Working Paper Series
ISSN 1170-487X

Parallel Programming With PICSIL1

Murray Pearson
Matthew Melchert

Working Paper 93/11

October, 1993

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

Parallel Programming With PICSIL1

Murray Pearson (m.pearson@waikato.ac.nz)
Matt Melchert (matt@waikato.ac.nz)

Department of Computer Science
University of Waikato
Hamilton
NEW ZEALAND

Abstract

This paper describes the background and development of PICSIL1 a visual language for specifying parallel algorithms using structured decomposition. PICSIL1 draws upon graphical and textual specification techniques; the first for high level structure of an algorithm, the second for more detailed functional specification.

The graphical specification techniques used in PICSIL1 are based on Data Flow Diagrams (DFDs) and are well suited to the assembly and interconnection of abstract modules. Minor modifications to DFDs have however had to be made to make them suitable for describing parallel algorithms, These include the ability to dynamically replicate sections of a diagram and change the structure of parts of a diagram dependent on data being processed.

Work is proceeding on the development of an editor to allow the direct capture and editing of PICSIL1 descriptions, In the near future development of compiler and visual debugging tools are planned.

Introduction

As the theoretical limits to processor speeds that can be obtained from technology changes impend, new approaches are going to be required to allow the long trend of a doubling in CPU speed every two years to be continue (Dillinger88). One approach showing promise is that of parallel processing.

While new architectures are continually being developed for parallel processing , the development of algorithms to take advantage of this parallelism has been more challenging. The use of parallelising compilers to extract parallelism from sequential code have produced disappointing results in terms of speedup produced (Pancake90). On the other hand, hand coded parallelism (using specialised parallel programming languages) has proved to be very difficult and time consuming. Two main reasons have been identified for this difficulty (Socha89). First, management of data and control may be difficult as parallel architectures are more complicated than their sequential counterparts. Second there is often a very large conceptual gap between a programmers mental conception of a problem and the realisation of an efficient parallel program that solves the problem.

This paper discusses how a parallel programming paradigm based on a mixture of graphical and textual notations can reduce this conceptual gap, and introduces a new parallel programming language, called PICSIL1, based on this paradigm.

Expression of Parallel Algorithms using Graphics and Text

Almost all existing parallel programming languages represent a parallel algorithm using a single dimensional text based description. These textual descriptions make it extremely difficult for the programmer to visualise the program's overall structure. In fact, most programmers working in the parallel domain have to resort to a graphical (or multi-dimensional) representations of their code (Suhler90). In most cases these diagrams do not form part of the formal specification for the program, so the diagrams have to be recreated when changes are to be made to the code. Where these diagrams do form part of the formal documentation it is difficult to enforce a mechanism to ensure the diagrams and text remain consistent during their maintenance.

While the use of graphics has proved to be good at allowing programmers to visualise the overall structure of a problem, they have been less successful at representing the more detailed aspects of a problem (Myers90), (Grundy93). In this domain a text based notation has the complementary ability to represent these more detailed aspects (Tse91). This led to the assumption that an ideal parallel programming language should

contain a mixture of graphical and textual representations.

A number of parallel programming environments do make use of text and graphics have appeared recently (Zabala93), (Socha89), (Beguelin93), (Suhler90). Most of these environments, however, make use of visual representations to visualise the execution of a program defined using a textual programming language. Two languages, CODE (Browne89) and Hence (Beguelin93) do, however, make use of graphics and text in the specification of a problem. Both of these languages develop a parallel program as a computational graph where nodes represent the computations to be performed and arcs represent dependencies between the computations. While both of these languages help to reduce the conceptual gap between a programmer's mental model of a problem and its specification in a programming language, further abstractions in the graphical language can help to further reduce this conceptual gap. For example, in Hence each node is numbered which does limits the expressive power of the language. As the only labelling on each node is a number, it is not possible to determine the overall structure of the system without reading lower level descriptions. In the CODE language, however, each of the nodes is given a name (which should indicate the processing carried out by the node) allowing the reader to get an overview of the algorithm without having to read lower level descriptions.

Other work in the area of parallel programming draws a resemblance between the domains of parallel programming and hardware (Thomas93). A notation resulting from this work, called the circuit model, represents a parallel algorithm as circuit. While this notation demonstrates a useful correspondence between parallel algorithms and hardware, the resulting descriptions tend to be at a low level of abstraction, and the language does not take advantage of the benefits provided by graphical notations to represent the overall structure of a problem.

