Working Paper Series
ISSN 1170-487X

PICSIL: Integrating graphic system design and automatic synthesis.

**by Murray W. Pearson,
Paul J. Lyons and
Mark D. Apperley.**

Working Paper 95/3
January 1995

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

PICSIL: Integrating graphic system design and automatic synthesis

MURRAY W. PEARSON[∂], PAUL J. LYONS[¥] and MARK D. APPERLEY[∂],

[∂]Department of Computer Science, University of Waikato, Hamilton New Zealand
[¥] Department of Computer Science, Massey University, Palmerston North, New Zealand

Abstract

We present an approach to the design of complex logic ICs, developed from four premises.

First, the responsibilities of a chip's major components, and the communication between them, should be separated from the detailed implementation of their functionality. Design of this *abstract architecture* should precede definition of the detailed functionality.

Secondly, graphic vocabularies are most natural for describing abstract architectures, by contrast with the conventional textual notations for describing functionality.

Thirdly, such information as can be expressed naturally and completely in the idiom of the abstract architecture should be automatically translated into more complex, lower-level vocabulary.

Fourthly, the notations can be integrated into a single, consistent design-capture and synthesis system.

PICSIL is a preliminary implementation of a design environment using this approach. It combines an editor and a synthesis driver, allowing a design's abstract architecture to be created using a graphical notation based on Data Flow Diagrams and state machines, and its functionality to be designed using a more conventional textual hardware description language. On request, it also translates a design into appropriate input for synthesis software, and controls the operation of that software, producing CIF files suitable for fabrication.

Thus computer systems become appropriate for *ab initio* design production rather than *post facto* design capture.

KEYWORDS

Hardware Description Languages, Data Flow Diagrams, Synthesis, Visual Languages

IC design is a complex business

Chip fabrication has progressed remarkably rapidly since it began over thirty years ago, but the greatest advances have been in the construction of simple devices like memories, which contain regular arrays of identical elements [2]. This regularity makes them easy to scale up, and they have been implemented in chips with transistor counts well into the millions. However, less repetitive devices, such as processors, can only be produced to the same scale if manufacturers invest huge amounts of effort and money in their design. Small-run, special purpose ICs of such complexity are thus rarely, if ever, produced.

IC designers have been fighting complexity in the design process since the late seventies, the first major victory in this battle being Mead and Conway's [16] structured design methodology. Since then, a large number of computer aided design (CAD) tools and Hardware Description Languages (HDLs) have been developed to aid in complexity management. CAD tools allow designers to work at higher levels of abstraction by automating parts of the design process, while HDLs allow designers to communicate their design intent to these CAD tools.

A large number of different hardware description languages have been developed in recent years. VHDL [21] and Verilog [24] have gained the widest acceptance in academia and industry. While they differ in their syntax and semantics, both represent designs using a model based on an interconnected set of components. Each component can be described by its behaviour or by a set of lower-level interconnected components. Also common to these languages is the notation used to express a design as single-dimensional description.

Using a one dimensional textual representation of a complex hardware design results in the obfuscation of many aspects of the design [9], [20]. Designers often produce informal, graphical representations of complex devices to assist them to develop their understanding of complex devices before representing them in an HDL. (Authors of textbooks use similar notations to assist readers understand systems.) The diagrams almost always contain interconnected modules, each responsible for some aspect of the system's overall functionality. However, conventional design systems do not allow such understanding-enhancing diagrams as part of the formal design specification, so they must then be translated, by hand, into a textual representation. Thus the designer has to re-create the same information, but in a form which is more tedious to generate and less immediately comprehensible. Further, because there is no automatic maintenance of consistency between the diagrams and their textual equivalents, the diagrams quickly become out-of date.

We have coined the term *abstract architecture* to describe a system representation which suppresses detailed description of functionality, and emphasises the major communicating devices, their hierarchy of responsibility, and the communication paths between them. We believe that abstract architectures are best represented by a two-dimensional graphical notation, whereas detailed functional specifications are more appropriately represented by textual notations (see also [25]).

The PICSIL IC design environment has been developed in accordance with this philosophy. It has a graphical editor for directly capturing a design's abstract architecture. The graphical information never needs to be regenerated as text by the designer. Instead, other software in the PICSIL suite amalgamates it with textual functional specifications. These two levels of system specification, the graphical abstract architecture, and the textual specification of detailed functionality, are the user's major input into a system design. After a design has been captured, PICSIL translates it, through various stages, to a form suitable for fabrication (CIF in the current implementation). Some of the subsequent translation phases are implemented by purpose-written components of the PICSIL software suite; some are implemented by public-domain software; the whole sequence is directly controlled by PICSIL's Synthesis Manager.

Related Work

PICSIL is not the first system to represent hardware graphically. Schematic capture tools have been widely used to capture designs at the gate level for some time, and although they are not yet widely accepted, tools which use graphical representations have been developed to capture designs at higher levels of abstraction.

One of these, state charts [8], a visual language for reactive systems, has been applied to the specification of behaviour. While state charts are well suited to specifying control, they deal with monolithic designs, and are thus inappropriate for use in the early, high-level, design stages when the designer is concentrating on separating the responsibilities of a system's major components.

Several systems have been implemented to allow the graphical specification of VHDL designs. AVE [7] and vVHDL [9] both allow the structural parts of VHDL description to be described graphically. In addition, vVHDL has a visual representation for all behavioural constructs. However, AVE and vVHDL are both based purely on a visual representation of VHDL and do not fully utilise some of the advantages of graphical notations.

While the ability to visualise a design at this comparatively low level of abstraction may work well for the small examples given in the papers cited above, it is less suited to the description of large systems. For example, neither language provides facilities to represent or partition the different types of data that are transmitted between modules. If there are a large number of communications channels in a diagram (particularly in the top level diagrams) the diagrams become very cluttered and difficult to read.

The HardwareC language [14], while purely textual, does contain a number of concepts which PICSIL uses. Like VHDL and Verilog, it allows a design to be described as a mixture of structural interconnections of components and sequences of operations. However a number of major differences exist. First, HardwareC has been designed so that all constructs are synthesisable, whereas, in VHDL and Verilog, the main emphasis has been on simulation. Secondly, HardwareC supports two different types of interconnections: ports and channels. Ports are similar to the interconnection provided in VHDL and Verilog in that it is left to the designer to specify any necessary communications protocols. Channels, on the other hand, can be used to connect components within the design and use a predefined synchronisation protocol which allows the designer to concentrate more on specification of the modules and less on how they interact.

