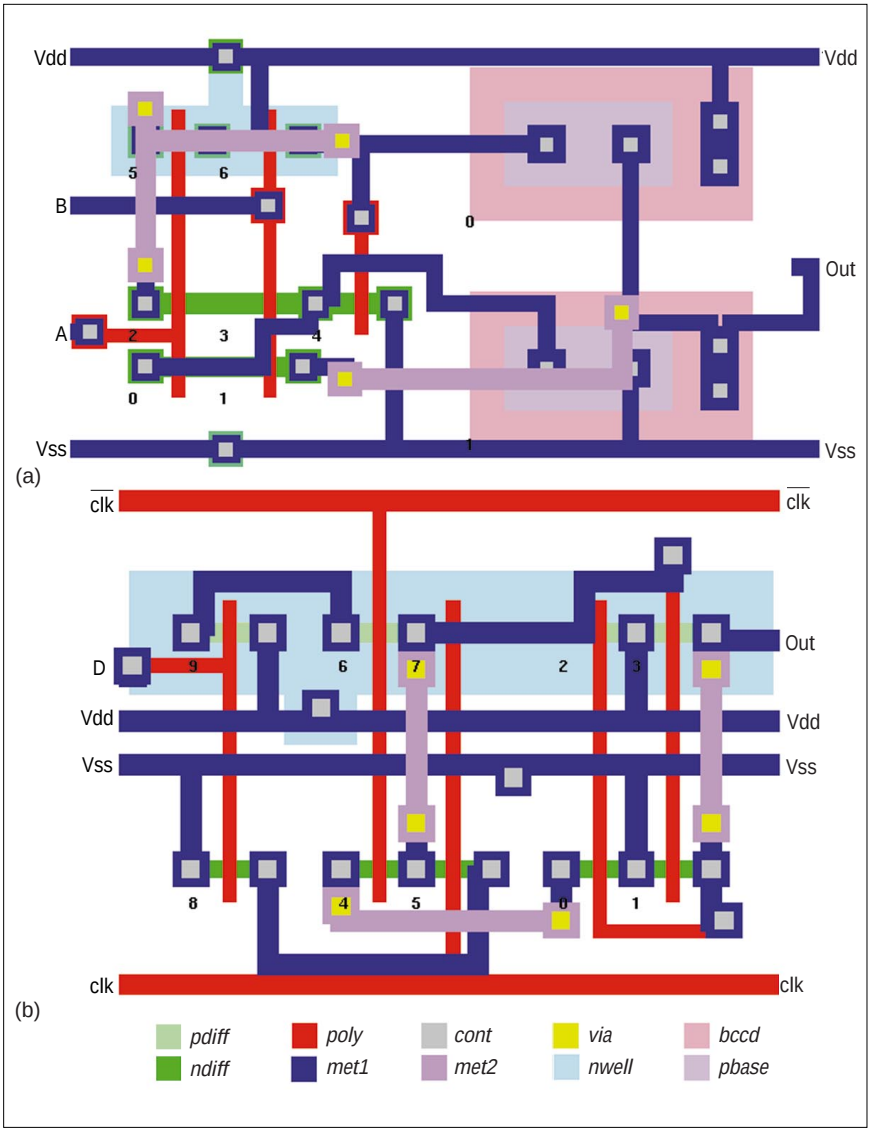

Figure 6. Two circuits generated by Agents: (a) a Bicmos two-input nand gate; (b) CMOS D-latch circuit.[7]

