VLSI DESIGN
1996, Vol. 5, No. 1, pp. 101–110
Reprints available directly from the publisher
Photocopying permitted by license only

© 1996 OPA (Overseas Publishers Association) Amsterdam B.V.
Published in The Netherlands under license by
Gordon and Breach Science Publishers SA
Printed in Malaysia

High-Level Graphical Abstraction in Digital Design

MURRAY W. PEARSON[a,*], PAUL J. LYONS[b,†] and MARK D. APPERLEY[a,‡]

[a]*Department of Computer Science, University of Waikato, Hamilton, New Zealand;* [b]*Department of Computer Science, Massey University Palmerston North, New Zealand*

(Received 2 March 1993; In final form 24 November 1994)

We base our approach to the design of complex logic ICs on four premises:

- Design of a chip's abstract architecture—its major components, their tasks, and their intercommunication—should precede definition of its functionality.
- Graphics is ideal for representing abstract architectures; text is better for functionality.
- The designer should not have to translate graphical information into text.
- Graphical and textual design capture can be integrated with synthesis.

Keywords: Hardware description languages, data flow diagrams, synthesis, visual languages

1. IC DESIGN IS A COSTLY BUSINESS

Production costs of high-volume ICs with millions of identical elements [1], like memories, have decreased phenomenally over the last 35 years, but the cost of designing more complex devices, like processors, makes them uneconomical to produce in low volumes. Mead and Conway's Structured Design Methodology [11] was the first widely accepted tool for managing this complexity, and it was followed by CAD tools, which increase design abstraction levels, and then HDLs (Hardware Description Languages), which facilitate communication with CAD tools. The popular HDLS VHDL [16] and Verilog [18] are typical in that they represent designs textually, as sets of connected components; each component can be described by its behaviour or by a set of lower-level components. However, these, and many other textual HDLs represent the more abstract parts of a design poorly, and designers often prefer to work up their ideas using informal graphical notations, which they then regenerate in the less intuitive textual form (cf.[6], [14]). Subsequent textual alterations are rarely back-propagated to the diagrams, and their mnemonic value is lost.

These designers—and text-book authors often use the same approach—are intuitively working at the level of the device's *abstract architecture*, that is, its major components, their hierarchy of responsibility, and the communication paths between them, but not

*Phone +64 7 838 4409 Email:mpearson@cs.waikato.ac.nz Fax +64 7 838 4155
†Phone: +64 6 350 4182 Email P.Lyons@massey.ac.nz Fax: +64 6 350 5611
‡Phone: +64 7 838 4528 Email: m.apperley@cs.waikato.ac.nz Fax: +64 7 838 4155

their detailed functionality. Abstract architectures are well-suited to graphical representation, whereas functionality is more suited to textual representation.

The PICSIL IC design environment is derived from this concept. Figure 1 shows its structure. A bespoke graphic editor captures the abstract architecture, which is represented as modified DFDs (Data Flow Diagrams). From the DFDs, this editor generates a textual framework within which the user places text representing the components' functionality. It also synthesises the data channels, via which the functional units communicate, and associated synchronisation hardware. The PICSIL compiler then amalgamates the textual and diagrammatic parts of the design into HardwareC and SLIF [9] representations. In a third phase, PICSIL's synthesis manager coordinates the automatic translation of these HardwareC and SLIF descriptions by the Olympus [2] and OctTools [12] suites, and some special-purpose synthesis programs, to produce a core layout and amalgamates this with a set of pin assignments which the user has created, thereby producing a conventional CIF layout file for fabrication.

2. PICSIL REPRESENTS BUILDING BLOCKS AS DFDS

Data Flow Diagrams [3] are the basis of a suitable notation for representing organisation within hardware designs. Figure 2 shows a conventional DFD, a

