A Transformational Algebra for Communicating Sequential Process Data-Flow Diagram Statements in Classes of Parallel Forthlet Objects for Design, Automated Place and Route, and Application Development on the SEAforth Architecture.

The model of Communicating Sequential Processes has been adapted into various programming languages to allow simplified programming and formal study of parallel systems. Forthlets are parallel programming objects for the hardware CSP implementation in the SEAforth architecture. In the CSP approach programs are modeled using data-flow diagrams. The diagrams formally describe the flow of data through parallel programs in template statements in the source code and are instantiated with properties based on position in an array when the Forthlets are placed and routed in compilation. Instantiated data flow statements in multiple Forthlets running in parallel on SEAforth architecture can using a formal algebra be used to perform design verification, automatic test code and application code template generation, and assist with application debugging. This paper will introduce the application of a formal algebra and the translation of CSP data-flow diagrams to SEAforth source test code modules used in design, debugging, and for automatic generation of test code and application code templates.

A Word of Background

In the Forth programming language everything is a word. The language uses words at a semantic level to describe programs which on SEAforth architecture are executing the Forth primitives directly in hardware as the processor's instruction set. For decades Forth was implemented by first building a layer of a Forth Virtual Machine on top of a machines native instruction set. The processor instruction set level is the lowest level of code and by starting with the instruction set already at the level of the Forth programming language primitives much of the complexity in software implementations of Forth for thirty years falls away from the software. Semantic languages are extended by adding new words with new meanings to the language. By starting at a higher level extending the language to support parallel programming objects programming becomes easier.

Parallelism has appeared in many forms and most have attempted to hide it from the programmers and the software for back-wards compatibility issues and to allow the programmers who have been thinking of computing as a sequential enterprise to continue to deal with their code which characterizes computing as sequential. Wider data busses, wider memory busses, cached architectures, pipelined architectures, multi-threaded architectures, Very Long Instruction Word architectures and branch prediction techniques are all forms of parallelism used to speed computers while mostly running old sequential code faster.

True parallel programming existed at the super-computer level for a long time and in some niche markets. The PC industry kept things running sequentially until the Internet connectivity of machines brought access to large networks of PC running in parallel to every desktop. In the twenty-first century very large scale integration and software development had reached the point that even with the large and heavily pipelined and cached processor architectures it became possible to deliver multi-core products designed to deliver more power through parallel execution on multiple nodes.

The Forth language has thrived on a different class of computing hardware used in embedded applications. In the embedded arena the more heavily factored Forth code tended to be much smaller (much cheaper on embedded systems) and much simpler (much cheaper to develop, debug and maintain on embedded systems) than code produced using programming languages designed and evolved as the hardware/software architecture for desktop computing. More heavily factored code was characterized by smaller routines to test and verify and a shorter write-test-reuse cycle. With code factored into words the small amount of code in individual words encourages and supports interactive testing in development and gives the developer insight into potential application problems that may not have been obvious in the initial design phase.

A statistical analysis of the Forth code in the inventor of Forth's desktop VLSI CAD software shows that the most commonly sized words are those that contain seven or less other Forth words. The factoring of code into such small pieces for fast interactive development, easy maintenance, and natural code compression are all characteristic of the use of the Forth language and techniques well suited to embedded system development.

Many programming languages have been adapted to support describing and generating code for architectures that achieve performance through the use of parallelism in the form of deep pipelines and caches and rich instruction sets. These techniques introduce a level of complexity that led developers to abandon low level code decades ago and rely on complex optimizing compilers. The complex optimizing compilers themselves were complex. The general purpose Operating Systems used to support the compilers and applications were large and complex. The optimizing compilers generate code that uses techniques like code inlining to satisfy the pipelined execution units which increases program and system size and cost.

Parallel programming in these environments usually involves another layer above the rest, often large and complex. The environments and the code produced for them are not well suited to attacking the embedded system problem that seeks to bring cost and power consumption to zero while providing just the computing power required in an embedded application.

