Abstract of Talk for UK Asynchronous Forum, 1996

Multi-View Design of Asynchronous Micropipeline Systems using Rainbow*

Howard Barringer, Donal Fellows, Graham Gough, Pete Jinks, Brian Marsden, Alan Williams†
Department of Computer Science, University of Manchester, UK

email: `rainbow@cs.man.ac.uk`

URL: `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`

1 Introduction

Sutherland's Micropipeline method [1] for designing asynchronous hardware systems has been used successfully for the development of commercial-scale devices, including asynchronous versions of the ARM processor in the AMULET Project [2]. However, the perceived benefits of asynchronous design — the production of high-speed and low-power systems — have so far been difficult to realise due to the relative lack of design representations and associated tools specialised to asynchronous micropipeline hardware. Certain design methods, such as Philip's Tangram system [3], are equipped with description languages. However, specialised design languages have not yet emerged for micropipelines, especially for higher-level abstract descriptions. In addition to suitable languages, designers also require accompanying simulation tools for early (high-level) experimentation, without having to complete much detailed gate-level implementation. Analysis tools are also needed for detecting deadlocks, for estimating performance, or checking design equivalence.

Clearly, such formal analysis tools need to be based on proper foundations, i.e. it is essential that the design language has a usable formal semantics. Current application of formally based models usually is limited to capturing the request/acknowledge control signalling between micropipeline components. This has been used with some success for building CCS models of the AMULET1 processor [4].

1.1 The Rainbow Approach

To address these problems, we are currently developing *Rainbow*, a wide spectrum hardware design language, for multi-view description of micropipeline designs. This enables a single design to be described in a variety of ways, using the style most appropriate for each component in the design. It reflects the way engineers construct designs, for example simultaneously using a mixture of structural, algorithmic and tabular description styles, presented in both textual and schematic form. Rainbow has been developed in close collaboration with the AMULET Group at the University of Manchester, so that it will closely match the needs of practising hardware designers. The formal semantics of Rainbow is defined via a translation to an underlying process term language called APA.

*This is an abstract of a full paper that has been submitted to Async'97

†The University of Manchester acknowledges support from the Engineering and Physical Sciences Research Council via research grant GR/K42073.

2 Rainbow

The Rainbow framework consists of a suite of interworking sub-languages. This includes a dataflow-style language based on micropipeline communication for structural descriptions, and a control-flow style language for algorithmic descriptions, using Ada-like rendezvous communication. Ultimately, the framework will contain at least the following sub-languages:

***Red*:** to support behavioural/specification descriptions using, for example, temporal logic or stream transformers [5].

***Yellow*:** to describe designs using a control flow style with Ada-like rendezvous communication [6, 7] between components.

***Green*:** to support hierarchical structural descriptions based on micropipeline primitives. It has schematic and textual versions.

***Blue*:** to operate at a level below Green, exposing the handshaking control, similar to the CCS models of AMULET described in [4].

Figure 1 shows the Rainbow framework: the user interacts with the system by describing a design using

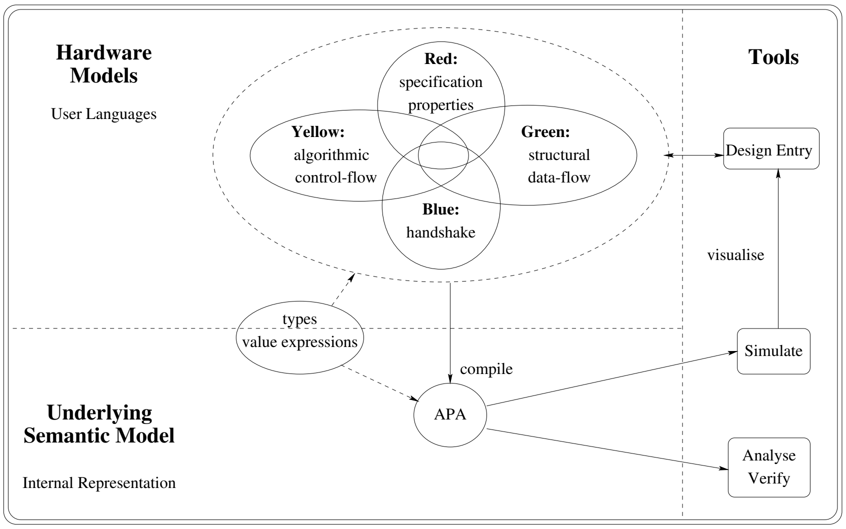

Figure 1: The Rainbow System

a combination of the sub-languages, indicated by the intersecting ovals. A single uniform semantics for Rainbow is defined by a translation to APA — a common set of data expressions is embedded into both. The internal APA representation will provide the basis for formal analysis and simulation tools, the results of which can be visualised via annotation of the design descriptions.

2.1 Green

Green is a static dataflow language [8], using explicit buffering between elements, that models micropipeline communication, i.e. before an element can fire all of the required input values for a given (stateless) element must be present, and the outputs must be ready to accept the values generated. Buffers are used to introduce state and provide decoupling between inputs and outputs and dataflow control elements provide ways to conditionally merge and split data streams. Elements for pipeline merge and split support flow control. A more generalised element type is the `table`, which combines the functionality of functions and flow control elements. Elements can be connected together in a pipeline fashion, or can be placed in parallel so that each element can evolve independently. Complex systems are constructed hierarchically by instantiating sub-networks.

Green networks can be described textually or as schematics (directed graphs); the latter gives the opportunity for design animation during simulation.

2.2 Yellow

In contrast to the static structural description style offered by Green, Yellow allows *evolving* descriptions to be constructed, using a control-flow style, similar in many respects to CSP [9] or Ada [7]. The (explicit) call/accept communication primitives used are adaptations of the Ada rendezvous. There are standard imperative sequence and register assignment operators. A choice construct provides the means for switching control conditionally between different code fragments. A conditional loop statement provides iteration. Other Rainbow components, such as Green procedures, can be instantiated in Yellow descriptions.

2.3 APA

The formal semantics of Yellow and Green are defined by a specialised process term language called *APA* (Asynchronous Process Algebra), which also operates at the micropipeline level: basic micropipeline components and combinators are modelled by atomic components and combinators in APA. By translating the Yellow and Green components into APA, the interfaces between these are well-defined, enabling the complete design to be simulated and analysed.

APA has some similarities to more familiar process algebras, such as CCS [10]. Its semantics is defined operationally with a set of SOS-style transition rules [11]. However, the composition operators have been designed to support bundled-data micropipeline communication — the communication primitives involved use an Ada-like rendezvous, and their semantics resembles the Ada semantics presented in [6]. APA also supports value-passing and has richly-structured actions. These features lead to a compact semantic representation of Rainbow designs, providing a uniform basis for simulation and formal analysis.

3 Example: a Simple Processor

In order to illustrate the mixed-view style of design description supported by Rainbow, we present a simple processor design, SMPU, using interacting Green and Yellow components. Figure 2 shows the basic top-level block structure of SMPU described in Visual Green, a style similar to that used in informal descriptions given in design documentation, such as for AMULET1 [2].

The functionality of each block is then described in the most appropriate style. For example, `decode` provides the basic control of SMPU and sequences the fetch-decode/execute cycle for each instruction; this can therefore be described initially using Yellow.

```
yellow decode(input instruction: instr;
              output memIFC: memctrl;     output regIFC: regctrl;
              output aluIFC: aluctrl) = {
  loop TRUE -> {
    reg i: instr end

    //-------------------------------------------------------------
    // Fetch (pc is at register 0):
    par
           call memIFC!is_instr
      || { call regIFC!(ARG_A,0) ; call regIFC!(WB,0) }
```

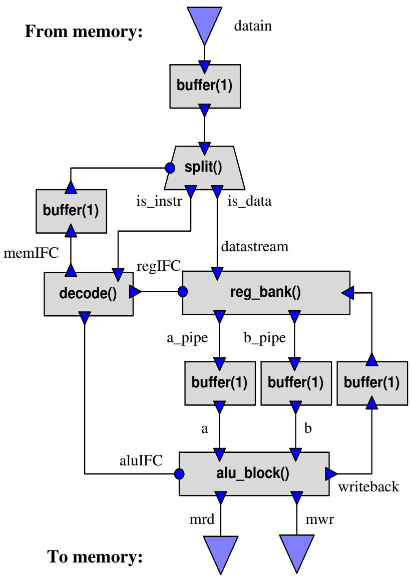

Figure 2: Visual Green SMPU

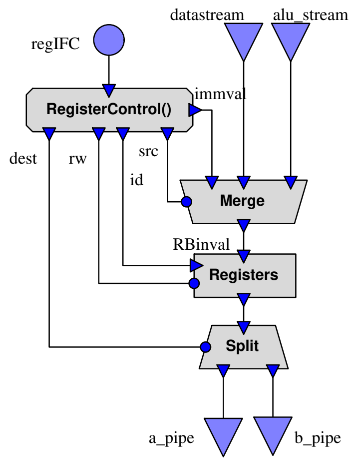

Figure 3: Visual Green Register Bank

```
      ||   call aluIFC!FETCHINC
      ||   accept instruction?ins do i:=ins end
    end_par;
    //----------------------------------------------------------
    // Execute: given in appendix
    //----------------------------------------------------------
  } end;
} end
```

The ALU is described using the `table` construct of Green, which offers a concise functional/control-flow description:

```
table alu_block (input  aluIFC: aluctrl;  input  a,b: val;
                 output mrd: memread;     output mwr: memwrite;
                 output writeback: val) = {
    FETCHINC, a, - => a,   -,     a+1      //a is pc value
 or FROMMEM,  a, b => a+b, -,     -        //a is base addr, b is offset
 or TOMEM,    a, b => -,   (a,b), -        //a is addr, b is value
 or ADD,      a, b => -,   -,     a+b      // -- Binary ALU operations
 or SUB,      a, b => -,   -,     a-b
 or AND,      a, b => -,   -,     a/\b
 or CMP,      a, b => -,   -,     a>b      // -- Unary ALU operations
 or MOV,      a, - => -,   -,     a
 or NOT,      a, - => -,   -,     not a
} end
```

A row in the table is executed when the values on its input can be matched with the input patterns (to the left of the '=>'), causing the expressions (to the right of the '=>') to be output. The entries may contain literals (e.g. `FETCHINC`) or variables (e.g. `a`), which match either with one or with any value on the input. An 'empty' entry '-' means that an input is not consumed or an output is not produced.

4 Simulation

To illustrate how Green can be used to simulate designs, Figure 4 shows a Green schematic that computes the sequence of Fibonacci numbers. In the annotated design, buffer *C* has just received the next

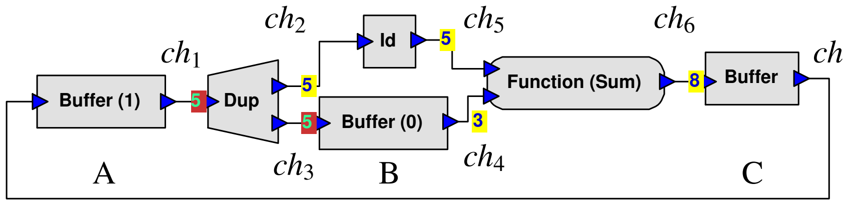

Figure 4: Visual Green Fibonacci Calculator Design

Fibonacci number ($8 = 3 + 5$), buffer B is supplying the old Fibonacci number (3), and buffer A supplies the current number (5). The channel labels with a light background indicate values that have successfully been transferred, and the labels with a dark background show values that are ready to be transferred but are currently blocked. Therefore, only one of the values from the Dup element has so far been consumed, with the second value 5 waiting to be written into buffer B. When this has occurred, then buffer A will be ready to receive the new Fibonacci number (8) from C, and the next number in the sequence can then be calculated.

5 Summary

We have described the Rainbow hardware design environment that supports the development of user-level, mixed-view descriptions of asynchronous micropipeline systems. An underlying formal semantics provides the means for defining the interworking between design components described in the different Rainbow sub-languages. Prototype design and simulation tools for Rainbow are being developed, and the formal semantic foundation gives the basis for development of formal analysis tools.

6 References

[1] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6), June 1989.

[2] N. Paver. *The Design and Implementation of an Asynchronous Microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1994.

[3] K. van Berkel. *Handshake Circuits: an Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[4] Ying Liu. *AMULET1: Specification and Verification in CCS*. PhD thesis, Department of Computer Science, University of Calgary, September 1995.

[5] Manfred Broy, Frank Dederich, Claus Dendorfer, and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-19202, Institut für Informatik, Technische Universität München, 1992.

[6] Wei Li. An Operational Semantics of Tasking and Exception Handling in Ada. Technical Report CSR-99-82, University of Edinburgh, January 1982.

[7] I. Sommerville and R. Morrison. *Software Development with Ada*. Addison-Wesley, 1987.

[8] J. Herath, Y. Yamaguchi, K. Toda, R. Mattingley, N. Saito, and T. Yuba. Comparison of Dataflow Computing Models. In John A. Sharp, editor, *Data Flow Computing: Theory and Practice*, pages 16–34. Ablex, 1992.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[10] R. Milner. *Communication and Concurrency*. Prentice Hall, Hemel Hempstead, Herts, England, 1989.

[11] M. Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.