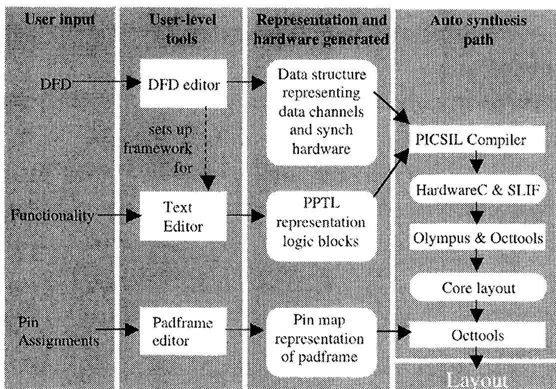


FIGURE 1 Design Flow using PICSIL

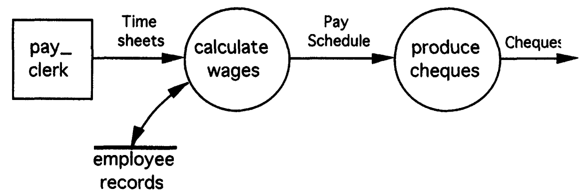


FIGURE 2 Conventional DFD notation

directed graph with arrows showing data flows between an arbitrary number of modules; *external entities* (squares), *processes* (circles), and *data stores* (heavy horizontal lines). *Primitive* processes have a text-based functional definition. *Non-primitive* processes are defined by lower-level DFDs, so that the abstract architecture tree may be defined before any functional definition is required.

To make DFDs suitable for describing hardware systems, they were "fortified" in three areas. First a single consistent diagrammatic syntax for PICSIL DFDs was chosen. The syntax of conventional DFDs is vague, as they were designed for hand drawing, and various authors have adapted them differently. Returning to DeMarco's original notation [3] was not desirable, as it lacks some important features, (e.g., a way to control process activation). Secondly, the informal "structured English" used for a specifying a DFD's *data dictionary* (flow formats) and *process transform* (functionality) was replaced by formal diagrammatic and textual notations. Thirdly, some modest additions to the common software-oriented vocabulary were introduced to make DFDs more suitable for hardware description. The most important components (as shown in Figure 3) are discussed in this paper. Pearson, Lyons, and Apperley [15] discuss others, such as routers, which steer data between groups of processes, and elements, which allow components in a diagram to be replicated.

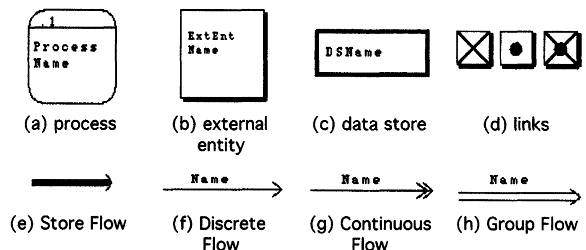


FIGURE 3 Components that appear in PICSIL DFDs

3. PICSIL MAPS WELL ONTO DESIGNERS' MENTAL MODEL OF HARDWARE

Design of large systems generally involves dividing complex tasks into simpler ones, and then specifying their functionality. The syntax used for this by many HDLs is based on conventional programming languages, which represent information in one dimension. Consequently, information equivalent to subroutine scope information and subroutine functionality specification are intertwined. In DFDs these specifications are separate, allowing the designer to focus creative energy on one level of abstraction at a time. Specifically, the DFD notation has these advantages:

- Processes highlight the organisation of responsibilities within a design.
- Functional specifications don't interfere with organisational specifications.
- 2-D DFDs have a structure which matches the 2-D nature of hardware.
- High-visibility, directional, communication paths emphasise patterns of responsibility.
- Organisational and functional specifications have separate graphic and textual vocabularies.

4. DIVISIONS OF A PICSIL DESIGN: DFD, CONTROLLER, TEXT

As described above, the graphical DFD notation is used for defining a system's abstract architecture. Another graphical notation is used for defining controllers and a third, textual, notation is used for defining the data processing functions of a process. Before dealing with their syntax and semantics, let us consider why this trio of notations is appropriate.

Controllers govern activation of a DFD's processes, so that they can respond to particular events. A process could use conditional tests to control its functionality, but this would impose a management duty on low level functional code. Controllers give PICSIL a management level for specifying control; the notation is based on the Finite State Machines used in the extended DFD methodologies developed by Hatley and Pirbhai [8], and Ward[20]. This avoids tainting the pure functional code with management duties (which would be the result of making sections of the code conditional on external events). More detail on controllers can be found in [14] and [15].

By contrast with controllers, which perform a management function, the data-processing specifications in PICSIL have syntax and semantics typical of current high-level programming languages (that is, comparatively low-level).

In a PICSIL DFD tree, data-processing functionality is associated with the leaf, or primitive, nodes, and is written in PICSIL Process Transform Language, PPTL [14] which is based on HardwareC [9]. In PPTL a designer designs hardware using procedural programming language constructs. Variables generate registers; arithmetic and logical expressions generate arithmetic and logical processing units to operate on the values in those registers; labels on statement groups make the corresponding hardware function sequentially or in parallel.