PICSIL uses three complementary hardware description notations integrated into a single design capture and synthesis environment. The notations are a graphical notation (Data Flow Diagram) for top-down, modular, representation of a design's abstract architecture; a graphical notation (state-machine) for representation of module activation within the design; a textual notation (based on HardwareC) and augmented with constructs for representing building blocks missing from HardwareC such as memory and high level module activation mechanisms for controlling the functionality of the design's modules.

PICSIL represents building blocks as DFDs

Data Flow Diagrams [5] were identified as the basis of a suitable notation for representing the organisational structure of hardware designs. Figure 1 shows a conventional DFD, which comprises an arbitrary number of intercommunicating modules (variously called external entities (squares), processes (circles), and data stores (heavy horizontal lines)), connected by data flows. It is displayed as a directed graph, with arrows to indicate the direction in which information flows. Primitive processes have a text-based functional definition. Non-primitive processes are defined organisationally, in terms of lower-level DFDs, so that the abstract architecture may be defined as a tree before any functional definition is required.

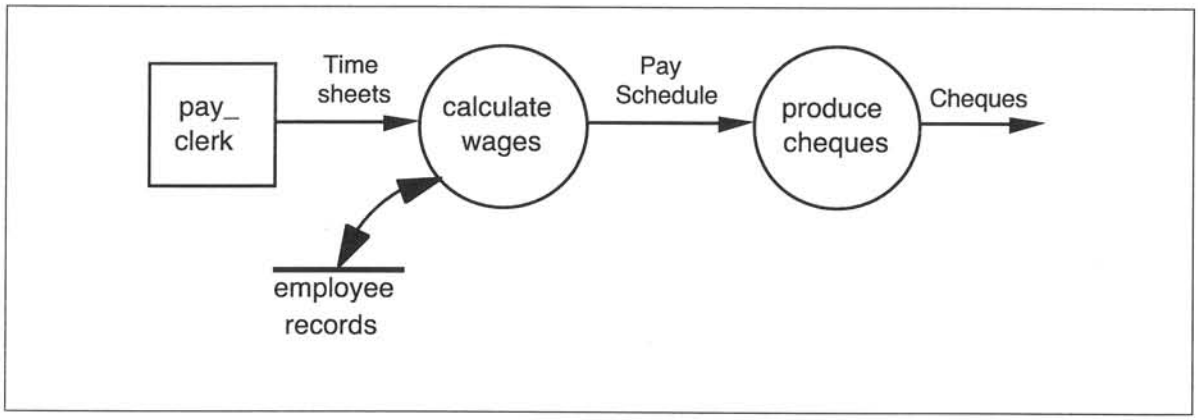

Figure 1. Conventional DFD notation

In adapting conventional DFDs for the unambiguous specification of hardware to a chip synthesis system, we had to "fortify" their syntax in three areas.

First, the diagrammatic syntax is not completely standardised, as DFDs were originally designed for drawing by hand, and various authors have adapted the notation to suit their own ends. Further, returning to the original notation was not desirable, as it lacks some important features, (such as a construct for specifying activation of processes). Consequently, the first improvement was merely to choose a single, consistent diagrammatic syntax for defining PICSIL DFDs.

Secondly, the informal "structured English" used in the conventional notation for specifying a DFD's data dictionary (definition of flow formats) and process transform (specification of process' functionality) was replaced by formal textual and diagrammatic notations.

Thirdly, some modest additions to the common software-oriented vocabulary were introduced to make DFDs more suitable for hardware description.

Maps well onto designers' mental model of hardware

Design of large systems, whether they are hardware or software, generally involves dividing complex tasks into simpler ones, as well as specifying the responsibilities of these simpler tasks . Conventional programming languages, which are the basis of many hardware description languages, were designed in the days before monitors capable of displaying multiple-windows and graphical input were readily available. Consequently they intermix these levels of responsibility (subroutine scope information and subroutine functionality specification are intertwined). DFDs, on the other hand, separate these activities into distinct phases, allowing the designer to focus creative energy on one area 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.
- Two-dimensional DFDs have a structure which matches the two-dimensional nature of hardware.
- High-visibility, directional, communication paths between processes emphasise patterns of responsibility within the system.
- Organisational and functional specifications are distinguished by their graphic and textual vocabularies.

Divisions of a PICSIL design: DFD, controller, text

PICSIL designs are expressed in terms of three separate notations, each optimised for use in a different area. We have described the DFD briefly above. Of the other two notations, one, for describing controllers, is graphical and one, for describing the data processing functions of a process, is textual. Before dealing with their syntax and semantics, let us consider why it is appropriate to choose such an arrangement.

Controllers govern the activation of a DFD's processes, so that they can be made to perform their function in response to particular events. Of course, the process transform could contain conditional tests to allow or inhibit its functionality, but this level of control is a management activity, which would be out of place in low level functional code. Controllers give PICSIL a means for specifying control at a management level; the notation is based on the extended DFD methodologies of Hatley and Pirbhai [11], and Ward[27].

A controller is a finite state machine responsible for activating and deactivating a DFD's processes. There may be only one controller per DFD; if none is specified, its processes are continuously

active. Associated with each state of the controller is an activation table defining the set of processes to be deactivated during that state. State transitions within the controller are initiated by Boolean combinations of events reported by the processes. Each transition may cause actions; single-bit outputs from the controller to particular processes.

By contrast with controllers, which implement a management-level abstraction, the data-processing specifications in PICSIL are comparatively low-level. Specifically, their vocabulary is textual and uses abstractions typical of current high-level programming languages.

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, a textual language based closely on HardwareC [14]. PPTL is procedural and allows a designer to specify hardware operations in terms of conventional programming language abstractions. Variables are used to represent registers; arithmetic and logical processing units operating on the values in those registers can be generated from arithmetic and logical expressions involving variables; functionality can be forced to be sequential, conditionally parallel or parallel, by appropriate labelling of statement groups in the language.

