View from the Fringe of the Fringe

(extended summary)

Steven D. Johnson*

Indiana University Computer Science Department, sjohnson@cs.indiana.edu

Abstract. Formal analysis remains outside the mainstream of system design practice. *Interactive* methods and tools are regarded by some to be on the margin of useful research in this area. Although it may seem relatively academic to some, it is vital that this the so-called "theorem proving approach" continue to be as vigorously explored as approaches favoring highly automated reasoning. *Design derivation*, a term for design formalisms based on transformations and equivalence, represents just a small twig on the theorem-proving branch of formal system analysis. A perspective on current trends is presented from this remote outpost, including a review of the author's work since the early 1980s.

In memory of Dexter Jerry Johnson, August 12, 1918 – June 23, 2001

1 On Behalf of Interactive Reasoning

"Formal methods for systems," the application of automated symbolic reasoning to system design and analysis, remains well outside the mainstream of engineering practice, even though it is philosophically central to the science of computing (for instance, eight (by my count) of the forty-one ACM Turing Award recipients are formal methods luminaries explicitly recognized for this aspect of their work.)

One could say, on the one hand, that formal methods research in software has significantly influenced practice. Type checking, structured and object-oriented programming languages, and other advances have been deeply assimilated. So, too, have equivalence checking and hardware description languages in hardware design practice. On the other hand, it is hard to tell whether, or to what extent, these tools have improved design methodology. Do more programmers use loop invariants? Does object oriented syntax improve the way engineers organize their thoughts, or does it merely impose structure, and if the latter, does that particular structure skew other aspects, such as communication and architecture?

Let us focus on verification. In the past twenty years, the question of *whether* formal analysis is meaningful or practical has turned into a question of *how and where* they can be most profitably deployed. The digital hardware industry is

* Work described in this paper was supported by the National Science Foundation under grants MIP8707067, MIP8921842, MIP9208745, and MIP9610358.

the vanguard of this movement, the economic benefit having been established. These inroads are changing design processes, and consequently will change the perspective on and management of system design.

In many cases, perspectives are changing much faster than the rhetoric. In the early 1980s I was told by experts that equivalence checking is hopeless, that hardware description languages are unbearably inefficent, that programs are tantamount to proofs, that formal analysis (mine, in particular) bears no relationship to actual design, etc. One cannot refute these assertions; they remain true in certain contexts, both scientific and practical [1].

It is a sign of progress that one seldom hears these particular assertions at formal-methods meetings (as I did each of the examples above). The debate about practicality has become much more sophisticated. The old taunt, "Does it scale?" persists, but as often as not refers not to the capacity of the tools but to the population of engineers will use them. Even then, it is now more widely and credibly acknowledged that verification requires a fundamentally different skill set than designing.

I hope that these developments signal an end to the "algorithmic versus interactive" debate, or at least moves it to a higher level. It is certainly valid to say that any avenue into formal design analysis adopts a balance between high-level and incremental tools, and that more practical explorations will emphasize greater automation. It is equally true that successful *methodologies* will differ not in the degree of interaction, but in the nature of that interaction.

What constitutes a tolerable degree of interaction is principally a human factor, even if it is significantly influenced by the engineer's education, tool set, and design process. The nature of that interaction is another matter. This question is the crux of formalized design research. It must not be investigated from a single point of view. Speaking as an educator, it is crucial that feedback from industry distinguish between near-term practicability and far-term practicality.

In his usual clarion fashion, Hoare pinpoints the analogous distinction between top-down and bottom-up theories of programming [4]. The former supports the systematic development of correct programs but is of scant help in analyzing existing programs (i.e., debugging). The latter provides a basis for analyzing implementation properties but is not of much help in relating a program to its intended purpose. One perspective is specification oriented, and the other is implementation oriented. Until theory can rectify this dichotomy, it is the designer's task to resolve it in each instance. The same dialectic Hoare describes is present in applied research and exploratory practice.