PPTL possesses constructs for implementing DFD functionality, and data transmission statements which implement the data-driven process-synchronising mechanism of DFDs. Thus the two notations correspond very closely, and it is possible to integrate the high-level organisational structure of the DFDs with PPTL automatically.

In order to illustrate the syntax and semantics of the PICSIL language, the design of a simple serial interface will be considered. The device accepts characters from a 6800 parallel bus and converts them into a continuous data stream for transmission. Simultaneously, it can convert serial input data into parallel data characters to be written to the 6800 bus. A status register can be read to determine whether a character has been read by the serial port and is ready for transfer to the CPU. The status register can also be read to determine whether the device is ready to receive another character from the 6800 bus for transmission over the serial port. The number of data bits and stop bits is determined by a control word written to the device over the 6800 bus. Figures 4 to 7 show a complete PICSIL definition for the serial interface (apart from the two primitive processes *SendSerialChar* and *ReceiveSerialChar*, which have been omit-

ted for space reasons). Figure 4 shows top level representation as it would appear on the designers screen. Each of the components of this diagram is described below.

Processes (See Figure 3(a))

Processes transform incoming flows into outgoing flows. They contain a name and a unique address within the diagram. Each process in a diagram is defined in more detail. Primitive processes have a level of abstraction which is low enough to be defined textually in the data dictionary. Non-primitive processes are more abstract and are refined in a child DFD.

Non Primitive Process Decomposition

Figure 5 shows the refinement of the non-primitive process *InterfaceSerialPort* into three sub-processes which in turn can be decomposed. Decomposition of a process adds no new functionality to the system. It only defines it in more detail.

Links (small shadowed boxes in Figure 3(d) Links are purely notational symbols which are added to the child diagrams automatically, so that all flows attached to a parent process also appear in the child diagram. Import links show data flows arriving at the diagram, and contain a disc, representing an approaching arrowhead; export links show data flows exiting from the diagram, and contain a cross, representing the tail of a departing arrow. Import/export links contain a cross overlaid with a disc.

Primitive Process Decomposition

The actual information-processing behaviour of a system is specified in the textual data dictionary entries of its primitive processes. All primitive processes in a system execute concurrently and restart themselves on completion unless prevented by a controller. Figure 6 shows the PPTL data dictionary entry for the primitive process *Interface6800Bus*. It can be seen that it resembles a C function definition: the name of the process is followed by the declaration of its local variables, then statements for input processing and output. This syntax of PPTL is akin to that of HardwareC [9], with a number of extensions to support the different types of inter-component communication, and improved representations for timing constraints and concurrency.

When a designer refines a primitive process for the first time, the PICSIL editor automatically generates a textual PPTL process skeleton for it, and then the designer fleshes this out with PPTL statements defining its functionality.

External Entities (See Figure 3(b))

External entities represent connections to external I/O ports, and are synthesised as connections to bonding pads on the periphery of the chip.