At an organisational level, PPTL possesses constructs for implementing the functional components of DFD processes, and provides data transmission statements which implement the data-driven process-synchronising mechanism of DFDs. Thus the correspondence between the two is very close, and automatic integration of the high-level organisational structure of the DFDs with PPTL is comparatively simple.

In order to illustrate the syntax and semantics of PICSIL's various components, let us consider the design of an Ethernet coprocessor, a device for transferring data between an Ethernet network and a processor bus. Commands may also be sent over the bus to configure the Ethernet coprocessor. Figure 2 shows a PICSIL representation of this system as it would appear on the designer's screen. We will deal with the components of the diagram in turn.

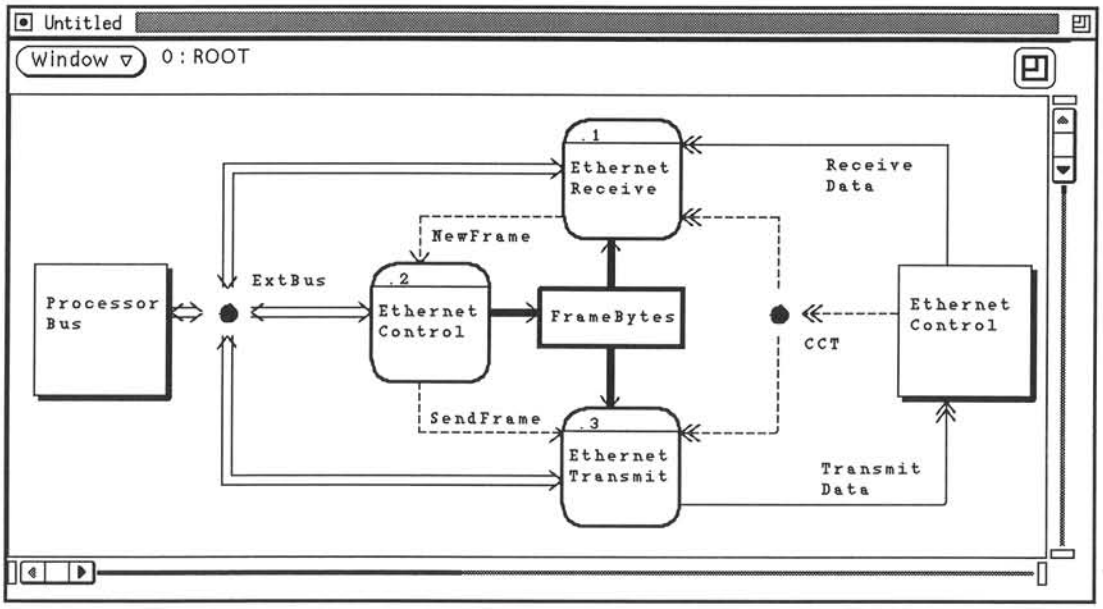

Figure 2. Root DFD for an Ethernet co-processor

Processes (see figure 3(a))

Processes transform incoming flows into outgoing flows. They contain a name and a unique address within the diagram. As processes are nested, a unique address within the tree can be generated for any process by recursively prepending the addresses of all its parent diagrams, up to the root of the tree).

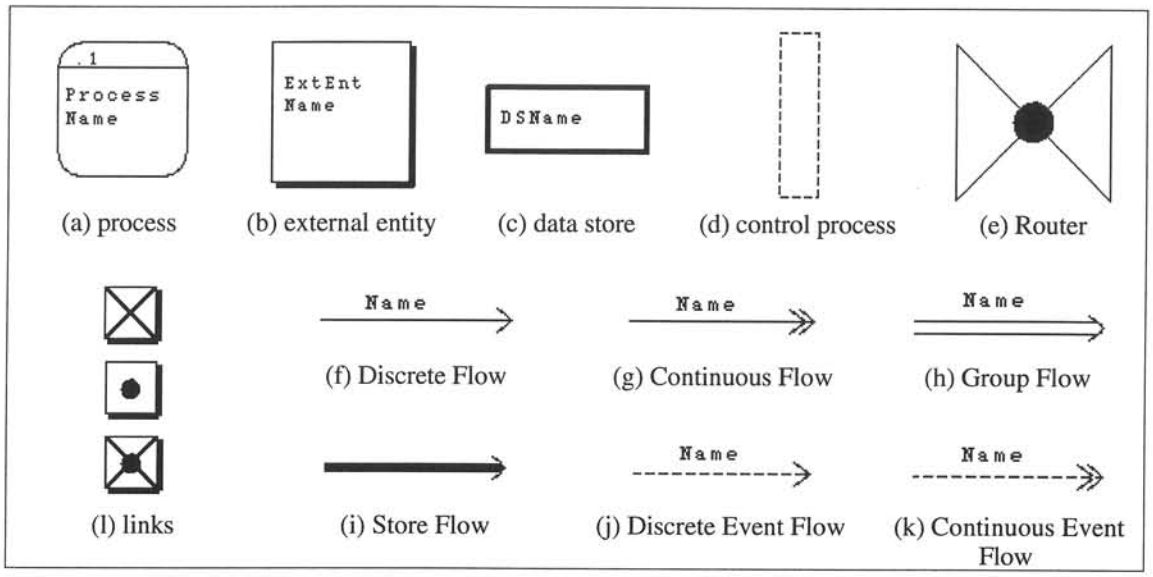

Figure 3. Components that appear in PICSIL DFDs

Each process in a diagram is decomposed and defined in more detail. If the level of abstraction is low enough for its functionality to be defined briefly and concisely, then it is defined in the data dictionary. Otherwise the process is defined as a lower level (or child) DFD. Processes that are defined in the data dictionary are called primitive processes while processes defined by a child DFD are known as non-primitive processes.

Non primitive process decomposition

Figure 4 shows the decomposition of the non-primitive process `EthernetReceive` into three sub-processes which in turn can be decomposed. It is important to note that decomposition of a process does not add new functionality to the system. It only defines it in more detail. Thus a child process must have the same set of inputs and outputs as its parent.