Pnueli points out that, "...there exists no purely scientific solution [as yet] to the *system correctness problem*. We should concentrate on an engineering approach: assembling many tools and methods, ..." While acknowedging that interaction is fundamental, "One of the worst places to use ingenuity and creativity is an interactive deductive proof of verification conditions" [11].

[1] One of the statements was recently highlighted in a press release profiling the new CTO of a major technology company, who was reminiscing about how he had debunked the academic mythology of program verification.

I can think of places that can be at least as bad. Some examples: maneuvering a synthesizer by tweaking its input and output, performing ad hoc reductions to satisfy a model checker, reasoning abstractly about concrete data, and editing a `make` file for last-minute turnaround. To the extent that every design tool and environment dictates a style of expression or a mode of reasoning, it is a potential drain on ingenuity and creativity. Pnueli is arguing that the essence of engineering methodology lies in choosing the right tool for a given purpose. This is also precisely the goal of theorem proving. Of course, it is counterproductive to encumber this choice by an an overly restrictive tool set or abstruse notation, but so, too, is having an overwhelming, disorganized tool box.

What is called "theorem proving" in applied research today has much more to do with tool management than with proof editing. While it does require a proficiency in logic, gaining this proficiency is a minor task compared to assimilating the commands of an operating system or a CAD environment. Although theorem-proving remains far too restrictive (as I will argue in Section 3), interactive reasoning tools teach invaluable lessons about controling the interplay between tools and maintaining overall direction. All engineers would benefit from exploring this aspect, although not in the heat of a production schedule; that is a job for verification engineers.

This research community needs to become more expansive in its discussions of these questions, paying particular attention to avoid fading biases. With over twenty years of work, including accomplishments that could hardly have been conceived ten years ago, we can offer great deal of guidance to those who are only beginning explore practical formal methods. We might begin by neutralizing "automatic versus interactive" distinction by adopting the umbrella *automated interactive reasoning*. Since all approaches employ both, this might make it clearer that the real challenge is to effectively combine them.

2 Interlude: Are we Making Adequate Progress?

At a 1998 workshop on formal methods education [8], in the course of a discussion about mathematics in the CS/CE, Douglas Troeger made the remark, "Calculus is a formal method." I have been in a great many discussions concerning The Calculus in computer science, but I was struck for the first time by the literal truth of Troeger's statement. I thought that learning how The Calculus came into being might give me a fresh perspective on computing methodology. A colleague recommended Carl B. Boyer's *The History of the Calculus and its Conceptual Development* [2].

WARNING: Beware the musings of someone who has read just one history book.

I found it somewhat comforting that, according to Boyer's scholarly account, The Calculus was the product of no less than a millennium of active thought (elapsed time: around 2,500 years), with due acknowledgment to the Greeks, Babylonians, Egyptians, Hindus, Arabs, and Medieval scholars. However, it was with increasing dismay that, the more I read, Greek Age seemed more familiar

than the Renaissance. I am fond of saying that computing has not yet seen its Newton (Liebnez if you prefer), but maybe I should be saying Archimedes (Eudoxus if you prefer). The Greeks also had, in Zeno, one of history's greatest nay-sayers. I'm not sure whether computing has seen its Zeno, although some candidates do come to mind.

The Greek Method of Exhaustion could solve integration problems, but was cumbersome and indirect; it lacked the logical foundation to deal even with primary ideas (e.g. measure and variable), much less key concepts (e.g. limit and convergence). Instead, The Method is based on the concept of approximation to to any desired degree of precision. If the Greeks had computers, that might have been the end of it.

It is fairer to say that Newton and Leibnez acquiesced to The Calculus than to say they discovered it; and they certainly didn't invent it. Invention cannot be attributed to any individual, but to a gradual evolution in thinking over several centuries. Newton and Liebnez, and perhaps others, discovered the surprising relationship between the integral and the derivative, and went on to use their discoveries. Two more centuries passed before a mathematical foundation was established to justify their methods. Throughout the history of The Calculus, formalism played both encouraging and inhibiting roles. In his conclusion, Boyer says,

> Perhaps the most manifest deterring force was the rigid insistence on the exclusion from mathematics of any idea not at the time allowing of strict logical interpretation. ... On the other hand, perhaps a more subtle, and therefore serious, hindrance to the development of the calculus was the failure, at various stages, to give to the concepts employed as concise and formal definition as was possible at the time.