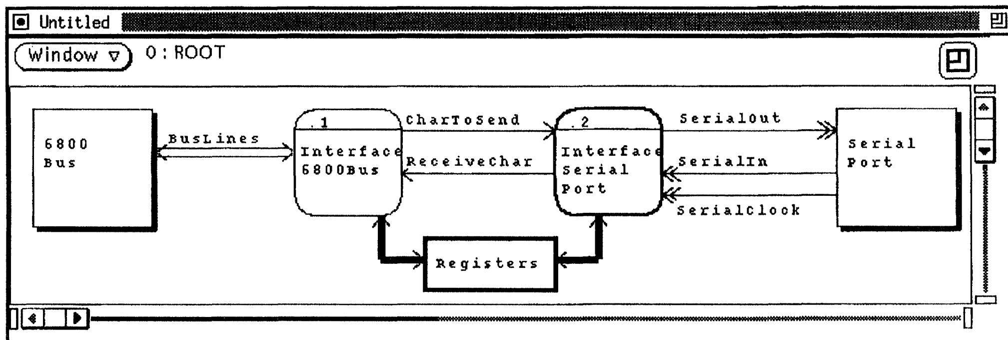


FIGURE 4 Top level DFD for 6800 serial interface

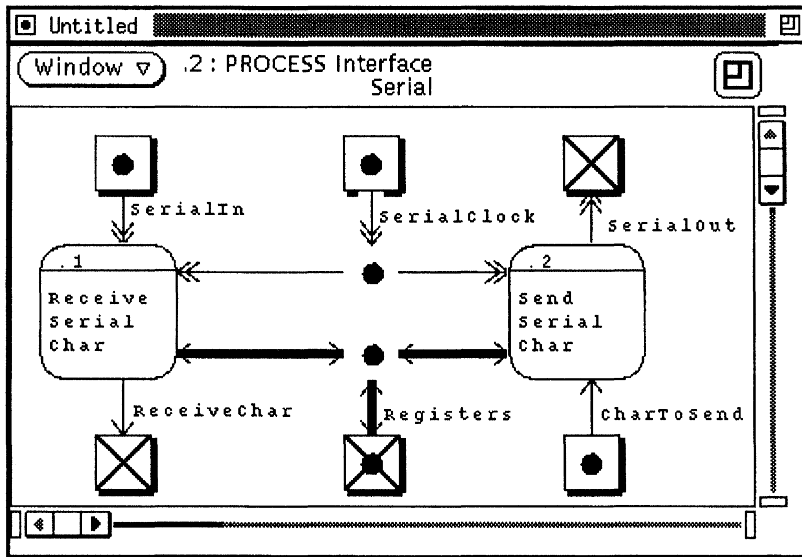


FIGURE 5 Refinement of non-primitive process InterfaceSerialPorts for the 6800 serial interface

```
PROCESS .1 Interface6800Bus
seqbegin
boolean Reg, Direction;
boolean{0:7} status, value;
  wait(!BusLines\CS);
  read(BusLines\RegSelect, Reg);
  read(BusLines\ReadWrite,Direction);
  stread(Registers[STATUS],status);
  if (Direction == WRITE) seqbegin
    read(BusLines\data, value);
    if (Reg == DATAREG) begin
      send(CharToSend,value);
      status{3} = 1;
      stwrite(Registers[STATUS],status);
    end
    else stwrite(Registers[CNTRL],value);
  seqend
else seqbegin
    if (Reg == DATAREG)
      if (msgwait(ReceiveChar)) begin
        receive(ReceiveChar,value);
        status{0} = 0;
        stwrite(Registers[STATUS],status);
      end
      else value = 0x0;
    else stread(Registers[STATUS],value);
    write(BusLines\data,value);
    wait(BusLines\enable);
    wait(!BusLines\enable);
    free BusLines\data;
  seqend
  wait(BusLines\CS);
seqend
```

FIGURE 6 Refinement of primative process Interface6800Bus for the 6800 serial interface

Data Stores (See Figure 3(c))

A data store is a configurable random-access memory. Its contents may be accessed using a write or non-destructive read by a number of processes. Data

```
#define WRITE 0
#define READ 1
#define DATAREG 0
#define STATUS 0
#define CNTRL 1
STORE boolean {0:7}[2] Registers;
FLOW Continuous boolean SerialClock;
FLOW Continuous boolean SerialIn;
FLOW Continuous boolean SerialOut;
FLOW Continuous boolean BusLines\RegSelect;
FLOW Continuous boolean BusLines\ReadWrite;
FLOW Bidirectional Continuous
          boolean{0:8} BusLines\data;
FLOW Continuous boolean BusLines\enable;
FLOW Continuous boolean BusLines\CS;
FLOW Bidirectional Group {
     BusLines\RegSelect;
     BusLines\ReadWrite;
     BusLines\data;
     BusLines\enable;
     BusLines\CS;
} BusLines;
FLOW Discrete boolean{0:8} ReceiveChar;
FLOW Discrete boolean{0:8} CharToSend;
```

FIGURE 7 Defintion of constants, stores and flows for the 6800 serial interface

stores enforce data locking to prevent consistency problems occurring when multiple processes attempt to access a data store at the same time.

Data Flows

A data flow is a communications channel connecting other components. The flow arcs show the names, types and directions of data that flow between the various components of a design. Six types of flow exist in the PICSIL notation; Discrete, Continuous, Store, Group, Event and Continuous Event.

Discrete Flows (See Figure 3(f))

A discrete flow conveys data between processes. The sending and receiving processes each wait till the other is ready to receive or send data before sending or receiving it respectively. The logic for implementing this *data-driven* protocol is generated automatically.