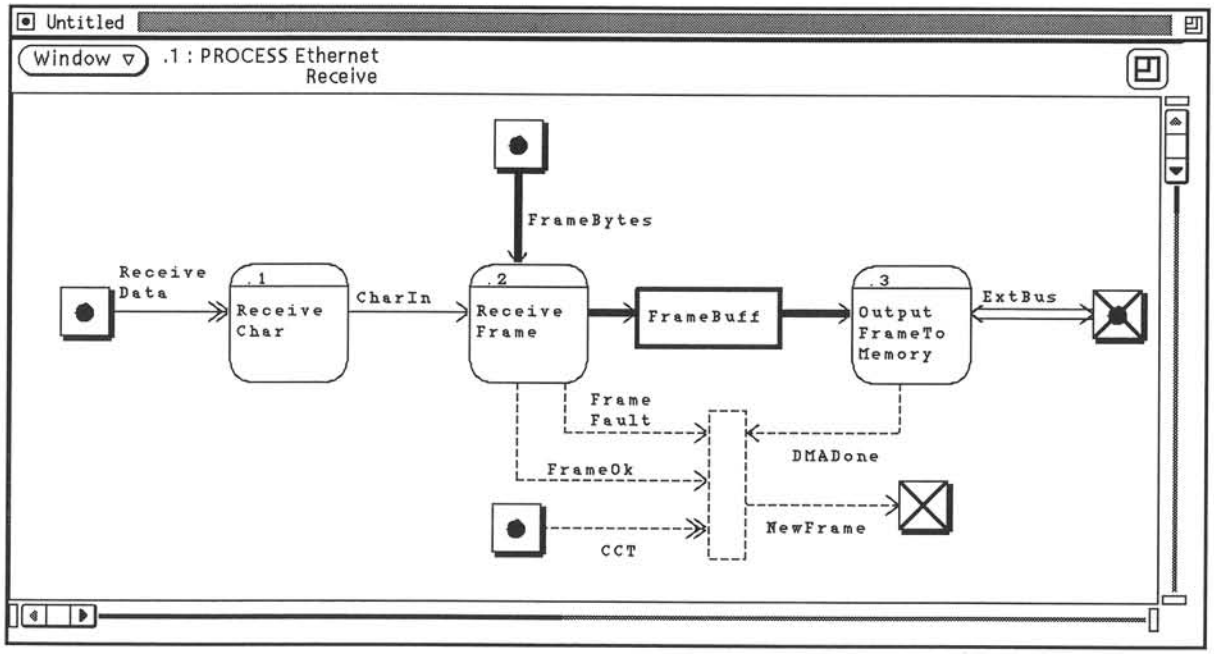

Figure 4. DFD 1.3. (The refinement of process 1.1 (Ethernet Receive))

Links (small shadowed boxes in figure 4)

Links are purely notational symbols which are added to the child diagrams automatically, to aid the designer ensure that all flows attached to a parent process also appear in the child diagram. There are three types of link: import, export and import/export; 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 containing a cross overlaid with a disc can also occur when a bidirectional data flow occurs between a child DFD and its parent.

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, unless prevented from doing so by a controller, restart themselves on completion. Figure 5 shows the data dictionary entry for the primitive process `ReceiveChar`. It can be seen that it resembles a C function definition, and can be interpreted in much the same way: the name of the process is specified first, followed by the declaration of its local variables, then statements for input processing and output.

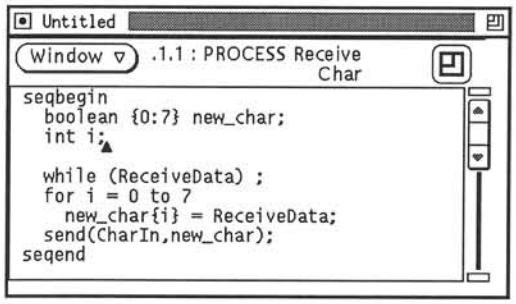

Figure 5. The Data Dictionary entry for the primitive process `Receive Char`

External Entities (see figure 3(b))

External entities represent connections to external I/O ports, and during synthesis, result in connections to bonding pads on the periphery of the chip.

Data Stores (see figure 3(c))

A data store is a configurable random-access memory. Its contents may be accessed (i.e. read or written) by a number of processes. The readout operation is non-destructive.

If a process wishes to read data from a data-store, process it and write the result back, and another process wishes to read the same data, a potential consistency problem arises. To prevent this, data stores enforce data locking. When a process reads data from a store in read/write mode, the store is locked, and no other processes may access it. The store remains locked until either the process which has access to it terminates or it executes a statement which explicitly unlocks the store. Any other process trying to access a store while it is locked is suspended until it becomes available again. Processes that use a store in a read-only or write-only mode only lock the data store for the time the process is accessing the store's contents.

Data Flows

A data flow is a communications channel carrying one or more bits of data between other components of the PICSIL design. 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 represents a channel for communicating data between processes in a DFD. The communication is data-driven. That is, 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 the data-driven protocol is generated automatically.

Send and *receive* statements are provided in the data dictionary language for specifying transfer of data over discrete flows. For example, when the process `ReceiveChar` in figure 4 produces some send data for the discrete flow `CharIn`, it executes a *send* statement (`send(CharIn, data);`). The process `ReceiveFrame` receives new characters off the flow by executing a receive statement (`receive(CharIn, data);`). Every discrete flow must have an entry in the data dictionary which defines the type of data it will carry. For example the discrete flow `CharIn` would is declared as "`flow discrete boolean {0:7} CharIn;`".

Continuous flows (see figure 3(g)) Continuous flows, for off-chip communication with devices which do not use the data-driven synchronising protocol. Values are read from, or written to, continuous flows using the data dictionary language's *read* and *write* statements. The value output during a write statement remains on the flow until a new value is output. When a read statement is executed (e.g. `read(ReceiveData, NewBit);`), the value input is the value of the continuous flow at the time the statement is executed.

A continuous flow may have arrows at both ends, indicating that half duplex data can flow in both directions, when it is known as a bidirectional continuous flow. 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(i)) Store flows allow processes to access the contents of a data store. A store flow is made up of three components: an address, which specifies which location in the store is addressed; locking information, which is supplied by the PICSIL compiler to prevent deadlocks; and the data which is being read or written.

The data dictionary language's `stwrite` and `stread` statements allow a process to access the contents of a data store. `Stwrite` allows the result of an expression to be written to a particular location in the named store. For example `stwrite(FrameBuff[position],Char)` writes the value `Char` into the data store `FrameBuff` 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. Store flows are unnamed and are used to indicate that a process accesses and/or updates information in the connected data store. The direction of 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. Flow grouping is not recursive.