While it is foolhardy to draw too many parallels between the history of the calculus of the physical world and the search for a calculus of system design (if, indeed, there is one), Boyer's comment surely applies to both. Formalism should describe, not prescribe, but it must be involved, and this applies not just to the mathematics, but also to its implementation in a computer.

I could not find in Boyer's account a clear sense of what was driving the persuit of The Calculus forward. Science and engineering, mathematics, metaphysics, theology, and politics all played a role, almost always in combination within each individual contributor. It is quite clear, of course, that The Calculus prevailed because its methods mad problems easier to solve. However, many of these problems were purely mathematical, predating any direct practical application. Once published, The Calculus became entrenched through scientific and engineering applications, despite rather strong criticism by formalists and mathematical theorists.

3 DDD - the Far Side of Theorem Proving

In Section 1, I spoke on behalf of interactive reasoning in automated formal verification. In this section I will describe some weaknesses in mainstream "theorem

proving" research, from the standpoint of someone whose work investigates an alternate formalism.

If automated formal verification has established a beachhead in commercial integrated circuit design, interactive analysis has gained a foothold, at least. In the past few years, the successful infusion of formalized analysis in the design of commercial microprocessors includes a significant component of theorem proving. In research, automated and interactive verification are adapting to each other, and, in some cases, overlapping. The systems used in interactive reasoning, more properly called proof assistants than theorem provers, are quite advanced, having been developed over decades of applied research. Nearly all of them are based on predicate logic.

Just as most mathematical arguments involve both logic and algebra, interactive design analysis is improved by provisions for reasoning based on equivalence rather than implication. I use the term *derivation* to distinquish this mode of reasoning from *deductive* reasoning based on logical inference.

Before contrasting these two approaches, I should discuss what they have in common, for this is far more significant in the greater scheme of things. In all forms of interactive formal reasoning the object is to construct a *proof*, or sequence of commands invoking inference rules. A proof might be compared to a Unix `make` file; it is a command script that can be and is repeatedly executed. Specification and implementation expressions are byproducts of the proof building process. Derivational formalisms are sometimes described as "constructing" correct implementations, whereas deductive formalisms relate pre-existing design expressions. This is an invalid distinction for the most part. Regardless of the proof rules, proof construction is a creative process involving forward and backward reasoning, decomposition into subgoals, backtracking, specification revision, implementation adjustment, and so forth. The final form of the proof says almost nothing about how it was obtained.

In practice, proof construction is often systematic. All proof assistants automate some proof building tactics, ranging from embedded decision procedures to parameterized induction principles. Most proof assistants provide a metalanguage for composing basic patterns into proof-engineering strategies.

3.1 DDD and its Origins

The acronym *DDD* refers either to "digital design derivation," the application of derivational reasoning to digital design, or the research tool that was developed to explore it. This line of investigation began in the early 1980s and continues to the present. Reference [6] contains a bibliography of DDD research for those interested in more details.

I originally described DDD as "functional programming applied to hardware," but eventually had to abandon this slogan. Even though any reasonable design formalism will encompass both hardware and software, equating hardware descriptions to software programs is still quite a leap for most people. Furthermore, the automation of reasoning based on functional model theory remains (so far as I can tell) out of step with the priority of language implementation over

formal reasoning in the functional programming community. In any case, this work found a more receptive audience in formal methods the emerging formal verification area.

The work did begin with a language implementation, however. As a research assistant I explored the use of functional programming techniques in implementing operating systems (and later, distributed programming). I developed a programming language called *Daisy* to explore abstractions of communciation and concurrency, which were modeled as streams and represented by lazy lists.