Send and *receive* statements are provided in PPTL for specifying transfer of data over discrete flows. For example, when the process *Interface6800Bus* in Figure 5 produces data for the discrete flow *CharToSend*, it executes the *send* statement *send(CharToSend, data);*. The process *SendSerialChar* uses the *receive-(CharToSend, data);* statement to receive new characters off the flow.

The type of every discrete flow is defined in the data dictionary. E.g., the discrete flow *CharToSend* is declared as "*flow discrete boolean {0:7} CharToSend;*".

Continuous Flows (See Figure 3(g))

Continuous flows are used for communication with off-chip devices which do not use the data-driven synchronising protocol. Values are read from, or written to, continuous flows using PPTL's *read* and *write* statements. Each output remains on the flow until a new value is output. A read statement (e.g. *read (SerialIn, NewBit);*) samples the current value on a continuous flow.

A continuous flow with arrows at both ends specifies half duplex data transfer. It is left to the designer to ensure that only one object outputs to a bidirectional continuous flow at a time. All continuous flows must be defined in the data dictionary.

Store Flows (See Figure 3(e))

Store flows provide access to data stores. They comprise a data store address, locking signals for preventing deadlocks, and data.

The data dictionary language's stwrite and stread statements control store flows. For example *stwrite (Registers [position], Char)* writes the value *Char* into the data store *Registers* at the address given by *position*. *Stread* allows the contents of a particular location of the name store to be read into the named variable.

Store flows do not require an entry in the data dictionary as their type (data, address, and locking information fields) can be derived automatically during the synthesis process. They are unnamed. The arrow shows the direction of data flow appropriate to the type of access: read-only, write-only or read/write.

Group Flows (See Figure 3(h))

Group flows are a notational convenience; they allow several independent data flows to be condensed into a single symbol. Use of a group flow symbol does not imply that the component flows are mutually synchronised.

5. THE PICSIL EDITOR

The PICSIL editor has been designed for simplicity and transparency. It uses a direct manipulation interface to allow diagrams to be input and edited graphically. The editor supports multiple windows, and a new window is opened for each object that is refined.

At present, only the graphical components of a PICSIL design are parsed interactively, so complete consistency can only be maintained between graphi-

cal objects. For example, if a data flow is changed, all other diagrams which use it are automatically updated. At present no consistency is maintained between graphical and textual views. However, researchers in visual programming languages [7], [10] discuss techniques that can be used to maintain the consistency between graphical and textual views.

Ten designs have been input using the editor in order to test it. This experience has demonstrated that the language is sufficiently powerful to represent a wide variety of device types succinctly and elegantly. However, the level of interaction in the current system could still be improved. Features such as block move operations (allowing any number of objects to be selected and moved at the same time), version control and the ability to reuse components from other designs would allow the design space to be explored more effectively.

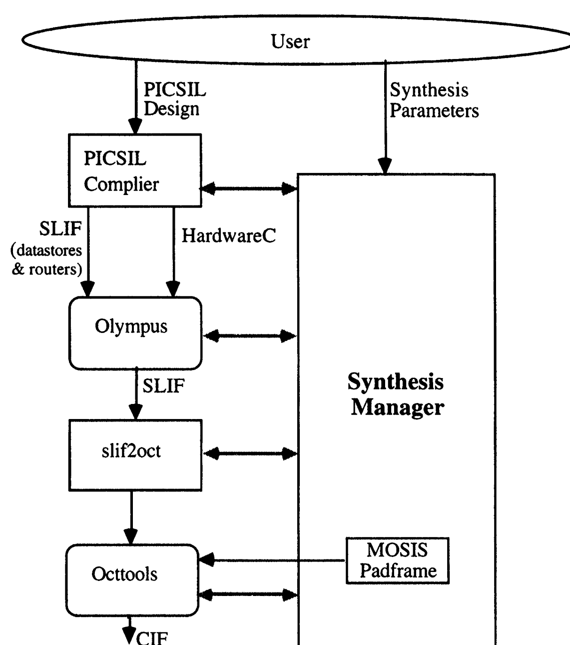


FIGURE 8 PICSIL Synthesis Path

6. THE PICSIL SYNTHESIS MANAGER

The various editors which are used to capture a PICSIL design are only the first phase in a chain of tools for translating a design through successively lower levels of abstraction. Some of these tools are public-domain systems and some were created by the authors of this paper to augment, and provide bridges between, the public-domain systems.