Today multi-core designs for parallel programming have gone beyond the DSP designs and soft core to both the desktop and down to the level of embedded processing. The great fear of management, sequential processing programmers, and the public in general is that with the promise of parallel programming comes the challenge of writing and debugging parallel programs which for the most part has been the domain of the super-computing crowd. The software techniques used on the largest and most expensive super-computing work are not suitable for embedded system development. The fear of the transition to the time when programmers will have to deal with parallel programming has been a common news story into the twenty-first century.

Communicating Sequential Processes, CSP

In 1967 C. A. R. Hoare proposed a system called Communicating Sequential Processes, or CSP. Hoare's Communicating Sequential Processes provided a framework for structuring parallel programs based on data-flow synchronization in programs and was described in the Occam programming language and implemented on the Inmos Transputer hardware in the 1980s. The concept that parallel programs could be described and factored into communicating sequential processes was easily accessible to programmers familiar with sequential processes and with communication between computer programs. The technique of using synchronized communication services to synchronize distributed parts of a parallel system while processing a flow of data removed much of the complexity found in other parallel programming approaches.

Hoare's CSP were adapted into other programming languages. In 1993 Dr. Montvelishsky published a simple portable multitasking and multiprocessing extension to the Forth language implementing CSP as Parallel Channels in Forth. In a few lines of code he demonstrated a simple approach for designing and implementing parallel programming on Forth systems and for adapting existing multitasking programs to multiprocessing programs with minimal changes. Dr. Montvelishsky has demonstrated the use of CSP Data-flow diagrams in parallel programming of applications on the SEAforth architecture at IntellaSys and following is a description of some of the work on which I have participated.

In these systems a processor with nothing else to do sleeps until it gets a message from another task on another processor. When the message is exchanged the two parallel components synchronize. In the SEAforth implementation processors sleep waiting for program or data messages to arrive and when sleeping consume very little power. When a message arrives they awaken from sleep to do the appropriate thing and components synchronize. Sending or receiving a message simply requires writing or reading a port address.

Synchronous Systems made with Asynchronous Architecture

The SEAforth processors are implemented with asynchronous circuit design. This means that there is no master logic clock being distributed to circuits and used to gate all internal operations. This is done to reduce execution overhead, cost, power consumption and generated noise. Having huge clock circuits with antenna running all over chips produces a large amount of power consumption and noise on and off chip.

Systems with parallel components handling individual events asynchronously are said to be synchronous systems. They require synchronization of components at the system level. The fact that the hardware is asynchronous and has no master clock on SEAforth but that the systems are synchronous parallel systems has caused some confusion.

Full Custom Hardware and Embedded Distributed Forth OS

The full custom hardware implementation of the low level of the Forth language reduces the size and cost of the processor and the size and cost of the code the processors execute. The instruction set has been researched and adjusted for decades to achieve a sort of natural code compression that is well suited to the goal of reducing size, cost and power consumption in embedded systems. The hardware implementation of the low level of the Forth language also improves the performance of programs reduced in size by being structured in Forth.

The full custom hardware implementation of the low level mechanism of synchronizing wake-sleep communication ports between processors and between clusters reduces the size and cost of the processor design and the size and cost of the code the processors need to implement CSP. The operation of these ports has been optimized for the goals of embedded system use and the full custom hardware implementation of CSP ports also improves the performance of the system.

On the SEAforth architecture the synchronized communication ports are labeled as Right Down Left and Up or R--- -D-- --L- and ---U. They can be read or written by a processor on either side of the port. The processor that does so first will go into a sleep state until it is awakened by a read or write at the other end of the port as is appropriate.

For greater efficiency the SEAforth architecture supports addressing more than one port at the same time. This is done using addresses that correspond to the labels RDLU or R-L- and allows for further optimizations that will not be discussed in this review of the use of basic data-flow diagrams in CSP.

Some parallel programming extensions to languages are built on top of conventional general-purpose OS services and are suited to designs that embed such an operating system on every node in the network. Embedding Linux with real-time extensions in parallel processing systems such as Beowulf is one solution that requires that each node be large enough to support a general-purpose conventional OS. Embedded Forth systems traditionally contained specific Operating System services for specific applications and integrated them into same Forth dictionary as other Forth and application words.

For parallel programming with CSP objects the most important services are synchronized communication channels. When implemented in hardware instead of a virtual layer in software the amount of OS service that must be embedded on any give node is dramatically reduced. The nature of embedded systems and of parallel programming allows Operating System services to be distributed as needed and in parallel.