Discrete and continuous event flows (see figures 3(j) and 3(k) respectively) Event flows alert the DFD's controller to events in the process, and allow bidirectional communication between the controller and the outside world. Note that activation and deactivation of the processes is not represented explicitly by any flow, as the controller controls all processes in the DFD.

Control Processes (see figure 3(d))

Controllers represent an interface to the diagram's controller. A diagram has no more than one controller, but there may be more than one interface to it shown on the diagram, for convenience of drawing. A DFD's processes can be activated and deactivated by the controller in response to

events which are reported to it along the event flows. If no interface to a controller is shown on a DFD, the DFD has a null controller, and all its processes are always active.

In addition to the components shown in Figures 3 and 4 there are also routers and elements, both of which are used to reduce complexity in heavily populated diagrams.

Routers

Routers represent a programmable data switch, and allow data on any incoming data flow to be delivered to any outgoing data flow. Before a data-exporting process can send data through a router, it must lock a connection to the desired export flow from the router to make it behave as a conventional discrete data flow. The connection must be explicitly unlocked when the process wishes to release it. Locking and unlocking are performed in the data-processing code.

Figure 6 shows a node in a local area network. Data arriving *via* an input port is routed to an output port according to information contained in a virtual circuit table (not shown).

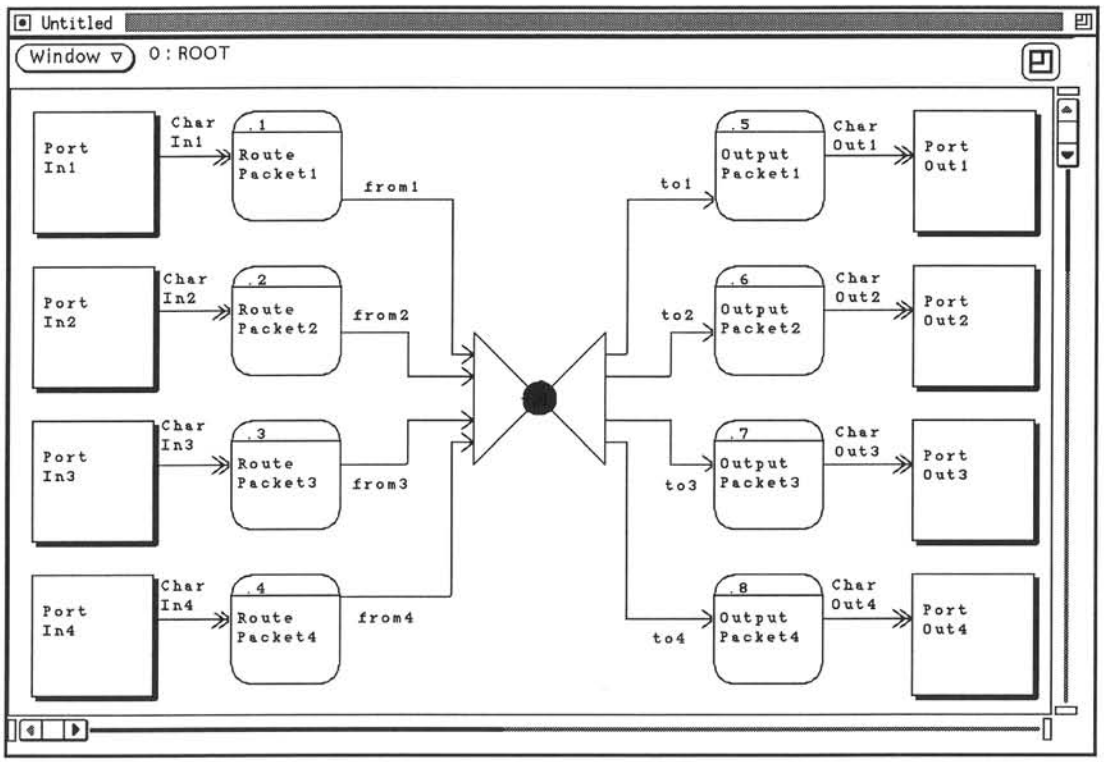

Figure 6. Routing packets to an output port in a LAN node

Elements (Dashed lines around a section of a DFD)

An element represents a single element of a repeated structure. That is, the subsystem enclosed within it is repeated a number of times. The specification in the bottom left corner of the element defines the limits of the array, and, by implication, the number of elements it contains. Flows which cross the boundary of the element are shown as a single representative flow within the element (though this might itself be a group flow), and a group flow with the appropriate number of components outside the element. However, continuous and store flows may be exceptions to this rule, being represented as single flows both within and without the element. This allows identical information to be delivered to all elements of an array. In this case, a connector can be placed at the element boundary.

Figure 7 shows the same devices as defined in Figure 6, but represents repeated components as an element.

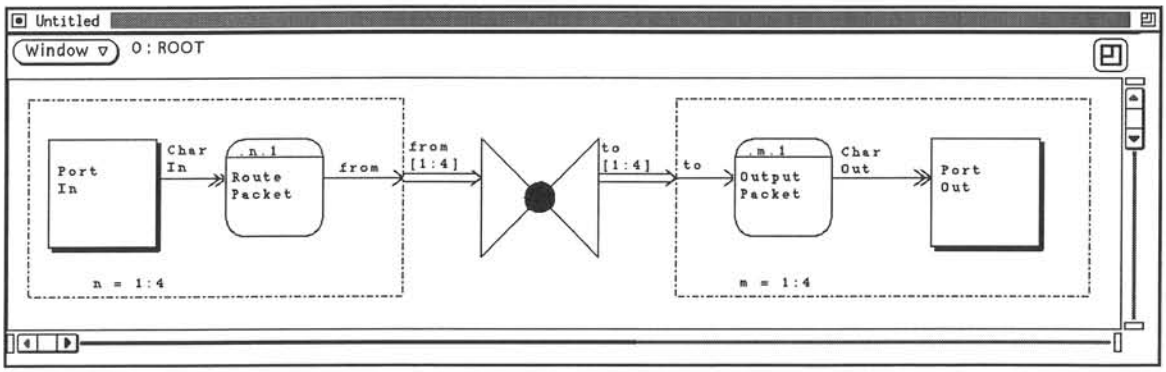

Figure 7. LAN node with four identically defined input ports and four identically defined output ports

The node has `n` identical input ports and `m` identical output ports (`m` and `n` are both equal to 4). A representative element of the array of input ports is shown with the legend `n = 1:4` at its top left corner, to indicate the number of identical items with that definition. In the PPTL code for each node, the identifier `n` is used to distinguish it from the other elements of the array.

As it is possible for a repeated component of a design to contain repeated sub-components, the PICSIL notation allows element definitions to be nested.

Components of a controller

We will now consider the syntax and semantics of the components of the graphical controller specification.

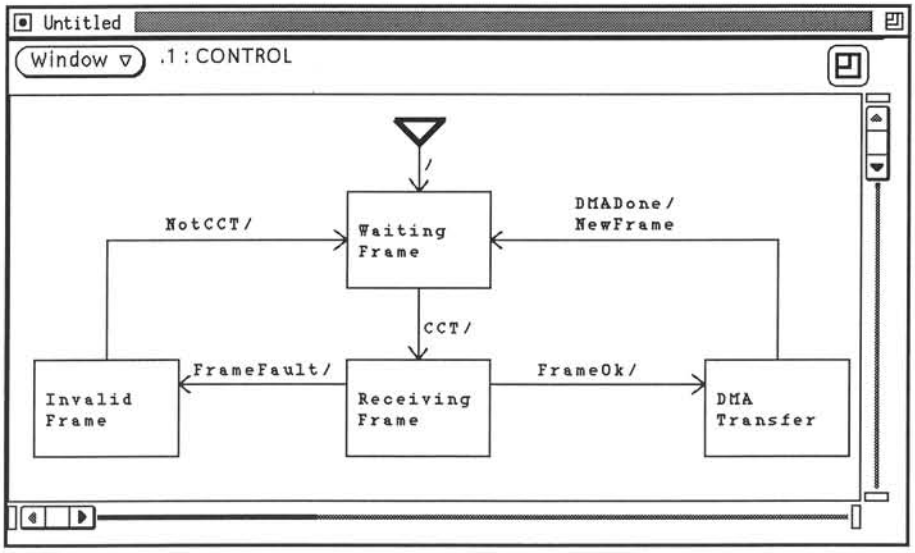

Figure 8. Controller for DFD 0.3 (combine streams)

The processes in a DFD can be continuously active - in which case, no controller need be specified - or conditionally active - in which case their activity is specified in a process activation table which associates process activations with the different states in the controller state machine.

If a controller exists, it is represented as a directed graph of named rectangles (the states) connected by labelled arrows (state transitions). The labels have two parts, separated by a slash (/)symbol. The first part is the name of a Boolean combination of events which, if true, cause a transition out of the state. The name is printed plain if it is the name of a single event or its inverse, and **bold** if is represents a more complex combination. It may be omitted if the state is to be occupied for only a single clock pulse, and it possess only one outward transition. The second part of the label is a

name denoting a list of events which are signalled when the transition occurs. Again, plain text denotes a the use of a single event's name as the label, **bold** denotes a more complex set.

The controller can specify the degree of activation of the DFD's processes. Input acceptance and output generation occur on state transitions (though a state transition need not be associated with either I/O event): in a particular state, the degree of activation of each of the processes in the DFD may be specified as continuously active, activated once, or deactivated.

Textual representation of functionality

PPTL, the language for defining processes' functionality, is based closely on HardwareC. PICSIL automatically translates the individual leaf-node DFD processes into HardwareC process skeletons. The designer then fills out the definition using a statement sequence which corresponds to a set of hardware operations. A number of extensions have been made to the language to support the different types of communication between objects in a PICSIL design. Most of these extensions have been outlined in earlier sections.

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 graphical objects. For example, if a data flow is changed, all other diagrams which use it are automatically updated. Nevertheless, some support for consistency maintenance between graphical and textual views can be provided, as DFDs "know" which primitive (textual) processes their data flows are connected to. This allows the system to propagate changes in DFDs by adding comments at the top of any affected primitive processes, to remind the designer of the necessary alterations. Researchers in visual programming languages [10], [15] discuss techniques that can be mor effectively used to maintain the consistency between graphical and textual views that would be necessary in a production system.

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 type 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 greatly enhance the editor and allow the design space to be explored more effectively.

Since work on PICSIL began, a number of frameworks for building CAD tools including MViews [10], Ptolemy [1] and Nelsis [16], have become available.They allow rapid development of visual programming environments and include facilties to implement version control and consistency management with minimal effort. We intend to use one of these systems for building a new version of PICSIL which will address the issues outlined above.

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 to augment, and provide bridges between, the public-domain systems. Presiding over them all is a synthesis manager which reduces the complex task of controlling the lower-level synthesis path to supplying a few parameters to a single function. Let us consider this last program first.

Because of its complexity, the production of a chip layout from a high level description is traditionally split into three stages: behavioural synthesis, logic synthesis and layout synthesis [26].

Tools based on logic and layout synthesis are mature enough to be used in production systems, but the development of high level synthesis systems is mainly restricted to research environments which do not generally deal with lower level aspects such as layout synthesis [3].

Because of the limited budget associated with this research, the selection of synthesis tools was essentially limited to those available in the public domain. A limited review identified the following synthesis tools available in the public domain: FIRST [6], VAST [13], a silicon compiler based on Asynchronous Architecture [12], the System Architect's Workbench [23], the Olympus system [4] and Berkeley Octtools [18].

None of the above tools were suitable alone for use in the PICSIL synthesis system as either they did not support the full synthesis path, or PICSIL constructs did not map well onto the synthesis system's input HDL. However, 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 9) 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).