The combination of Olympus (for high level synthesis) and Octtools (for logic and layout synthesis) was identified as a satisfactory framework for the PICSIL synthesis system. To provide a complete synthesis path (see Figure 8) two additional tools were developed; the PICSIL compiler and the slif2oct translator (SLIF—Sequential Logic Intermediate Form—is an intermediate design representation generated by Olympus).

The first phase of synthesis is performed by the PICSIL compiler, which translates PICSIL descriptions (captured by the PICSIL editor) into Olympus' input language, HardwareC. While HardwareC has constructs corresponding to most PICSIL components, it lacks any representation of data stores and routers, so instead, they are synthesised by the PICSIL compiler into SLIF and stored until they can be incorporated into the output generated by Olympus.

The second phase of the synthesis is performed mainly by the Olympus suite of programs, under the control of the Synthesis Manager, and involves generation of a SLIF (netlist) representation of the whole design. In the third phase of the synthesis, the Synthesis Manager uses a bridge program (slif2oct) to translate the SLIF design into an appropriate input form for the OctTools suite of programs which, still under the control of the Synthesis Manager, perform logic and physical synthesis. The resulting layout and a proprietary pad frame are integrated and used to produce a CIF file which is suitable for input to the fabrication process.

The synthesis path involves a large number of programs which have to be invoked in the correct sequence, provided with the correct data, and instructed to perform the correct actions. The PICSIL Synthesis Manager reduces the overhead in learning to use these by abstracting this complexity into a small set of high-level parameters (see Figure 9), which the designer specifies, and from which it generates and issues the necessary commands to drive the synthesis

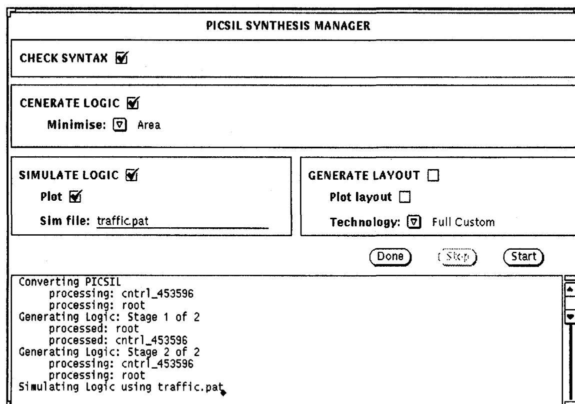


FIGURE 9 Synthesis Dialogue pop-up window

process. Within the synthesis manager mechanisms are also provided for reporting errors, interfacing to the Olympus simulator and mapping a designs I/O's to specified pads around the periphery of the chip.

7. RELATED WORK

PICSIL is not the first system to represent hardware graphically. Gate-level schematic capture tools have been used widely, and a few tools have been developed which use graphical representations at higher levels of abstraction. State charts [5], a visual language for reactive systems, have been used for defining behaviour, specifically control in monolithic designs, but they are inappropriate for specifying the responsibilities of a system's major components in the early, high-level, stages of a design.

AVE [4] and vVHDL [6] represent the structural aspects of VHDL graphically. vVHDL also represents behavioural constructs visually. However, the languages only transliterate VHDL. Neither exploits graphical notations fully. Features of the PICSIL notation which are not present in other visual HDLs include: different data flow types to represent the different types of data present in a system, addressable data stores accessible from a number of other devices, a general data routing device, a generator construct which allows easy representation of repeated subcircuits, and a properly formalised representation of control definition [15].

HardwareC [9] is a textual language with a similar interconnection paradigm to the one used in VHDL and Verilog. However it also provides channels which use a predefined synchronisation protocol. This frees the designer to focus on the modules' functionality rather than their interaction. The language was oriented towards synthesis from the outset, whereas VHDL and Verilog were originally simulation-based.

8. PRACTICAL EXPERIENCE WITH PICSIL

To date, 10 devices have been represented in the PICSIL notation (both graphical and textual aspects) from a variety of application areas including data communications, state machine controllers, interface hardware and signal processing. Three of these have been automatically synthesised and one (a traffic light controller) has been successfully fabricated using Orbit Semiconductors Foresight program [13]. They were selected to test the algorithms used to map PICSIL components to hardware, and the synthesis manager's ability to automate the synthesis process fully. During the synthesis of these designs, a number of test points (including visual checks and simulations) were inserted into the synthesis path, to demonstrate that all the tools in the synthesis path have functioned correctly.