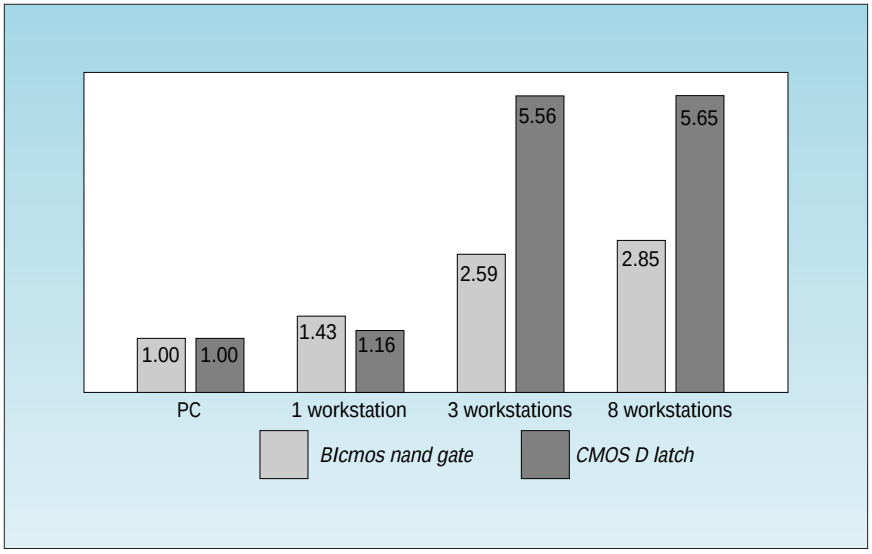

Figure 7. Relative speed increases for the average time interval between layout generations.

References

1. M.R. Genesereth and S.P. Ketchpel, "Software Agents," *Comm. ACM*, Vol. 37, No. 7, July 1994, pp. 48–53, 147.
2. D.A. Moreira and L.T. Walczowski, "Automated Placement for a Leaf Cell Generator," *Proc. IEEE Int'l Symp. Circuits and Systems*, IEEE Press, Piscataway, N.J., 1994, pp. 117–120.
3. J.W. Smith and T.R. Johnson, "A Stratified Approach to Specifying, Designing, and Building Knowledge Systems," *IEEE Expert*, Vol. 8, No. 3, June 1993, pp. 15–25.
4. T. Finin et al., "KQML as an Agent Communication Language," *Proc. Third Int'l Conf. Information and Knowledge Management* (CIKM '94), ACM Press, New York, 1994.
5. J.L. Ribeiro Filho, P.C. Treleaven, and C. Alippi, "Genetic-Algorithm Programming Environments," *Computer*, Vol. 27, No. 6, June 1994, pp. 28–43.
6. M.H. Arnold and W.S. Scott, "An Interactive Maze Router with Hints," *Proc. 25th ACM/IEEE Design Automation Conf.*, IEEE Press, 1988, pp. 672–676.
7. D.A. Pucknell and K. Eshraghian, *Basic VLSI Design*, 3rd ed., Prentice Hall, London, 1994, color plate 8(a).

Dilvan de Abreu Moreira is a lecturer in the Department of Computer Science and Statistics at the University of São Paulo, Brazil. His research interests include distributed client-server systems, object-oriented technologies, and intelligent agents. He received his BS from the Federal University of Bahia, Brazil; his MS from the State University of Campinas; and his PhD from the University of Kent—all in electrical engineering. He is a member of the IEEE, the IEEE Computer Society, and the ACM. Contact him at SCE-ICMSC-USP, Caixa Postal 668, Av. Dr. Carlos Botelho, 1465, São Carlos-SP 13560-970, Brazil; d.moreira@ieee.org.

Les T. Walczowski is a senior lecturer in the Electronic Engineering Department at the University of Kent, where he has run the ECAD Laboratory from its inception in 1985. His research interests include CAD tools for VLSI. He is one of the main developers of the ChipWise VLSI design system, used widely in the UK. He graduated in physics at the Imperial College of Science and Technology, London, and obtained his PhD in electronics from the University of Kent. He is a member of the ACM, the IEEE, the IEEE Computer Society, and the IEEE Circuit and Systems Society. Contact him at the Univ. of Kent at Canterbury, Electronic Engineering Laboratory, Canterbury, Kent CT2 7NT, UK; l.t.walczowski@ukc.ac.uk.