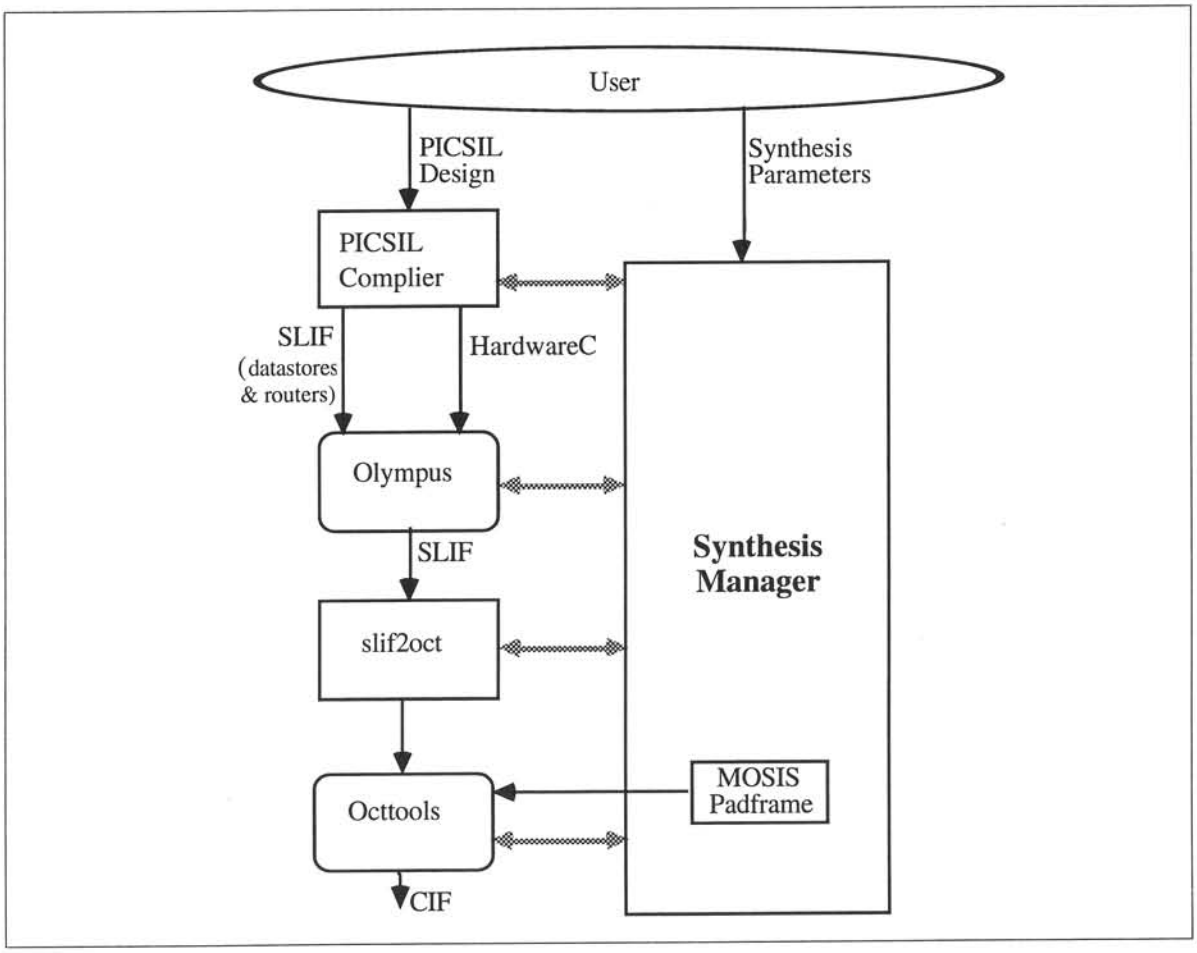

Figure 9 PICSIL Synthesis Path

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. The hardwareC file is translated, in several steps, with dummy components in place of the routers and data stores. Then the dummy components are then replaced by the correct SLIF versions, which were synthesised and saved by PICSIL, to create a complete SLIF 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, 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.

Parameterising the synthesis path

Defining the precise synthesis path to be followed is simple in concept. However, in reality, the path is implemented by 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. PICSIL's Synthesis Manager restores the underlying conceptual simplicity to this sequence of operations by allowing the user to specify a limited number of options *via* a control panel in a dialog box accessed from within the editor (see figure 10).

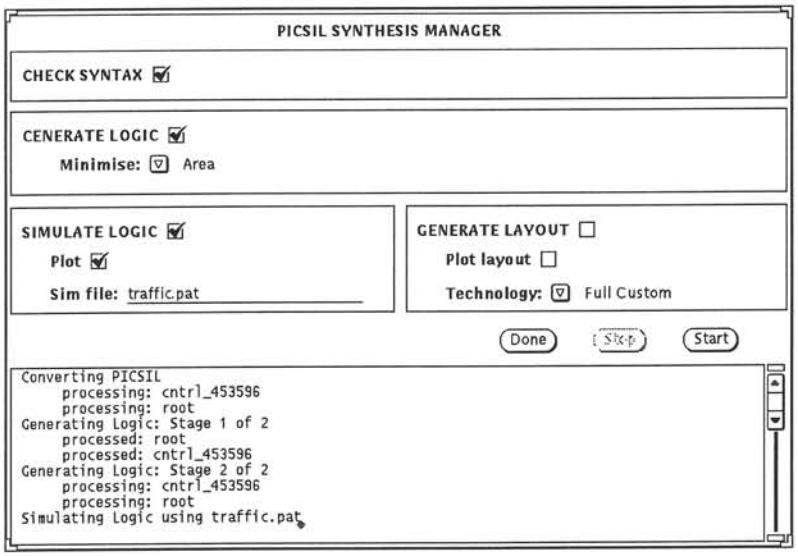

Figure 10 Synthesis Dialogue pop-up window

The Synthesis Manager, using these high-level parameters, then generates and issues the necessary commands to drive the synthesis process. If a syntax error is detected in a PICSIL description during initial synthesis, the Synthesis Manager opens a window in which the object or line of text containing the syntax error is highlighted. A syntax error message is also displayed in a separate pop-up dialog box and the synthesis process is terminated. Also during synthesis, the synthesis manager captures the output of the currently invoked tool, which it uses to make decisions on synthesis steps required later, and to report status information back to the designer.

Verification

Verification of the correctness of an integrated circuit is an important part of its design. Olympus and Octtools both provide simulation tools for verification purposes. To allow the designer to remain in the PICSIL environment when simulating a design, the synthesis manager provides an interface to Mercury, an Olympus tool which simulates SLIF descriptions. To perform a simulation, Mercury requires, in addition to the SLIF description, a file containing an input test pattern. The file is made up of two parts, a list of the inputs, and a set of input vectors. While the list of inputs can be generated automatically by PICSIL, the set of input vectors cannot, and these have to be supplied by the designer using a text editor provided within the PICSIL environment.