The designs were represented in other Hardware Description Languages besides PICSIL to allow a comparison of the relative ease of representation which they provided. In all cases the PICSIL representation was considered by the designers to be a clearer way of representing the abstract architecture of the device. Pages of close-packed operational specification were replaced by a two or three diagrams clearly showing communication between components. At some level, the operational specification has to be introduced in any system, but in PICSIL, each component's functionality is more isolated in the design; PICSIL's intercommunicating processes

bear a close resemblance to the message-passing objects which are currently enjoying popularity as a way of improving conventional programming language use.

To support design using the PICSIL HDL, an editor to capture a design and synthesis manager to automate the synthesis of the design have been developed. While, at the current stage of development, these tools lack superficial gloss, they have demonstrated that it is possible to capture PICSIL designs in a more natural vocabulary that existing Hardware Description Languages and that these designs can be synthesised automatically leaving the designer free to concentrate on the higher-level aspects of the design which require greater creativity. When evaluated against Sequin's guidelines [17] which outline the key issues essential to manage complexity in VLSI design, it has been found that PICSIL does provide the necessary features to effectively manage design complexity.

Notwithstanding the advantages which we claim for the approach, the current PICSIL system is a prototype and not a production tool. The layouts which it produces, particularly those including controllers and data stores, are comparatively large and slow. This project has achieved its aim: demonstrating the practicability of a new generation of HDLs which take full advantage of today's GUI environment.

References

[1] Burger, R.M. and Holton, W.C., "Reshaping the Microchip", *Byte*, vol. 17, no. 2, 137–148, Feb 1992.

[2] De Micheli, G., Ku, D., Mailhot, F., and Truong, T., "The Olympus Synthesis System," *IEEE Design & Test of Computers Magazine*, vol. 7, no. 5, 37–53, Oct 1990.

[3] DeMarco, T., *Structured Analysis and System Specification.* Prentice-Hall, 1978.

[4] Dettmer, T., Rasche, A., and Sohlenkamp, M., "Concepts for Graphical Editing of VHDL with AVE," in *Proceedings of EURO-VHDL 91, Swedish Institute of Microelectronics*, 1991, pp. 184–187.

[5] Drusinsky, D. and Harel, D., "Using Statecharts for Hardware Description and Synthesis," *IEEE Transactions on Computer Aided Design*, vol. 8, no. 7, 798–807, July 1989.

[6] Golin, E.J. and Feng, A.C., "A Visual Hardware Description Language," in *Proceedings of the CHDL-93 Conference on Hardware Description Languages and Their Applications*, April 1993.

[7] Grundy, J.C. and Hosking, J.G., "Constructing Multi-View Editing Environments Using MViews," in *Proc. IEEE Symposium on Visual Languages*, 1993.

[8] Hatley, D.J. and Pirbhai, I.A., *Strategies for Real-Time System Specification.* Dorset House , 1987.

[9] Ku, D. and De Micheli, G., "HardwareC—A Language for Hardware Design Version 2.0," Tech. Rep., CSL-TR-90-419, 1990.

[10] Lyons, P., Simmons, G., and Apperley, M., "HyperPascal: A Visual Language to Model Idea Space," *Proceedings of the 13th New Zealand Computer Society Conference*, vol. 13, 492–508, 1993.

[11] Mead, M. and Conway, L., *Introduction to VLSI Systems.* Addison Wesley, 1980.

[12] Octtools, *Tool User Guides and Tutorials, Octtools version 5.0*, Electronics Research Laboratory, University of California, Berkeley, May, 1991.

[13] Orbit, *Foresight Users Manual*, Orbit Semiconductor, 1230 Bordeaux Dr., Sunnyvale, California 94089, 1.4, Jul, 1991.

[14] Pearson M. W. PICSIL: "Design and Synthesis of Digital ICs From Data Flow Diagrams", PHD Thesis, Department of Computer Science, Massey University. New Zealand, 1992.

[15] Pearson ,M.W., Lyons, P.J., and Apperley, M.D. "PICSIL: Integrating Graphic System Design and Automatic Synthesis", Working Paper 95/3, Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1995.

[16] Perry, D.L., *VHDL.* McGraw Hill, 1991.

[17] Sequin, C.H., "Managing VLSI Complexity : An Outlook," *Proceedings of the IEEE*, vol. 71, no. 1, 149–435, January 1983.