As the number of components that can be placed on a single integrated circuit (IC) continues to increase chip designers have also been experiencing problems with a widening conceptual gap between the designers conceptual model of a problem and its realisation as a piece of hardware. To help reduce this gap, a large number of CAD tools have been developed which support IC design.

The principle input to these CAD tools is a representation of the IC device under design, using a hardware description language (HDL). The HDLs generally allow an IC to be described either structurally as an interconnected set of components, or behaviourally as a mapping of inputs onto outputs, or as a mixture of both. The levels of abstraction that these HDLs support vary widely from the logic level, where a system is described as a set of logic gates, to high-level behavioural languages, where the behaviour of a system is described using constructs similar to those found in programming languages. For the same reasons that programmers working in a parallel domain require diagrams to reason about their programs, so hardware designers need them to reason about their hardware descriptions. Several HDLs which recognise this weakness have been developed which support hardware design at high levels of abstraction and allow these diagrams to be incorporated as an integral part of the hardware description. These include PICSIL (Pearson92) and Envision-VHDL (Toomajanian92).

The PICSIL HDL has been identified as being suitable to base a parallel programming language on because of the useful mix of graphical and textual constructs. Graphical constructs are used to represent the high level structure of an algorithm, while text is used for more detailed functional specification.

PICSIL as a Parallel Programming Language

The PICSIL hardware description language is a hierarchical specification language for digital systems combining graphical (data flow diagrams (DeMarco78)) and textual (hardwareC (Ku90)) notations to allow the capture of specifications, and subsequent synthesis of IC layouts.

PICSIL allows the control of data processing functions to be specified separately from the functions themselves. The data processing functions of a system are defined using functional specifications, including data flow diagrams and data dictionary entries. A data flow diagram allows a system to be broken into component functions (processes) which are interconnected by a network of data flows. The data dictionary in PICSIL is used to define the textual components of a PICSIL design, including the definitions of processes and the definitions of the data carried on the data flows. The control of the data processing functions is defined using a controller specification, including state transition

diagrams and process activation tables (Hatley87).
Although the PICSIL HDL has been identified as a suitable basis for a parallel programming language, there are some differences in the notation required to describe a parallel algorithm than to that to describe a hardware system.

An HDL needs constructs to define general purpose communications protocols so that a system under design can interface to any external device, no matter what the required protocol. In the case of the parallel algorithm the interface to the I/O components (e.g. disk drives and VDUs) are dependent on the system the program is to run on, and can be determined at compile time.

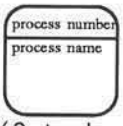
Hardware is inherently static in that once a hardware system has been created it is not possible to dynamically create and delete components in the system as required. In the parallel programming case, however, new processes can be created and deleted dynamically as required.

The rest of this paper describes the PICSIL1 parallel language which is heavily based on the PICSIL HDL.

The PICSIL1 Language

To illustrate the definition of PICSIL1, the design of an image processing system for an underwater robot will be used. The robot scanning a deep sea bed sees different objects that it has to recognise in order to distinguish among fish, rocks, algae and so on. Typically the membership of a class is determined by comparing attributes of a filtered version of the image to those stored in a data base containing attributes of all of the possible attribute types. As the class of an object is required in close to real time, comparison of an image to each of the possible object classes is to be performed in parallel. When the program is started up the image database must be loaded from a disk file into memory. Figure 1 shows the top level PICSIL1 diagram for a under water robot. This example is used in the following sections to illustrate the four basic components of a PICSIL1 diagram. Other components will be introduced later.

Processes



A process transforms incoming flows into outgoing flows. In figure 1 there are three processes (SetupImageClasses, FilterInputImage and DetermineFilteredImagesClass) which each accept various sorts of data from earlier in the system, alter it, then pass it on.