Parallel Programming Using CSP and Data-Flow Diagrams

A popular model for parallel programming uses parallel Objects that have an Input function and an Output function. We can express that program reads from its IN function with a lower case “i” and that it writes to its IN function with an upper case “I” as we can express a read from the OUT function with a lower case “o” and that it writes to its OUT function with an upper case “O”. We can express that program reads from IN and writes to OUT with the diagram

```
iO
```

In the SEAforth architecture for a processing node to read from a neighbor, do some processing, write the result to a different neighbor some combination of one or more of the three addressing registers P, A, and B is used. A program that reads from a port designated as input by the value IN and that writes to a port designated as output by the value OUT can be written using the A and B registers as pointers. After the A register has been set to point to the input port and the B register set to point to the output port the following code may be executed:

```
@a !b
```

If we assume the A register is out IN pointer and that the B register is the OUT pointer the data-flow diagram for that code is:

```
iO
```

If the two input and output port neighbors are both asleep waiting the input node will awaken, complete its write, and continue then the output node will awaken and complete its read and continue. If the neighbors are not already waiting to synchronize on the message exchange the node executing the above code will sleep until the neighbor completes the transaction.

To create a loop that does this forever one may write:

```
(iO)
```

The equivalent machineForth code using the same assumption as above is:

```
begin @a  !b  again
```

To create real application source some code that performs the computation is added as needed to the data-flow code template. The data-flow in a diagram is not changed by computation on a node. Diagrams must also express conditional data-flow behavior on the node. Expressions and numbers evaluate to designate loop counts. The symbol ?(is used to specify conditional data flow testing for zero.

The Use of Data-flow Diagrams in Parallel Forthlet Code

The compiling program passes arguments for the input data port and output data port and in this case for a loop counter. The parameters are passed by the compiling program by setting the values IN, OUT, and N.
Data-flow diagrams represent classes of parallel objects. A two-dimensional array distribute, compute, and gather results diagram is not effected by the compute section. A two-dimensional array distribute and gather class object can be constructed using one-dimensional array distribute and gather class object. A one-dimensional array of N*M data elements can be distributed to a linear group of nodes by loading N*M data elements at one end of the line and using M elements on each of N nodes for a computation, and gathering results back. One case of this class collects a single element as the result of N nodes computations on M elements each.

The data-flow diagram for one of the modules in a machine vision application is used here as an example of a linear group distribution and gather. A template for the code is generated that matches a description of the flow of data on a group of nodes. In this group the data-flow for a sequence of connected nodes is numbered as nodes 0 to N where N designates the node number in the group. The value N will be set on each node instantiated in this group. M is a value that describes the size in data elements of a packet of data being processed on each of node 0 to N nodes. In this example some of the image recognition program nodes perform the following one-dimensional distribute and gather class data-flow:

```
(NM*(iO)M(i)N?(o)I)
```

IN was set to Down and OUT is set to Up when the template is instantiated on node N6 in this example. A two-dimensional array is parallelized as a group of groups of one-dimensional array groups. N6 the first node in a linear group and the instantiated diagram for N6 is:

```
(NM*(dU)M(d)N?(u)D)
```

The instantiated diagram for this class of Forthlet on this node says:

Repeat the following forever; for N*M times read the Down port and Write the Up port, then for M times read the Down port, then if N is not zero read the Up port, and always Write the Down port.

(NM*(iO)M(i)N?(o)I) translates to the following standard code template:

```
IN # a!
OUT # b!
begin
  [ N M * ] [IF] [ N M * 1- ] # for @a+  !b unext [THEN]
  [ M ] [IF] [ M 1- ] # for @a+ unext [THEN]
  [ N ] [IF] @b [THEN]
!a+ again
```

The code statements in upper case and inside brackets are being computed at compile time and are Standard Forth. The code statements in lower case and not bracketed are native SEAforth code to be compiled on each of set of nodes in a group.

