Delay-Insensitive Interface Specification and Synthesis

Mark B. Josephs and Dennis Furey
Centre for Concurrent Systems and VLSI
School of Computing, Information Systems, and Mathematics
South Bank University, London

Abstract

Delay-insensitive interfacing was first demonstrated on the macromodules project in the 1960's, but globally synchronous (clocked) schemes have so far dominated the VLSI era. In deep sub-micron technologies, problems of clock skew, including excessive size and power consumption of clock buffers, and heterogeneity of systems on a chip are rekindling an interest in global asynchrony. DI-Algebra is presented here as a language for the specification of modules with delay-insensitive interfaces. Such modules can be implemented either in synchronous or in asynchronous logic. A design flow is also illustrated in which specifications are automatically translated into Petri nets, validated, and synthesised into asynchronous logic.

1 Introduction

Asynchronous systems differ from the conventional approach to digital system design in that they make no use of a central clock. When multiple units interact in a typical synchronous system, such as by exchanging data, their cooperation is enforced by their mutual conformity to the operating speed dictated by the central clocking mechanism. On the other hand, in the case of an asynchronous system, the operating speed of the constituent units is not centrally regulated, but is constrained only as needed to comply with the conventions (or "protocols") agreed upon in their neighbor-to-neighbor interactions. For the right application, a cleverly designed asynchronous system is potentially faster, more efficient, and lower in engineering costs than its synchronous counterpart, but this style of design poses its own special challenges that have been a subject of ongoing research and theoretical investigation. [1]

Asynchronous circuits are often packaged into modules which communicate according to a delay-insensitive signalling scheme [3], such as four-phase handshaking [18, 14, 20, 1]. Complex VLSI systems can then be realized by hierarchical composition of such modules. A variant of Hoare's CSP language [8], known as DI-Algebra [10], is tailored to the specification of modules that provide delay-insensitive interfaces, and to the verification of their compositions.

Signal transitions (changes in the logic level of wires) control the operation of asynchronous circuits. The events or actions of DI-Algebra can be interpreted as signal transitions, and its algebraic laws capture the possibilities of reordering and interference as those transitions are propagated along wires [19].

Variants of Petri nets, such as I-Nets [15], Signal Transition Graphs [2], or Change Diagrams [11] are popular formalisms for specifying dependences between signal transitions. The `petrify` tool [4] has been used successfully to synthesise asynchronous circuits from Petri nets. Asynchronous VLSI design teams at the University of Manchester and at Cogency Technology in the UK, and at the Technical University of Denmark, have made extensive use of the `petrify` tool. The Burst-Mode approach, another graphical notation and associated tools [16, 21], is the main alternative to the Petri net approach. Burst-Mode has also been applied to real-world designs, such as cache controllers and instruction decoders.

DI-Algebra has suffered hitherto from a lack of CAD tool support, aside from some noteworthy tool building activity at the University of Groningen. (DI-Algebra has been implemented in HOL [6] and PVS [17] in order to provide support for algebraic manipulation of expressions [7, 12], and `digg` automates translation from DI-Algebra into finite state machines [13].) A route to automated circuit synthesis from DI-Algebra specifications is therefore desirable, and could be obtained by way of a translation from them to Petri nets. Our work pertains to the development of a translation tool called `di2pn` that will perform this translation automatically. The idea is that designers could

1. undertake "high-level" specification, decomposition, and verification using DI-Algebra
2. translate the specification of modules into Petri nets

[1] The support of the UK EPSRC under grant GR/M51567 and the EC under contract nr. 21949 (ACiD-WG) is acknowledged.

using `di2pn`

3. perform validation, simulation, performance analysis, and "gate level" synthesis using existing tools, such as `petrify`.

Furthermore, in view of the sizable research communities devoted respectively to process algebras and Petri nets, it is interesting to investigate a connection between them.

In the next section we provide a concrete syntax for DI-Algebra that allows the convenient expression of input/output-bursts. Section 3 explains a design flow involving `di2pn` and `petrify`.

2 Algebraic specification of modules

The behaviour of a module is specified by a set of mutually recursive equations of the form

$$I = P$$

where the expression P is constructed from the operations described below. Recursion provides the means to specify cyclic behaviour. Each identifier I corresponds to a state of the module, the final equation identifying the initial state.

The benefit of an algebraic specification language to an engineer lies in the availability of a well understood semantic model for DI-Algebra (Receptive Process Theory [9]) to facilitate such things as automated checking of a proposed design for conformity with its specification, verifying that an improved design is consistent with the original, use of CAD tools, etc. An informal, back-of-the-envelope design methodology would not afford these opportunities, and experience has shown that subtle flaws in an asynchronous design can be easily overlooked without automated assistance.

The acceptable kinds of expression follow. In these expressions, P and Q stand for arbitrary expressions, $\{xs\}$ stands for an input-burst, such as `{a,b,c}`, and $\{xs : ys\}$ stands for an input/output-burst, such as `{a,b,c:d,e}`. (An output-burst alone could be written `{:d,e}`).

- I describes a module that behaves according to the expression associated with that identifier.
- `stop` describes a module that does nothing.
- `chaos` describes a module in an undesirable state. it is implicit that the environment should avoid supplying an input-burst that would allow a module to evolve to `chaos`.
- P `with` $\{xs\}$ describes a module that behaves as P would behave when supplied with an input-burst $\{xs\}$.
- $\{xs : ys\};P$ describes a module that waits for input-burst xs, then emits the output-burst ys, and then behaves like P.
- `any` N `end` , where N can be any sequence of one or more expressions separated by the keyword `or`, describes a module that may non-deterministically choose to behave like any of the expressions in the sequence.
- `select` S `end` , where S can be any sequence of zero or more guarded-expressions of the special form $\{xs : ys\}$ `then` P separated by the keyword `or`, describes a module that will wait for an input-burst xs to tell it how to behave, whereupon it will emit the associated output-burst ys and behave like P.
- `connect` P `using` $\{xs : ys\}$ `to` Q `end` describes a module formed by connecting the modules described by P and Q together, with the output signals of each identified with the similarly named input signals of the other. xs, the input signals of P that are output signals of Q, and ys, the output signals of P that are input signals of Q, become unavailable for further connections.

Readers previously familiar with DI-Algebra will have noticed that we have replaced symbols by keywords so as to make the language more "user-friendly". Another change has been the use of bursts, rather than sequences of individual signal transitions. With the following algebraic laws, however, we can always transform a sequence of distinct input transitions followed by a sequence of distinct output transitions into a single input-output burst:

$\{xs_0\};\{xs_1\};P = \{xs\};P$, where xs_0 and xs_1 partition xs

$\{: ys_0\};\{: ys_1\};P = \{: ys\};P$, where ys_0 and ys_1 partition ys

$\{xs\};\{: ys\};P = \{xs : ys\};P$

The following algebraic law shows how to eliminate output-bursts from guards:

$\{xs : ys\}$ `then` $P = \{xs\}$ `then` $\{: ys\};P$

Nevertheless, the generalisation from a single input transition as a guard to input-bursts as guards can lead to more concise (and natural) specifications.

For readers unfamiliar with DI-Algebra, we list a few more laws that testify to the rich algebraic structure of the language.

$\{xs_0\};\{xs_1\};P =$ `stop` , if xs_0 and xs_1 are not disjoint

$\{: ys_0\};\{: ys_1\};P =$ `chaos` , if ys_0 and ys_1 are not disjoint

`stop` = `select end`

$\{xs : ys\};P =$ `select` $\{xs : ys\}$ `then` P `end`

`any` P `or` Q `end` = `select` $\{\}$ `then` P `or` $\{\}$ `then` Q `end`

Note also that "silly recursions" such as `X = X` are equivalent to `X = chaos`.

A number of case studies in DI-Algebra have been published that use an "expansion theorem" to convert connection into selection, for the purpose of verifying that a module has been correctly decomposed into a network of submodules.

We conclude this section with the algebraic specification of several well-known modules.

2.1 JOIN element

```
J = {a,b:c}; J.
```

2.2 MERGE element

```
M = select {a:c} then M
        or {b:c} then M end.
```

2.3 Martin's D element

```
D = {ar:ak}; {ar:br};
    {bk:br}; {bk:ak}; D.
```

2.4 Mutual Exclusion element

```
X = select {r0:g0} then {r0:g0};X
        or {r1:g1} then {r1:g1};X
       end.
```

3 Design flow

`di2pn` reads the contents of a file, which contains an algebraic specification, and writes a file containing an equivalent Petri net. A standard format is used for the Petri net file so as to facilitate compatibility with other tools. In particular: the tool `draw_astg`, packaged with `petrify` and interfaced to `dot`, allows the Petri net to be displayed; `petrify` itself, when provided with a library of gates, attempts to generate the logic description of an asynchronous circuit that implements the specification. In other words, `di2pn` can be used as a front-end to `petrify`.

`di2pn` also writes a file containing a list of the expressions and signal names associated with each place in the Petri net. This file is intended for documentation or debugging purposes, clarifying the correspondence between algebraic specification and Petri net.

The translation time needed by `di2pn`, invariably no more than a few tens of seconds on conventional hardware even for Petri nets with upwards of a hundred places, has not been significant enough to be worth benchmarking. The bottleneck in the design flow tends to be elsewhere. Furthermore, the asymptotic behaviour of the translation algorithm poses no impediment to its application to larger examples. In any case, `di2pn` has been written only in an interpretive, prototyping language. We estimate that an order of magnitude improvement would be possible if necessary by porting it to native code.

Details of the translation algorithm itself can be found in [5]. They are straightforward but a bit lengthy to recount fully here. Essentially, the strategy is to represent process-expressions and signal buffers as places in the Petri net. An 'expression' place is marked when the process has evolved to behave in the way expressed. A 'signal buffer' place is marked when a signal transition has been transmitted but not absorbed. For each case in which a process requires an input event in order to proceed, the places in question have arcs leading to a common transition, which upon firing deposits tokens into the places corresponding to the subsequent state and relevant output events.

The rules by which `di2pn` translates an expression in DI-Algebra into a Petri net are very simple and do not attempt to model the environment explicitly, i.e., input signal transitions are always enabled. Unfortunately, `petrify` requires a closed Petri net describing both a module and its environment in order to synthesise an asynchronous circuit that implements the module.

Our solution to this mismatch is to specify both the module and its environment in DI-Algebra. Then `di2pn` will produce a closed Petri net. A welcome side-effect of this approach is that `petrify` will now *validate* the specification: the net should be "1-safe" if the module and its environment are communicating delay-insensitively.

A word of caution needs to be added here: an implementation of the module is only guaranteed to be correct if the environment behaves according to its specification. Algebraic transformation can help, for example, by proving that the specification of the module does not change when it is restricted to the specified environment.

We now return to the examples of the previous section, specify suitable environments, and present the logic equations synthesised by `petrify` from the Petri nets produced by `di2pn`.

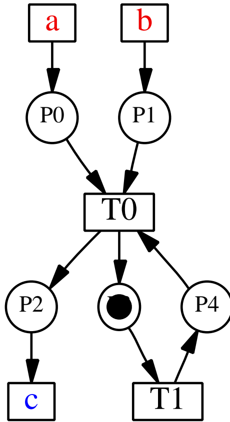
3.1 JOIN element

```
  F = {c:a,b}; F
env = F with {c}.
```

The Petri net that results from translating the combined specification of `J` and `env` is given in Fig. 1. `petrify` generates the equation $[c] = b(a+c)+ac$.

3.2 MERGE element

```
  E = any {c:a};E or {c:b};E end
env = E with {c}.
```



INPUTS: a,b
OUTPUTS: c
DUMMY: T0,T1

```
place signal buffer
----- -------------
   P0 e.a
   P1 e.b
   P2 e.c
   P3 s.a
   P4 s.b
   P5 s.c

place expression
----- -------------
   P6 F
   P7 J
   P8 {a,b:c} ; J
   P9 {c:a,b} ; F
```

Figure 1. Petri net description of JOIN element.

petrify recognises that M can be implemented with an XOR gate. Note that in the more restrictive environment given by

```
  E = any {c:a};{c:a};E
       or {c:b};{c:b};E end
env = E with {c}.
```

petrify synthesises the cheaper OR-gate implementation.

3.3 Martin's D element

```
  E = {ak:ar}; {ak:ar};
      {br:bk}; {br:bk}; E
env = E with {ak}.
```

petrify inserts an internal signal and generates the equations $[ak] = csc0' + bk; [br] = ar' csc0'; [csc0] = ar' csc0 + bk$.

3.4 Mutual Exclusion element

```
 E0 = {g0:r0}; E0
 E1 = {g1:r1}; E1
env = connect E0 with {g0} using {}
           to E1 with {g1} end.
```

Although di2pn produces a Petri net, petrify is unable to synthesise a circuit. (Indeed, MUTEX should be implemented using analog techniques.)

4 Conclusion

A concrete syntax for DI-Algebra has been presented, together with some of its algebraic laws. The language now includes input/output-bursts. Examples were given of delay-insensitive interface specification using DI-Algebra. Our tool di2pn translates such specifications into Petri nets and so can be used as a front-end to the asynchronous circuit synthesis tool petrify. This automated design flow, which includes a validation step, was applied to the examples.

References

[1] K. v. Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[2] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.

[3] W. A. Clark and C. E. Molnar. Macromodular computer systems. In R. W. Stacy and B. D. Waxman, editors, *Computers in Biomedical Research*, volume IV, chapter 3, pages 45–85. Academic Press, 1974.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, Nov. 1996.

[5] D. Furey and M. B. Josephs. Translation of DI-algebra to Petri nets. Technical Report SBU-CISM-98-10, South Bank University, Centre for Concurrent Systemsn and VLSI, 1998.

[6] M. Gordon and T. Melham. *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[7] R. Groenboom, M. B. Josephs, P. G. Lucassen, and J. T. Udding. Normal form in a delay-insensitive algebra. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 57–70. Elsevier Science Publishers, 1993.

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

[9] M. B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[10] M. B. Josephs and J. T. Udding. An algebra for delay-insensitive circuits. In R. P. Kurshan and E. M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.

[11] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[12] W. C. Mallon and J. T. Udding. Using metrics for proof rules for recursively defined delay-insensitive specifications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 175–183. IEEE Computer Society Press, Apr. 1997.

[13] W. C. Mallon and J. T. Udding. Building finite automatons from DI specifications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, 1998.

[14] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[15] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[16] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *Proc. International Conf. Computer Design (ICCD)*, pages 192–197. IEEE Computer Society Press, Oct. 1991.

[17] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In D. Kapur, editor, *11th Int. Conf. on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 748–752. Springer-Verlag, 1992.

[18] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[19] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[20] V. I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[21] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, Oct. 1992.