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

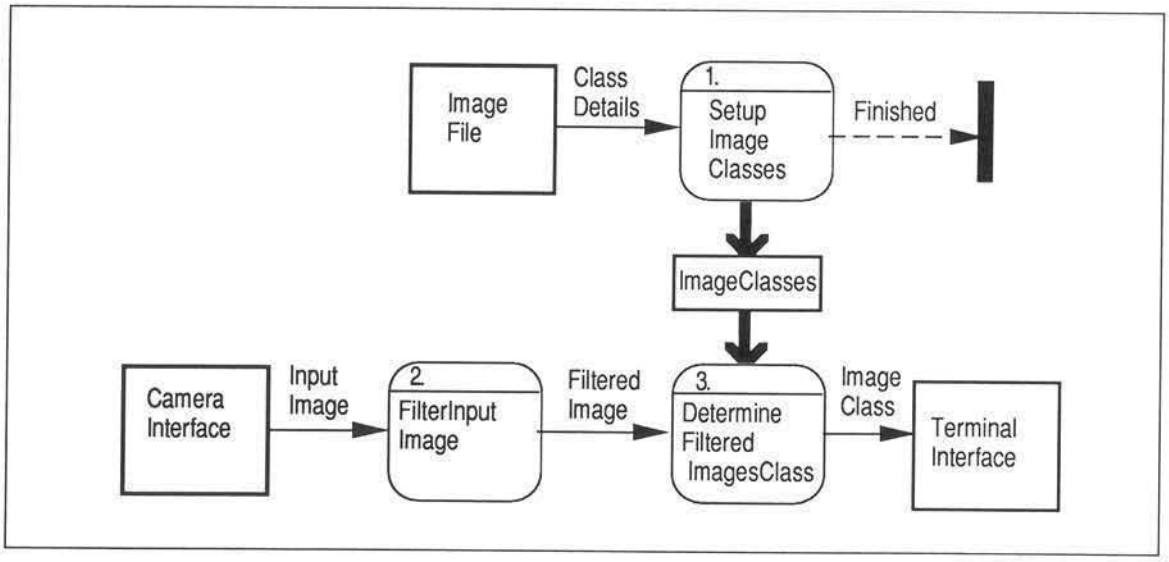


Figure 1 Top level PICSIL1 description for image processing example

Process Refinement

Figure 2 shows the decomposition of the non-primitive process DetermineFilterImagesClass. The process has been decomposed into three sub-processes which in turn can be decomposed. This decomposition of data flow diagrams into increasingly detailed diagrams is called hierarchical decomposition (Cox90).

It is important to note that decomposition of a process does not make new statements about the system, only more detailed ones. As a parent process and its decomposition represent the same information at different levels of abstraction, their inputs and outputs should be identical.

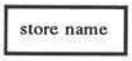
Import and Export Links

When the designer first decomposes a non primitive process, the editor will automatically show all the flows into and out of the parent process as import,  , export,  , and bidirectional, , links. The automatic incorporation of these symbols into the child diagram ensures that the inputs and outputs of the child diagram match those of the parent process. During the editing of a diagram the programmer can see immediately that the diagram is not balanced if links do not have flows attached to them.

Every process in a tree of PICSIL1 data flow diagrams is identified by a label comprising a tree address (automatically assigned) followed by the name of the parent process. In the top level diagram the processes are numbered 1, 2, 3, and so on (in the order of their creation). Processes at lower levels have a correspondingly longer prefix, the children of process 2, for example, being numbered 2.1, 2.2, and so on.
As all processes within a particular diagram have the same prefix to their tree address (the address of the parent process), this is shown at the top left corner of the diagram, and the processes contain only their own unique extension of the prefix.

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 (see section on control extensions), restart themselves on completion. Figure 3 shows the data dictionary entry for the primitive process ReadyImageForParallelSearch.

Data Stores



A data store is a random-access repository for data. Its contents may be accessed (i.e. read or written by multiple processes). The readout operation is non-destructive. That is when data is written to a store it may be read any number of times until new data is written to the same