When a design is to be simulated, the synthesis manager produces a script which initiates Mercury, supplies it with the input test pattern, and instructs it to perform the simulation. If required, a plot of the simulation results can be displayed within the PICSIL environment.

Pad Frame Definition

To complete a design after its logic has been synthesised, bonding pads must be added around the periphery of the chip, so that connections may be made between the I/Os on the chip and the leads of the IC package. While Octtools can place a padframe around a synthesised design, extra information describing the padframe and the mapping of the design's I/Os to particular pads is required, and this information cannot be derived directly from the PICSIL design. Facilities have been included into the PICSIL environment to allow these details to be defined. If the logic core is too large to fit into the specified padframe, the synthesis manager will omit this final step and notify the designer.

The current version of the synthesis manager uses the MOSIS TINY padframe[1] which encloses a logic core of approximately 2mm by 2mm. To allow larger padframes to be incorporated will require only minor changes to the PICSIL environment and definition of a template describing the structure of the padframe.

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. To date, three have been automatically synthesised and one has been fabricated. 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, which has shown that all the tools in the synthesis path have functioned correctly. One of the synthesised designs (a traffic light controller) has been successfully fabricated using Orbit Semiconductors Foresight program [19].

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 nearly all cases, the PICSIL representation took up less space than the conventional HDL representation. 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; there is a distinct resemblance between the PICSIL's intercommunicating processes and the message-passing objects which are currently enjoying fame as a way of improving conventional programming language use.

CONCLUSIONS

PICSIL allows a designer to specify less detail in the description of a system than other hardware description languages. However, that is not its chief virtue. That is to allow designers to deal with a complex design problem in a structured way, by providing natural vocabularies for representing a design's organisational interactions, on one hand, and its functionality, on the other. Languages for describing functionality have been extensively researched, and it has been necessary only to adapt an existing language slightly to fit in with PICSIL; a language other than HardwareC might have been chosen for this purpose, and indeed, HardwareC might yet be replaced - by VHDL, for example.

---

[1] MOSIS (MOS Implementation system) was set up in 1980 to provide fabrication services to US government contrators, agencies and university classes. The MOSIS tiny padframe is madup of 40 pads (10 along each side) which are at fixed locations inside a 2.4 by 2.4 mm box.

The important development has been the identification of system organisation as a separate aspect of the design problem, which needs a vocabulary of its own, unconstrained by the inherited linguistic culture of functionality-based languages. PICSIL DFDs are:

- two-dimensional for natural representation of networks;
- restricted to a small vocabulary for ease of learning.
- formal for direct translation to hardware;

While the vVHDL and AVE systems partially identify system organisation as a separate aspect of the design problem, they are strongly influenced by existing hardware description languages and do not fully take advantage of features available with graphical notations. Features of the PICSIL notation which are not present in other visual hardware description languages 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.

To support design using PICSIL and help bridge the gap between an initial specification and its representation in a computer readable form, we have developed the PICSIL editor. The editor provides for direct input and modification of a design expressed in a notation combining graphics and text. It is simple to use and is based on a consistent user interface which makes use of multiple windows, menus and buttons.

While the PICSIL notation can represent designs simply and elegantly, more powerful editing operations such as multiple undos, better consistency management, version control and facilities to allow parts of designs to be reused would be necessary in a production system.

In spite of this lack of superficial gloss, the PICSIL editor is capable of capturing designs expressed in a more natural vocabulary than any other Hardware Description Language currently available. Also, when evaluated against Sequin's guidelines [22] 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.

The Synthesis Manager conserves the designer's effort for creative work by automating the mechanical chore of synthesis. It is capable of converting a PICSIL description into a form suitable for synthesis using OLYMPUS and OCTOOLS and can automatically initiate all the necessary tools in sequence in these suites to synthesise a PICSIL design into a CIF layout which can in turn be supplied to a silicon foundry for fabrication.

This project has achieved its aim: demonstrating the practicability of a new generation of HDLs which take full advantage of today's GUI environment - and which therefore would therefore have been infeasible until recently. The PICSIL notation described herein is our first attempt at such an HDL. It is characterised by a natural vocabulary for representing designs simply and elegantly at the system level of abstraction. The associated editor and synthesis manager capture the design and automate the "busy-work" of layout synthesis, leaving the designer free to concentrate on the higher-level aspects of the design which require greater creativity. 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.

References

[1] Buck, J., Ha, S., Lee, E.A. and Messerschmitt, D.G., Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", Internaional Journal of Computer Simulation, special issue on Simulation Software Development, Jan 1994

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

[3] De Micheli, G., "Guest Editorial: High-Level Synthesis of Digital Circuits," *IEEE Design & Test of Computers*, vol. 7, no. 5, 6-7, Oct 1990.

[4] 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.

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

[6] Denyer, P.B. and Renshaw, D., *VLSI Signal Processing: A Bit Serial Approach*. Addison Wesley, 1985.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] Hirayama, M., "VLSI Oriented Asynchronous Architecture," *13th Annual International Symposium on Computer Architecture*, 290 - 296, June 2 - 5 1986.

[13] Kam, M.C. and Hellestrand, G.R., "The Vast VLSI Architecture and Design Environment," in *Proceedings of the 9th Australian Micro-electronics Conference*, The Institution of Radio and Electronics Engineers Australia, Jul 1990, pp. 145 - 250.

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

[15] 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.

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

[17] Nelsis, *The Nelsis CAD Framework*, Electronic Systems Engineering, Delft Institute of Microelectronics and Submicron Technolgy, Feb 1994

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

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

[20] Pearson, M.W., Lyons, P.J., and Apperley, M.D., "PICSIL A Data Flow Approach to Silicon Compilation," in *Proc. NELCON91*, 1991, pp. 168 - 173.

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

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

[23] Thomas, D.E., Dirkes, E.M., Walker, R.A., Rajan, J.V., Nestor, J.A., and Blackburn, R.L., "The System Architect's Workbench," in *Proceedings of the 25th Design Automation Conference*, Jun 1988.

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

[25] Tse, T.H. and Pong, L., "An Examination of Requirements Specification Languages," *The Computer Journal*, vol. 34, no. 2, 143 - 152, 1991.

[26] Walker, R.A. and Camposano, R., *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, 1991.

[27] 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.