This data-flow test code performs the same data-flow described in the diagram of the module of the image template matching in the machine vision application and was used as both a template for the real application source code and as test code used as a replacement for that code for debugging purposes. The application was debugged module by module to confirm that the data flow was correct then the empty data-flow template code modules were replaced by the code that did the real processing on the data flowing through the application. The Forth methodology of factoring into small modules that can be individually tested and reused was applied to parallel programming and use of CSP in Forth. Data flow can be modeled and debugged separately from the details of procedural processing of the data.

If a group of six nodes performs the data-diagram (NM*(iO)M(i)N?(o)I) with N going from 5 to 0 the data flow of that group can be simulated on the first node with a simplified version of the diagram where output to OUT and input from OUT can be filtered out. This leaves the input only diagram of:

(NM*(i)M(i)I)

Which further simplifies to:

(N1+M*(i)I)

This translates to the following standard code template:

```
IN # a!
OUT # b!
begin
  [ N 1 + M * 1- ] # for @a+ unext [THEN]
!a+ again
```

This test program will have same input and output flow as the first node in the linear array distribute and collect class group. If flow to and from this node is proven correct the full templace can be instantiated on all nodes in the group and will have identical flow into and out of the group as this simple test routine.

In the SEAforth architecture the terms North, South, East, and West refer to global directions as one would find on a map while Right, Left, Up and Down are local terms such that North and Up are not always the same. The important feature of this addressing is that ports have the same address on each end. So if one neighbor writes to the Right port the Right neighbor must read from the Right port. Both neighbors see the shared port as a Right port. A property of Data-flow on connected nodes is that they must have complementary Data-flow diagrams when their diagrams are filtered for their shared port. For a node with a diagram of:

(N(iO)M(i)oI)

If it has Right as its IN connection and if the diagram is filtered for Right direction only one gets:

(N(r)M(r)R)

This can also be expressed as:

(NM+(r)R)

This requires that the node to the Right and communicating on the other end of a Right port must have complementary data-flow. When its data-flow diagram is filtered for right and must have:

(NM+(R)r)

Data-flow expressed in these formal terms can be proven to be complete and free of deadlock conditions. Through the use of filters and merges some data-flow diagrams can be generated. Data-flow diagrams in turn can generate both flow test code and code templates for application code.

When that Forthlet machine vision template recognition module was placed on node 06 on a SEAforth24 IN was set to Down and OUT was set to Up. M was set to 16 and N was set to 4 on node 06 as part of a five node group with N going from 4 to 0. Once assigned to the group and positioned in an array the connecting IN and OUT ports assignments are instantiated. Since declaration need only be done in the diagram the placement and routing of the group can be done manually or automatically.

Data-flow diagrams can be expanded to include the names of library routines and Forth code that specify the procedural function of the Forthlet. Thus Forth code can be used with the formal algebra and inserted into a diagram using postfix notation and spaces as needed as in:

(NM*(iO)M(i) vmatch N?(o+)I)

Instantiating that statement on individual members of a group of connected nodes results in complete code generation for all members of such groups. At the same time formally specifies and documents the data-flow in the distribution and collection of data in that group.

Generated code:

```
INCLUDE vmatch.f
```

```
IN # a!
OUT # b!
begin
  [ N M * ] [IF] [ N M * 1- ] # for @a+  !b unext [THEN]
  [ M ] [IF] [ M 1- ] # for @a+ unext [THEN]
  vmatch
  [ N ] [IF] @b . + [THEN]
!a+ again
```

Other Topics

In the SEAforth architecture computers can read and write to ports using the Program-counter, A or B registers. When reading or writing with the P register the word read or written will be treated as an instruction word to be executed or will be treated as data in the T register on the top of the data stack. When reading or writing memory or a port using either of the A or B registers the word read or written will be treated as data going to or coming from the T register at the top of the push-down on-chip data register stack.

Programs may be treated as data at any time and when data is being written to a port it may be code being executed or data being loaded on the other side. Port contents can be both code and data at the same time. Programs can duplicate and modify themselves and do so without the penalties imposed on systems that employ parallelism in the form of pipelines and caches and other mechanisms that tend to make code larger anyway. Ports can be written in multiple directions or read in multiple directions at the same time. In the SEAforth architecture these features are used by the compiler, library routines, and programmers to produce smaller, faster, or more efficient code.

Jeff Fox, Senior Principal Engineer, Systems Architect, IntellaSys
12/19/07