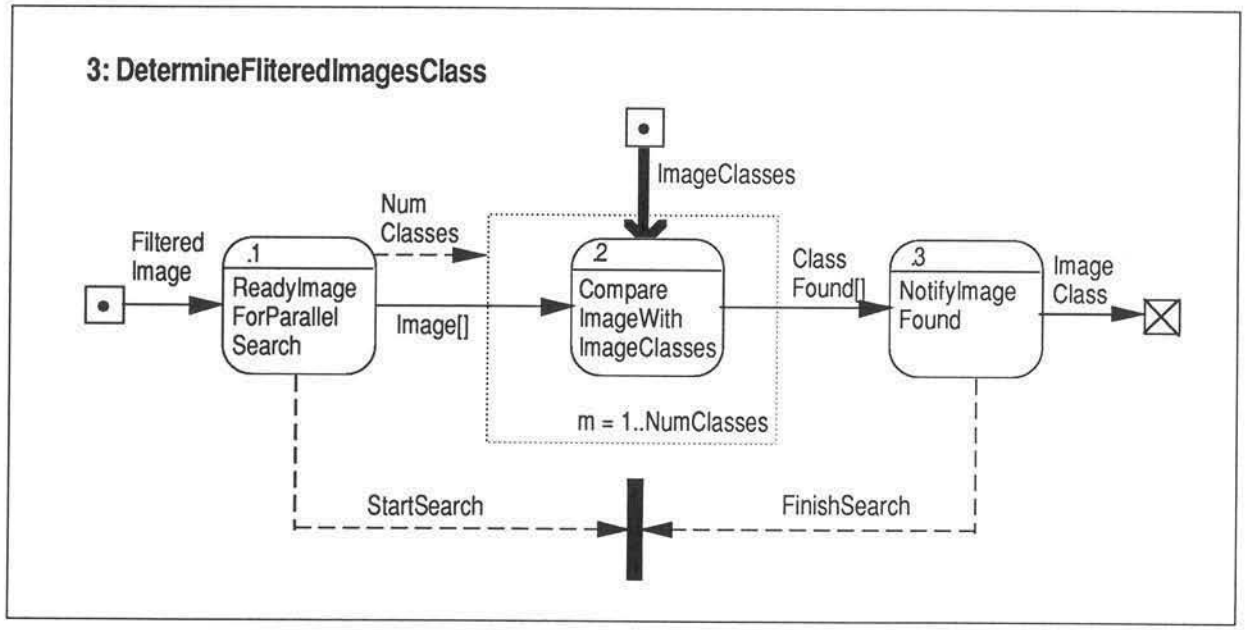


Figure 2 The refinement of the process DetermineFilteredImagesClass

```
process ReadyImageForParallelSearch;
Imports FilteredImage;
Exports Image[], NumClasses, StartSearch;

begin
  set NumClasses = FilteredImage.NumClasses;
  par for count = 1 to FilteredImage.NumClasses do
     Image[count] = FilteredImage.image;
  report(StartSearch);
end
```

Figure 3 Data Dictionary entry for the primitive process ReadyImageForParallelSearch

location in the store.

The name of a data store should reflect the data that is stored within it. In addition, every data store must have an entry in the data dictionary which defines the contents to be stored within it.

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/location becomes locked, and no other processes may access it. The store/location remains locked until either the process finishes its current execution . Any other process trying to access a store/location while it is locked is suspended until the store/location becomes available again.

Data Flows

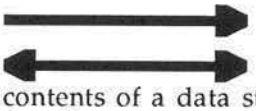
A data flow is a communication channel carrying data packets between the components in the system. The flow arcs show the names, types and directions of data that flow between the various components of a design. Three types of flow exist in PICSIL1 ; discrete, store and control. Apart from store flows, each flow in a PICSIL1 description contains an entry in the data dictionary which describes the format of the information it carries.

Discrete Flows

Discrete flows such as FilteredImage (see figure 1) are used to move data between objects (processes or external entities) within a DFD. An information transfer takes place when both the sending (exporting) object has data available and the receiving (importing) object is ready to accept the data.

Every discrete data flow must have an entry in the data dictionary which defines exactly the type of data the flow will carry. Figure 4 shows the data dictionary entry for the discrete flow InputImage and the record definition of a pixel.

Store Flows



Store flows allow processes to access the contents of a data store. In addition to data that flows between a process and data store, the store flow also represents any addressing and locking information necessary for the data transfers to take place.

Control Flows

A control flow is used to either report a change in state to a controller (see the control extensions section), or to indicate the necessary control flow associated

```
Flow discrete InputImage[1..255,1.255] of pixel

record pixel
       short int red;
       short int green;
       short int blue;
end;
```

Figure 4 Data Dictionary entry for the discrete flow InputImage and the definition of a pixel

with the dynamic allocation of replicated sections of a diagram (see sections on elements and replicators)

External Entities

External Entities represent the I/O devices that a PICSIL1 program uses. For example the external entity CameraInterface represents the I/O device associated with the camera. To ensure that the top level diagram is a complete description of the algorithm, and that new concepts are not introduced in lower level diagrams, external entities may only appear in a top level diagram.