[18] Thomas, D.E. and Moorby, P., *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[19] Ward, P.T., "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering*, vol. SE12, no. 2, 198–210, February 1986.

Authors' Biographies

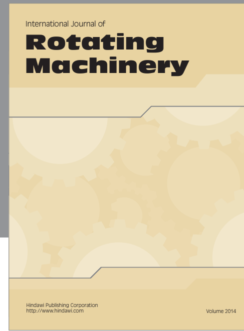
Murray Pearson is a lecturer in the Department of Computer Science at the University of Waikato. He studied Computer Science at Massey university and received the degrees BSc (hons) in 1987 and PhD in 1992. His current research interests include Hardware Description Languages, Computer Architecture and Computer Networks.

Paul Lyons is a Senior Lecturer in Computer Science at Massey University. His research publications have concentrated mainly on Network Design, Silicon Compilation and Visual Programming Languages. He supervised Murray Pearson's Ph.D. research into PICSIL, a project in which the last two interests supported each other admirably.

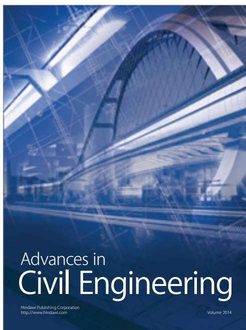
Mark Apperley is Professor and Chairperson of the Department of Computer Science at the University of

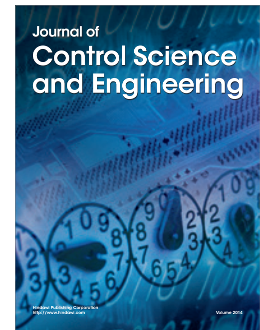
Waikato. Mark studied Electrical Engineering at the University of Auckland, and then went on to complete a PhD on digital data processing in radio astronomy. For the past twenty years the main focus of Mark's research has been human-computer interaction. His work in this area includes the MINNIE cad system, the invention of the bifocal display, the development of the Lean Cuisine notation for the description and design of direct manipulation interfaces, techniques for the evaluation of interfaces, and more recently the study of CSCW systems. Mark has associated interests in techniques for information representation and interacting with information, and in computer graphics and image processing.

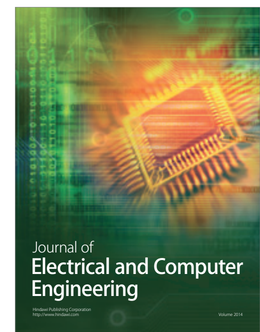
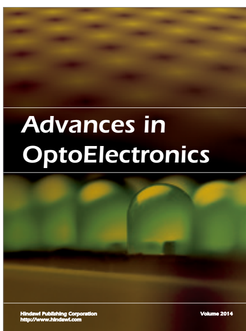
Journal of
Engineering

The Scientific
World Journal

International Journal of
Rotating
Machinery

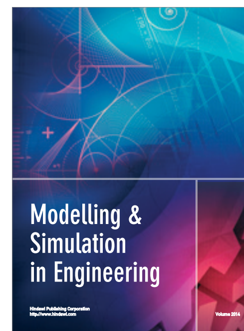
Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

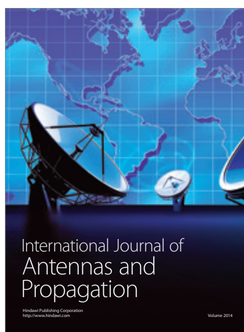
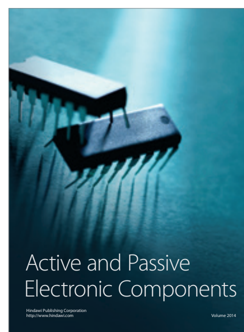
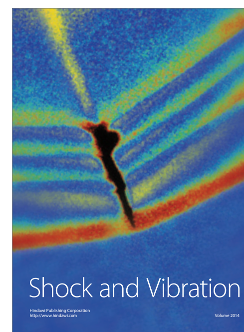
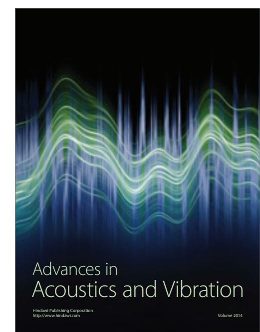
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration