Multi-View Design of Asynchronous Micropipeline Systems using Rainbow

H. Barringer, D. Fellows, G. Gough, P. Jinks and A. Williams
Department of Computer Science
University of Manchester, Manchester M13 9PL, UK
phone: +44 (0) 161-275-6248 fax: +44 (0) 161-275-6211
email: `rainbow@cs.man.ac.uk`

Abstract

We present the Rainbow hardware design environment for asynchronous micropipeline systems. Rainbow contains a suite of user-level sub-languages for multi-view design description. An underlying formal model defines the behaviour of Rainbow components and thus provides a means of determining their combined behaviour. A simple processor design is used to illustrate the integrated description style provided by Rainbow and a smaller example shows how the components interwork at the semantic level.

Keywords

Multi-view design, micropipelines, asynchronous systems, formal methods.

1 INTRODUCTION

The Micropipeline method (Sutherland 1989) 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 (Paver 1994). 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 are equipped with description languages, such as Philips' Tangram system (van Berkel 1992), or the synthesis approach adopted in (Martin 1990). However specialised design languages have not yet emerged for micropipelines, especially for higher-level abstract descriptions. Designers also require simulation tools for early (high-level) experimentation, and analysis tools for detecting deadlocks, estimating performance, or for checking design equivalence. Of course, such analysis tools must be based on proper foundations, i.e. it is essential that the design language has a usable formal semantics. Current formally based approaches have usually been limited to capturing the request/acknowledge control signalling between micropipeline components, e.g. the CCS models of the AMULET1 processor (Liu 1995). Others have developed special process algebras for delay-insensitive systems (Josephs & Udding 1990), or have used Petri Net models (Yakovlev et al. 1995).

1.1 The Rainbow Approach

Our approach has been to develop *Rainbow*, a suite of user-oriented, multi-view, description languages specialised for modelling micropipeline designs. A single design can be described in a variety of styles, using the language most appropriate for each design component. The Rainbow languages have been developed in close collaboration with the AMULET Group at the University of Manchester, so that they will closely match the needs of practising hardware designers. The formal semantics of each of the Rainbow languages is defined via a translation to a common underlying process term language called APA (Asynchronous Process Algebra), which also operates at the micropipeline level: basic micropipeline components and combinators are modelled by atomic APA components and combinators. The semantics of APA is defined via SOS-style transition rules (Hennessy 1990). This provides the foundation for the development of formal analysis tools. The general approach adopted to the Rainbow semantics follows the successful line we took with developing formal analysis tools for the (synchronous) hardware description language ELLA (Barringer et al. 1996b).

In section 2, we describe Rainbow and the component sub-languages. An example of the design development style now possible is illustrated with a description of a simple processor (section 2.3). We then present an outline of the APA semantics for Rainbow (section 3) and show how this is used to provide interoperability between the different sub-languages (section 4).

2 RAINBOW

The Rainbow framework currently includes a dataflow-style language (*Green*), based on micropipeline communication and primitives, for hierarchical structural descriptions, with schematic and textual versions. It also includes a control-flow style language (*Yellow*) for algorithmic descriptions, using Ada-like rendezvous communication (Li 1982, Sommerville & Morrison 1987) between components. We are extending the framework to include a high-level language (*Red*) for behavioural/specification descriptions using, for example, temporal logic or stream transformers, and a *Blue* language to operate at a level below Green, by exposing the handshaking control, similar to the CCS models of AMULET in (Liu 1995).

2.1 Green

Green is a *static* dataflow language (Herath et al. 1992), using explicit finite buffering between elements*, that models micropipelines. Buffers are used to introduce state and provide decoupling between inputs and outputs — an empty buffer can accept an input; once this has been acknowledged then the (full) buffer can output the value, and is only ready to accept a new input when the output has been released. A stateless element can only fire when all of the required inputs are present. It then generates output values, but does not release any input channel until all its outputs have been accepted and released. For example, a Duplicate node simply copies its input to a number of outputs; sources (or sinks) generate (or absorb) streams of values. Dataflow control elements provide ways to conditionally merge and split data streams. A more generalised element is the Table, which combines functionality and flow control. Elements can be combined serially (as a pipeline) or in parallel. Refer to (Barringer et al. 1996a) for more details of Green and its editing/simulation tools.

As an example of Green, Figure 1 shows part of the simulation of a schematic computing sequences of Fibonacci numbers (an APA translation will be given in section 3). Buffer C has just received the next Fibonacci number ($8 = 3+5$),

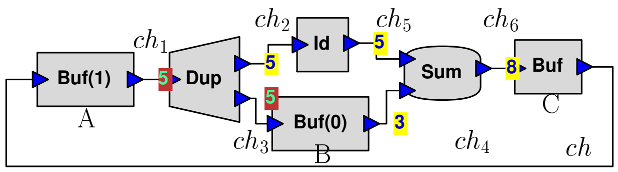

Figure 1 Fibonacci Calculator Design

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.

*Note that wires are delay-less.

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 (Hoare 1985) or Ada. The (explicit) call/accept communication primitives used are adaptations of the Ada rendezvous. An input statement '`accept inps do P end`' starts execution of `P` when all of the inputs `inps` are ready. When `P` terminates then the values on `inps` are acknowledged, and the processes supplying the inputs are then freed to continue. An output statement '`call outs`' makes the outputs `outs` available, and then is suspended until all of the outputs have been consumed by matching inputs in `accept` statements. Yellow also includes the standard imperative sequence $(P\ ;\ Q)$ and assignment operators $(r := e)$. A choice construct provides the means for switching control conditionally between different code fragments. A loop construct 'loop $grd \longrightarrow\ P$ end' repeatedly evaluates P if grd is true, otherwise the construct is exited. Other Rainbow components, such as Green procedures, can be instantiated in Yellow descriptions.

2.3 Example: SMPU — a Simple Processor

A simple processor design example, SMPU, uses interacting Green and Yellow components and illustrates the mixed-view style of design description supported by Rainbow. Figure 2 shows the top-level block structure described in Visual Green, a style similar to that used in informal descriptions given in design documentation, such as for AMULET1 (Paver 1994).

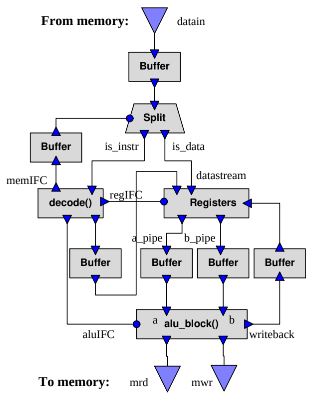

Figure 2 Visual Green SMPU

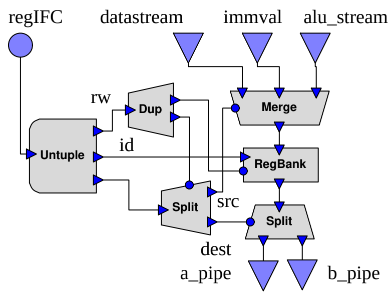

Figure 3 Registers

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. First, it outputs controls to the SMPU components to fetch an instruction from memory. When the instruction is returned, `decode` stores it in register '`i`', and then proceeds with decode and execution:

```
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) }
      ||   call aluIFC!FETCHINC
      ||   accept instruction?ins do i:=ins end
    end_par;
    //---- Execute: <code omitted> ------------------------------
  } end;
} end
```

The ALU is described using the `Table` construct of Green, which offers a concise functional 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 must match the input value. An 'empty' entry '`-`' means that an input is not consumed or an output is not produced. For other components, such as the register bank, more detailed (structural) descriptions may be given using Green sub-networks, as shown in Figure 3.

3 APA

The interfaces between the various components described in Green and Yellow must be well-defined, and is achieved by translating the sub-language components into a common underlying process term language called APA. This has some similarities to more familiar process algebras, such as CCS (Milner 1989), CSP (Hoare 1985) or LOTOS (Bolognesi & Brinksma 1987), having the usual types of process operators and an operational semantics. However, our composition operators are designed to support bundled-data micropipeline communication, and their semantics resembles that given in (Li 1982). APA supports value-passing and has richly-structured actions, which leads to a compact semantic representation. It has the following syntax:

$$
\begin{array}{lcll}
D & ::= & pnm(fargs) \triangleq P & \text{Proc defn} \\
P & ::= & \mathsf{skip} \mid \mathsf{fail} & \text{Atomic proc} \\
 & \mid & \mathsf{sync}\{ins\}\{outs\}\{P_1\} & \text{Synchronise} \\
 & \mid & grd \rightarrow P_1 & \text{Guard} \\
 & \mid & P_1 \; ; \; P_2 & \text{Sequence} \\
 & \mid & P_1 \mid P_2 & \text{Parallel}
\end{array}
\qquad
\begin{array}{lll}
\mid & P_1 + P_2 & \text{Choice} \\
\mid & P_1 \backslash \{ch\} & \text{Restrict} \\
\mid & (\nu \; x : S \; \bullet \; P_1) & \text{Var bind} \\
\mid & pnm(args) & \text{Instance}
\end{array}
$$

The semantics for APA, defined via SOS-style rules, determines the possible transitions of each term. A transition is labelled by an action containing sets of input/output channel bindings and a flag indicating whether the process has terminated. For example, the action '$\alpha = \langle \mathsf{F}, (i_1?5, c?true \;/\; o!5) \rangle$' is from a non-terminating process, with values 5 and $true$ being input on channels i_1 and c respectively, and the simultaneous output on channel o of the value 5.

The familiar process algebra combinators used in APA have their usual meaning. For example, the sequence combinator executes P_1 and then P_2. The parallel operator evolves both component processes independently combining any resulting actions with an action product operator. When either process terminates, indicated by an action '$\langle \mathsf{T}, a \rangle$' with channel bindings a, execution simply continues with the active process. However, since APA uses richly-structured actions then the action product is more complex; if the values bound to same-name output and input channels match, then references to that channel are removed from the action label, otherwise the transition fails. Any unpaired channel is left in the label unchanged.

Of particular interest is the synchronisation construct '$\mathsf{sync}\{ins\}\{outs\}\{P_1\}$'. Instead of a prefix action being performed and then followed by a separate continuation process, sync ensures that input action synchronisation is not completed until both the outputs have been consumed and the body has completed. 'Blocked' actions are introduced, written as $\pi(a)$ for an action a: when all of the inputs ins are available then the outputs $outs$ and body process P can begin execution (rule Sync-1), utilising any of the input values;

meanwhile the inputs are 'blocked'. Having completed execution of P and $outs$, then the inputs are consumed (Sync-2):

$$\boxed{\textsf{Sync-1}}\ \frac{\text{for some } b' \subseteq b \quad b'' = b - b' \quad P \xrightarrow{\langle \mathsf{F}, c\rangle} P'}{\mathsf{sync}\{a\}\{b\}\{P\} \xrightarrow{\langle \mathsf{F}, \pi(a)\ /\ b'\rangle \star \langle \mathsf{F}, c\rangle} \mathsf{sync}\{a\}\{b''\}\{P'\}}$$

$$\boxed{\textsf{Sync-2}}\ \frac{P \xrightarrow{\langle \mathsf{T}, a\rangle} P'}{\mathsf{sync}\{a\}\{b\}\{P\} \xrightarrow{\langle \mathsf{T}, a\ /\ b\rangle \star \langle \mathsf{T}, c\rangle} \mathbf{0}}$$

where b'' is the difference between the two channel maps b, b'. Therefore sync can be used to model micropipeline communication directly, without resorting to separate control channels.

3.1 APA for Green

Green constructs include buffers, duplicators, merge/split and functions. These can be modelled as APA processes, for which we can derive transition rules. For example, the behaviour of a buffer $\mathcal{B}[\ ]$ is given by the two rules*:

$$\boxed{\textsf{Buf-1}}\ \frac{}{\mathcal{B}[x] \xrightarrow{o!x} \mathcal{B}[-]} \qquad \boxed{\textsf{Buf-2}}\ \frac{\text{for some } x \in S}{\mathcal{B}[-] \xrightarrow{i?x} \mathcal{B}[x]}$$

Rule Buf-1 shows that a full buffer $\mathcal{B}[x]$ storing some value x can transform into an empty buffer $\mathcal{B}[-]$ via a transition with label $o!x$. Similarly Rule Buf-2 shows that an empty buffer $\mathcal{B}[-]$ can transform into a full buffer $\mathcal{B}[x]$ by accepting some value x from set S on an input channel i, indicated by the transition label $i?x$. The derived rules follow directly from the APA transition rules for skip, sync{_}{_}{_} and sequence.

Green split and merge constructs provide demultiplexing and multiplexing operations. For example, Split directs the value on its input to either output o_1 or to o_2 depending on the control value supplied in channel c:

$$\boxed{\textsf{Sp-1}}\ \frac{\text{for some } x \in S}{\mathsf{Split} \xrightarrow{i?x,\, c?true\ /\ o_1!x} \mathsf{Split}} \qquad \boxed{\textsf{Sp-2}}\ \frac{\text{for some } x \in S}{\mathsf{Split} \xrightarrow{i?x,\, c?false\ /\ o_2!x} \mathsf{Split}}$$

*Because Green processes never terminate, then there is no need to show the termination flag on each transition label, since it is always 'F'.

The Green stream-copying operator makes duplicate values of its input available on independent outputs. It does not acknowledge the input until both of the output copies have been consumed:

$$\boxed{\text{D-1}}\ \frac{\text{for some } x \in S}{\mathsf{Dup}_{<>} \xrightarrow{i?x\ /\ o_1!x,\, o_2!x} \mathsf{Dup}_{<>}} \qquad \boxed{\text{D-2a}}\ \frac{\text{for some } x \in S}{\mathsf{Dup}_{<>} \xrightarrow{\pi(i?x)\ /\ o_1!x} \mathsf{Dup}_{<o_2!x>}}$$

$$\boxed{\text{D-3a}}\ \frac{}{\mathsf{Dup}_{<o_2!x>} \xrightarrow{i?x\ /\ o_2!x} \mathsf{Dup}_{<>}}$$

with corresponding rules D-2b, D-3b for when output o_2 is consumed first. The Green combinators for parallel and pipe are modelled using APA parallel, restriction and renaming in the usual way.

For example, the APA term that represents the Green description of the Fibonacci calculator (Figure 1) is:

$\mathcal{B}[5](i \mapsto ch)\ \ >>\ \ \mathsf{Dup}\ \ >>\ \ (Id() \mid \mathcal{B}[3])\ \ >>\ \ Add\ \ >>\ \ \mathcal{B}[-](o \mapsto ch)$

3.2 APA for Yellow

In a similar way, we model Yellow primitives as APA processes, and derive transition rules describing their behaviour. Many Yellow primitives and combinators have direct correspondence with terms defined in APA.

The input channel communication statement '`accept {ins} do P end`' translates into the APA term '$(\nu\ \overline{x}_{ins} : S\ \cdot\ \mathsf{sync}\{ins\}\{\}\{P\})$', where $\overline{x}_{ins}$ are the free variables in the input channel bindings ins. The output statement '`call outs`' translates into the APA term '$\mathsf{sync}\{\}\{outs\}\{\}$'. Yellow sequence and parallel are also supported directly in APA.

At the pure APA level, registers in Yellow are modelled by processes. However, for clarity in the derived rules for Yellow, we introduce an evaluation environment σ to store bindings, used during evaluation of any expression e via a meaning function M-$[\![e]\!]\sigma$. Assignment to registers updates the evaluation environment σ, as shown in the following derived rule:

$$\boxed{\text{Assign}}\ \frac{\text{M-}[\![e]\!]\sigma = v}{< r\ := \ e, \sigma > \xrightarrow{\langle \mathsf{T}, \emptyset \rangle} < \mathbf{0}, \sigma \dagger \{r \mapsto v\} >}$$

For convenience, a loop operator '$\mathsf{loop}\ grd \longrightarrow\ P\ \mathsf{end}$' is introduced, which executes P repeatedly when grd evaluates to true, and exits otherwise. The following are derived rules for the loop construct:

$$\boxed{\text{L-1}}\ \frac{\text{M-}[\![grd]\!]\sigma = true \quad < P, \sigma > \xrightarrow{\langle \mathsf{F}, \alpha \rangle} < P', \sigma' >}{< \mathsf{loop}\ grd \longrightarrow\ P\ \mathsf{end}, \sigma > \xrightarrow{\langle \mathsf{F}, \alpha \rangle} < P'\ ;\ \mathsf{loop}\ grd \longrightarrow\ P\ \mathsf{end}, \sigma' >}$$

$$\boxed{\text{L-2}}\ \frac{\text{M-}[\![grd]\!]\sigma = false}{< \mathsf{loop}\ grd \longrightarrow\ P\ \mathsf{end}, \sigma > \xrightarrow{\langle \mathsf{T}, \emptyset \rangle} < \mathbf{0}, \sigma >}$$

APA thus provides a single uniform semantics for current Rainbow sub-languages, thereby supporting interworking — suitable translations from Rainbow to APA are defined. The task of translation is made easier by introducing APA processes and corresponding derived transition rules, to model many Rainbow components directly.

4 INTEROPERABILITY IN RAINBOW

A simple block adder example is used to illustrate interoperability of Yellow and Green sub-designs; the top-level is given by the Green network in Figure 4. This contains sequential control and structural dataflow components, just as

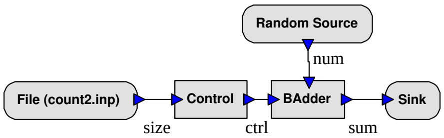

Figure 4 Top-Level Green Block Adder Unit

the SMPU in section 2.3 does. Yellow and Green are used respectively to describe the components (Figures 5 and 6).

The block adder unit `BAdder` first reads an integer value x from the input channel `size`, and then adds the next x values presented on channel `num` before outputting the cumulative sum to channel `sum`. The Yellow `control` reads `size` and then sends `true` to indicate that `adder` is to output the sum and prepare to start a new summation, or `false` to indicate that `adder` is to read the next input and add it to the sum accumulated so far.

We now show how the Yellow and Green fragments are translated into APA and the APA transition rules are used to determine possible behaviours. For the Yellow `control` process, statements are executed in sequence, so that the control-flow can be followed through the code. For example, the initial be-

```
yellow control (input size: integer;
                output ctrl: boolean) = {
   reg count: integer end
   call ctrl!true;
   loop true ->             //--Loop1
     {accept size?x do count:=x end;
      loop (count > 0) ->   //-Loop2 start
        {call ctrl!false;
         count := count - 1}
      end;
      call ctrl!true}
   end
} end
```

Figure 5 Yellow `control`

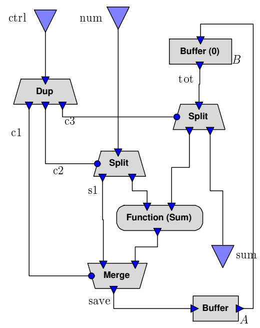

Figure 6 Green `adder`

haviour of `control` is for it to output `ctrl!true`, reaching Loop1. Figure 7 shows part of the semantics for `control` as a transition diagram. There are three states *control*, Loop1 and Loop2, with the starting state circled. Each transition is labelled by the possible input/output actions, together with conditions for execution and any register update that occurs in the transition. For example, when $count > 0$ there is a transition from Loop2 back to itself, with output `ctrl!false` and the register being decremented in the next-state (i.e. $count'$ is the next value of register $count$).

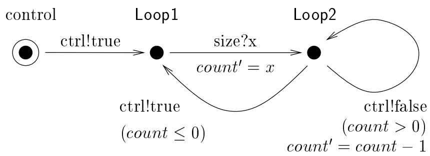

Figure 7 Transitions for Control

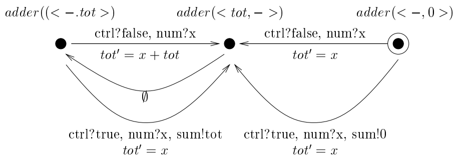

Figure 8 Transitions for Adder

In contrast, the Green dataflow description of the adder does not use the same notion of execution, in that there is no locus of control for code execution. The activity of each component is determined by its state and the various actions that can be accepted from its inputs or written to its outputs. The behaviour of the complete network is taken as the possible legal combinations of component activity.

For example, let the Green adder network be represented by '`adder(<a,b>)`', when the two buffers A, B have the values a, b. If the network is in state '`adder(<-,0)`' and it accepts `ctrl?true`, `num?x` on its inputs, then one resulting behaviour is for `sum!0` to be output, with the new state `adder(<x,->)` due to the following component transitions:

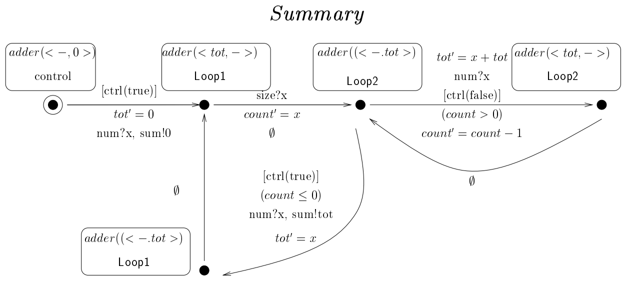

Figure 9 Transitions for Combined Machine

$$
\boxed{\mathsf{Par}}\;
\frac{
\begin{array}{ll}
\mathsf{Dup}_{<>} \xrightarrow{ctrl?true\ /\ c1!true,\, c2!true,\, c3!true} \mathsf{Dup}_{<>} & \\
\mathsf{Split} \xrightarrow{num?x,\, c2?true\ /\ s1!x} \mathsf{Split} & \\
\mathsf{Merge} \xrightarrow{s1?x,\, c1?true\ /\ save!x} \mathsf{Merge} & \mathcal{B}[-] \xrightarrow{save?x} \mathcal{B}[x] \\
\mathsf{Split} \xrightarrow{tot?0,\, c3?true\ /\ sum!0} \mathsf{Split} & \mathcal{B}[y] \xrightarrow{tot?y} \mathcal{B}[-]
\end{array}
}{
adder(< -, 0 >) \xrightarrow{ctrl?true,\, num?x\ /\ sum!0} adder(< x, - >)
}
$$

Figure 8 shows a (partial) transition diagram for the Green adder.

In order to combine the Yellow and Green components in the Add unit, the control output `ctrl` from the Yellow `control`, together with input `num` are piped into the Green `adder` as shown in Figure 4. In this case, the Pipe transition rule determines how the Yellow and Green processes interact; for example, the first initial move of the system may be calculated as follows:

$$
\boxed{\mathsf{Pipe}}\;
\frac{
\begin{array}{ll}
\text{(a)} & \{control \mid num\} \xrightarrow{ctrl!true,\, num!x} \{\mathsf{Loop1} \mid num\} \\
\text{(b)} & adder(< -, 0 >) \xrightarrow{ctrl?true,\, num?x\ /\ sum!0} adder(< x, - >)
\end{array}
}{
\begin{array}{l}
\{control \mid num\} \;>>\; adder(< -, 0 >) \\
\qquad \xrightarrow{sum!0} \{\mathsf{Loop1} \mid num\} \;>>\; adder(< x, - >)
\end{array}
}
$$

Figure 9 shows the 'product' transition diagram for the complete system, calculated from Figures 7 and 8 using the Pipe transition rule.

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 have been developed, and the formal semantic foundation gives the basis for development of formal analysis tools.

This work was supported by the UK Engineering and Physical Sciences Research Council via research grant GR/K42073.

6 REFERENCES

Barringer, H., Fellows, D., Gough, G., Jinks, P., Marsden, B. & Williams, A. (1996a), Design and simulation in rainbow: A framework for asynchronous micropipeline circuits., *in* A. Bruzzone & U. Kerckhoffs, eds, 'Proceedings of the European Simulation Symposium (ESS'96)', Vol. 2, Society for Computer Simulation International, Genoa, Italy, pp. 567–571. See also the Rainbow Project web pages, URL: `http://www.cs.man.ac.uk/fmethods/projects/AHV-PROJECT/ahv-project.html`.

Barringer, H., Gough, G., Monahan, B. & Williams, A. (1996b), 'A Process Algebra Foundation for Reasoning about Core ELLA', *The Computer Journal* **39**(4), 303–324.

Bolognesi, T. & Brinksma, E. (1987), 'Introduction to the ISO Specification Language LOTOS', *Computer Networks and ISDN Systems* **14**(1), 25–59.

Hennessy, M. (1990), *The Semantics of Programming Languages*, Wiley.

Herath, J., Yamaguchi, Y., Toda, K., Mattingley, R., Saito, N. & Yuba, T. (1992), Comparison of Dataflow Computing Models, *in* J. A. Sharp, ed., 'Data Flow Computing: Theory and Practice', Ablex, pp. 16–34.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice-Hall.

Josephs, M. & Udding, J. (1990), Delay-Insensitive Circuits: An Algebraic Approach to their Design, *in* J. Baeten & J. Klop, eds, 'CONCUR'90 (LNCS)', Vol. 458, Springer-Verlag, Amsterdam, The Netherlands.

Li, W. (1982), An Operational Semantics of Tasking and Exception Handling in Ada, Technical Report CSR-99-82, University of Edinburgh.

Liu, Y. (1995), AMULET1: Specification and Verification in CCS, PhD thesis, Department of Computer Science, University of Calgary.

Martin, A. (1990), Synthesis of asynchronous VLSI circuits, *in* J. Staunstrup, ed., 'Formal Methods for VLSI', North-Holland, chapter 6, pp. 237–283.

Milner, R. (1989), *Communication and Concurrency*, Prentice Hall, Hemel Hempstead, Herts, England.

Paver, N. (1994), The Design and Implementation of an Asynchronous Microprocessor, PhD thesis, Department of Computer Science, University of Manchester.

Sommerville, I. & Morrison, R. (1987), *Software Development with Ada*, Addison-Wesley.

Sutherland, I. (1989), 'Micropipelines', *Communications of the ACM* **32**(6).

van Berkel, K. (1992), Handshake Circuits: an Intermediary between Communicating Processes and VLSI, PhD thesis, Eindhoven University of Technology.

Yakovlev, A., Varshavsky, V., Marakhovsky, V. & Semenov, A. (1995), Designing an Asynchronous Pipeline Token Ring Interface, *in* 'Asynchronous Design Methodologies', South Bank University, London.