Elements

In the specification of some types of parallel algorithm it is desirable to partition some data sets into a set of smaller blocks and process each of these blocks simultaneously. For example in the robot example, once an image has been received from the camera it is necessary to identify the image as quickly as possible. One way to do this in parallel is to compare the input image with each of the possible classes in parallel. Rather than having to show a process for each of the classes, the PICSIL1 language provides the element (shown as a dashed box in figure 2) which allows the replicated parts of a diagram to be shown once, and a statement indicating the number of replications is shown in the box. For example if ten replications were required then the statement m=1..10 would cause ten instances of the process to be created.

In a number of cases the number of replications of a process will depend on the current set of data being processed. For example in the robot example as part of the initial filtering of the image it might be possible to determine that only a subset of classes needs to be searched. In this case a control flow (i.e. the control flow named NumClasses in figure 2) from the process which sends data into the element to the element can be used to indicate that the process can dynamically decide how many replications are required for each set of data. In this case the statement m=1..NumClasses is enclosed inside the element.

The Control Extensions

Although functional specifications make it possible to illustrate the functions a system is to perform, they contain no provision for enabling or disabling a subset of those functions under particular conditions. The decisions that can be represented within a functional specification are restricted to the lowest level, with statements such as if-else and while. In the robot example shown in figure 1 when the system is first turned on we do not want the process FilterInputImage and DetermineFilteredImagesClass to start processing until the process StepupImageClasses has finished setting up the ImagesClasses data store. Another situation where higher level control is required arises in figure 2 when an image class has been found for an input image. Assuming that each of the CompareImageWithImageClasses processes takes a different length of time to execute then it is a waste of processing power to leave all unfinished processes executing once the image class has been identified.

This is a higher level of control than can be naturally specified in a functional specification, so a controller which allows this higher level control to be specified has been devised. The controller detects major changes in the systems operating mode and may turn on and off large groups of processes. It also receives information about the status of other components, both internally and externally, and transmits similar information about itself. The controller used in the PICSIL1 notation is based on extended DFD methodologies of (Hatley87) and (Ward86).

Two extra symbols are required in the PICSIL data flow diagram notation to allow control to be specified: *control flows* and *control processes*..

Control Processes

Control processes explicitly activate and deactivate data processes, and coordinate events within the system. The refinement of a control process takes the form of a state transition diagram (see figure 5 (a)) and a process activation table (figure 5(b)). *Events* are represented in the state transition diagram (STD) as labels before the slash on the arrows between the states. They are status information reported to the Control Process along *control flows* (see below) and they cause state transitions when they become true and their arrow originates from the current state (states are represented by rectangular boxes in figure 5 (a)). The *Process Activation Table* defines the status of each process in that state. The types of process status are: |■| for a deactivated process on entering a state,  for an activated continuously process and |▶| for a process

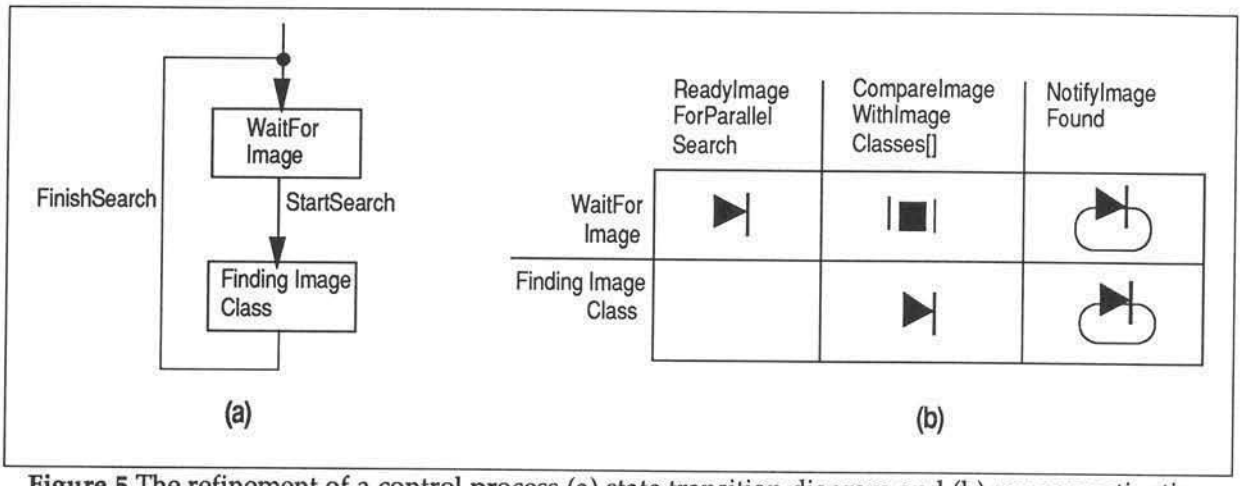
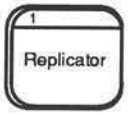


Figure 5 The refinement of a control process (a) state transition diagram and (b) process activation table

activated once on entering the state activated process. Cells with no entry indicate that there is no change in status of the process caused by the transition into that state

If a particular DFD requires no high level control, the controller is omitted and all processes in the diagram remain switched on (activated) all the time.

The Replicator



During the specification of some parallel algorithms it is necessary to be able to dynamically define the structure of a algorithm dependent on the each set of data. For example consider the specification of an algorithm which perform an arbitrarily complex function on each element in an array and then sums the results of each of the computations. The programmer of this algorithm may wish to represent the overall structure using a tree as shown in figure 6. If the size of the array can vary between sets of data, the number of leaf nodes and hence the depth of the tree also vary.

The replicator is provided as a means to allow the programmer to specify a structure which can adapt to each set of data. The replicator is like a non-primitive process in that it is defined using a child DFD. It may, however, contain constructs which dynamically select parts of the diagram

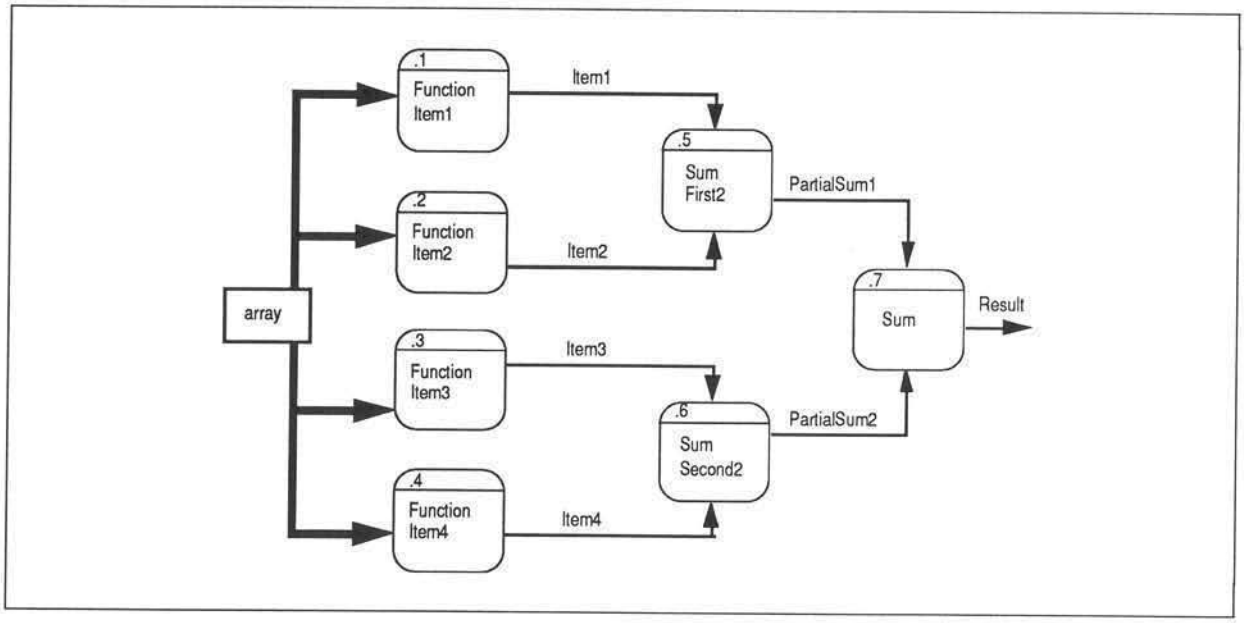


Figure 6 PICSIL1 algorithm to perform an arbitrarily complex function on an array of four numbers then sum the results

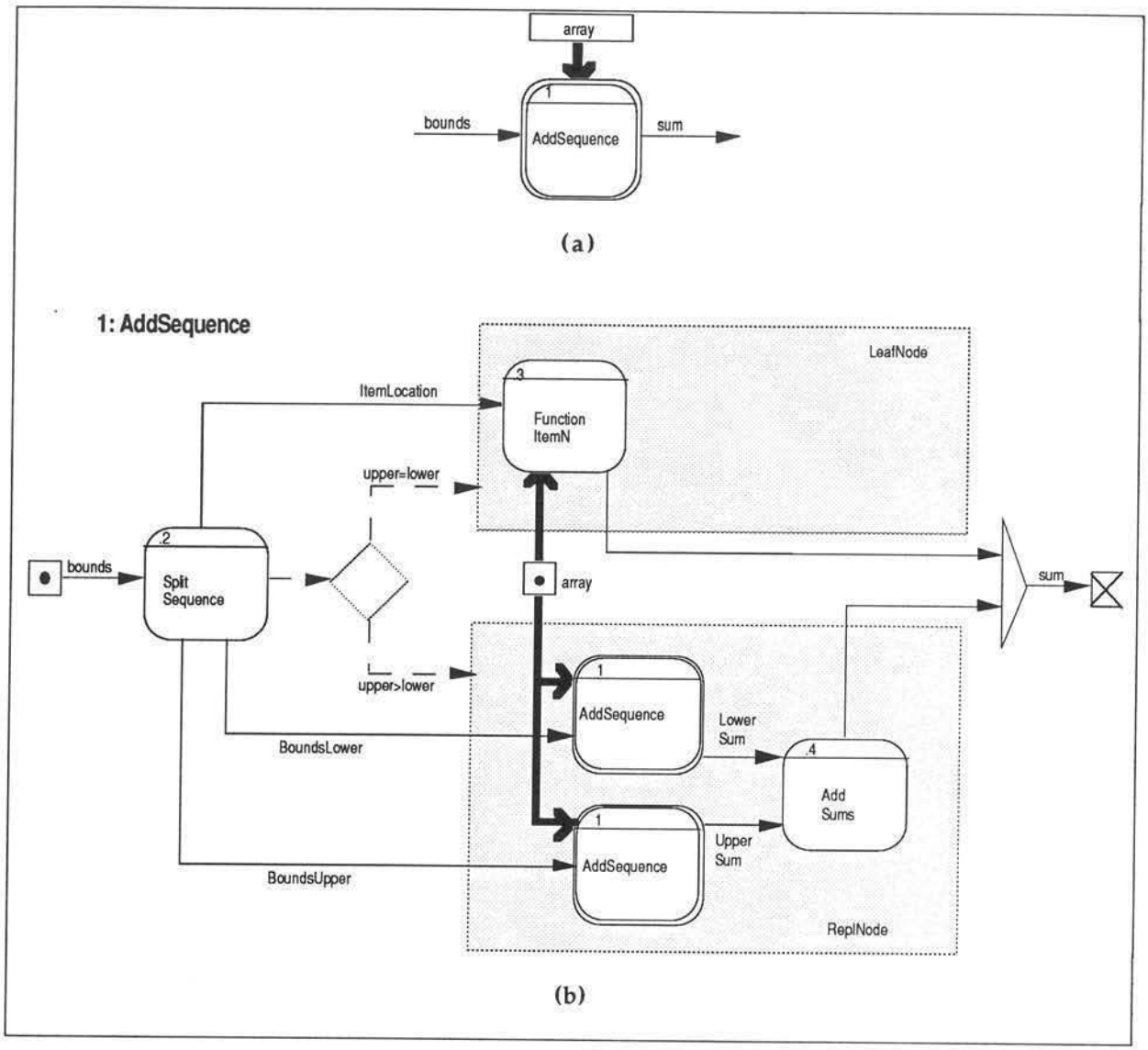


Figure 7 **(a)** Example use of a replicator and **(b)** its refinement

dependent on the data, and contain instances of itself. Figure 7(a) shows the use of a replicator to perform an arbitrarily complex function on the items in the array defined by the flow bounds (which contains sub components UpperBound and LowerBound) and output the results on the flow sum. The refinement of the AddSequence replicator is shown in figure 7(b). When a set of data on the bounds flow is received by the SplitSequence process, either the element LeafNode or the element ReplNode is instantiated. If the UpperBound is greater than the LowerBound the ReplNode element is selected causing two more instances of the AddSequence child diagram to be created. The new sets of bounds computed in the SplitSequence process are then sent to their respective child diagrams. The results returned from the child diagrams are then added by the AddSums process and the result being returned to the parent diagram.

When the upperbound is equal to the LowerBound the SplitSequence process will select the LeafNode element. This causes the function to be performed on the element in the array given by data on the flow ItemLocation. The result being passed back to the parent diagram.

Conclusions

We have described PICSIL1, a visual Parallel programming language which provides a more natural representation of parallel algorithms than conventional textual languages. Using PICSIL1 the overall structure of a parallel Algorithm is represented in a graphical notation based on Data Flow Diagrams. The more detailed aspect of the algorithm are however

described using a text based language. A number of example systems represented using PICSIL1 have shown that the language contains a useful mix of graphical and textual constructs, providing a more natural representation of parallel algorithms.

To support programming using PICSIL1 it is necessary to create an environment to support the capture, compilation and debugging of a parallel algorithm.

Work is proceeding on the development of an editor to capture PICSIL1 designs.

Once the editor has been completed the development of a set of tools to support the compilation of PICSIL1 programs to one or more parallel architectures is planned. Initially it is expected that the PICSIL1 programmer will be required to guide the compilation process so that efficient code to run on the desired architecture can be produced. The ultimate goal however is to have the compilation process completely automated.

Due to the large amounts of data generated and the distributed nature of parallel programs debugging them can be extremely difficult. It has been shown, however, that the use of graphics, colour and sound can effectively portray the execution of a program and allow problems and bottle necks to be identified more easily than with traditional debugging techniques.(Zabala93). The development of tools to support debugging are also planned in the future.

References

Beguelin, A., Dongarra, J., Geist, A., and Sunderam, V. (1993): Visualization and Debugging in a Heterogenous Environment, *IEEE Computer* **26**, 6 , pp. 88-95.

BROWNE, J.C., AZAM, M., AND SOBEK, S. (1989): CODE: A Unified approach to parallel programming, *IEEE Software* , pp. 10 - 17.

COX, P.T., GILES, F.R., AND PIETRZYKOWSKI, T. (1990): Prograph: a step towards liberating programming from textual conditioning. *In Proc. of IEEE Symposium of Visual Languages* , pp. 150-156

DEMARCO, T. (1978): *Structured Analysis and System Specification,* Prentice-Hall.

DILLINGER, T.E. (1988): *VLSI Engineering,* Prentice Hall.

GRUNDY, J.C. AND HOSKING, J.G. (1993): Constructing Multi-View Editing Environments Using MViews. *To appear in Proc. of IEEE Symposium on Visual Languages,*

HATLEY, D.J. AND PIRBHAI, I.A. (1987): *Strategies for Real-Time System Specification,* Dorset House .

KU, D. AND DE MICHELI, G. (1990): "HardwareC - A Language for Hardware Design Version 2.0", Computer Systems Laboratory, Stanford University , no. CSL-TR-90-419 .

MYERS, B.A. (1990): Taxonomies of Visual Programming and Program Visualisation, *Journal of Visual Languages and Computing* **1**, 1 , pp. 97 - 123.

PANCAKE, C.M. AND BERGMARK, D. (1990): Do Parallel Languages Respond to the Needs of Scientific Programmers, *IEEE Computer* **23**, 12 , pp. 13-23.

PEARSON, M.W. (1992): *PICSIL: Design and Synthesis of Digital ICs from Data Flow Diagrams,* Ph.D. dissertation, Department of Computer Science, Massey University, Palmerston North, New Zealand.

SOCHA, D., BAILEY, M.L., AND NOTKI , D. (1989): VOYEUR: Graphical Views of Parallel Programs, *SIGPLAN Notices* **24**, 1 , pp. 206-215.

SUHLER, P.A., BISWAS, J., KORNER, K.M., AND BROWNE, J.C. (1990): TDFL: A Task-Level Dataflow Language, *Journal of Parallel and Distributed Computing* **9**, 2 , pp. 103-115.

THOMAS, I. (1993): *The Circuit Model of Parallel Programming,* Master's thesis, Department of Computer Science, Auckland University, Auckland, New Zealand, .

TOOMAJANIAN, G. (1992): *Graphical Behaviour Capture to VHDL* , Personal Communication .

TSE, T.H. AND PONG, L. (1991): An Examination of Requirements Specification Languages, *The Computer Journal* **34**, 2 , pp. 143 - 152.

WARD, P.T. (1986): The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing, *IEEE Transactions on Software Engineering* **SE12**, 2 , pp. 198 - 210.

ZABALA, E. AND TAYLOR, R. (1993): "Process and Processor Interaction: Architecture Independent Visualisation Schema", Computer Systems Engineering Group, Dept. of Electronics, University of York , Technical Report , York Y01 5DD, United Kingdom.