Given the simplicity of digital devices at the time, and since I was more concerned with how processes communicated than with what they did, it was convenient to illustrate ideas about programming with with hardware models. This rather innocuous tactic immediately changed the direction of the work (although not its ultimate goal, which transcends hardware/software distinctions). For one thing, it induced a sharp distinction between description and realization, something I had trouble isolating in software studies. For another, designing hardware requires a more balanced regard for design aspects (principally architecture, behavior, coordination, and data).

A thesis emerged that each aspect of design has a distinct conceptual structure, and perhaps a distinct notation. A design formalism must be able to move among these structures, and cannot simply compose them without destroying one or more. Figure 1 gives a sense of this thesis while illustrating some of DDD's formal representations. Behavioral specification is represented by a recursive system of functions definitions. Architectureal specification is represented by a recursive system of streams. Data (abstract types and implementation relationships) and synchronization (e.g. as expressed in timing diagrams) aspects are not shown, but rather their infusion in architectural expressions.

As this is applied research, it is important to demonstrate that the formalism can be used to obtain real hardware of a reasonable complexity (given that the work is done at academic institution without engineering programs). Well over half of the research effort is spent doing case studies, the majority of that effort being to integrate DDD with a never-ending sequence of CAD tools and VLSI technologies. Two major demonstrations were the construction of a language-specific computer and comparative study of microprocessor verification:

- In a series of studies, DDD was used to derive working versions of Warren Hunt's FM8501 and FM9001 microprocessor, which Hunt verified using the *Nqthm* and *ACL2* theorem provers. The derived devices worked on first power-up, but were also extensively tested against Hunt's chip for instruction-level comformance. This study demonstrated that derivation alone is insufficient support for design; it must be integrated with a other formalisms to to deal with data and process abstractions, and ingenious design decisions.
- A language-specific computer for direct execution of compiled Scheme reflects the functional-programming roots of the research. DDD is implemented in Scheme and operates on Scheme expressions, so applying it to Scheme brings a kind of closure. DDD provides implementation verification of the

system components, a CPU, memory allocator, and garbage collector; but does not prove system-level noninterference properties; we will use a model checker for that. And DDD does not prove specifcation correctness; we will use a theorem prover for that. However, the CPU and garbage collector are extremely similar to the those of *VLISP* a Scheme implementation, whose compiler and run-time system were proved using rigorous but unmechanized mathematics [3]. In conjunction, these two studies represent a complete formal design whose scope of abstraction ranges from pure denotational semantics to hardware realization.

In the mid 1990s, students from the DDD group started a company to commercialize their research. They have produced two commercial products using formalized derivation, a synthesizable PCI bus interface and a configurable hardware core for Java byte code generation. The Java processor was one of the first to enter the marketplace. These accomplishments are evidence that formalized design derivation can be practical.

3.2 Toward Integrated Automated Interactive Reasoning

All of the studies just described involved the coordinated use of several reasoning and synthesis tools. I would now like to focus on some of the difficulties we have encountered doing these studies. They point to a need for improving the methodology, the tools, and also the attitudes, of interactive design analysis.

One of the more aggravating statements about DDD, appearing more that than once in published papers, is that it is not "formalized." This is an unwarrented detraction considered as a mathematical or logical statement. However, means merely that the soundness proofs of DDD's transformation rules have note been checked in some theorem prover or other. In that extremelyh parochial sense, the statement is correct but also somewhat hypocritcal. Not many theorem proving tools have proved their own inference rules; those that have are unlikley to have proven the implementation of those rules; and those that have done that, if any, have not greatly enhanced their utility by doing so.

The suggestion that all formalisms should be embedded in a single predicate logic is quaintly formalist, but the corollary that all interactive reasoning tools should be implemented in a particular theorem prover is simply wrong from both engineering and methodological standpoints.

3.3 Foundational Issues in Logic

Paul Miner explored the problems of interaction between DDD and the PVS theorem prover [9]. The problems are by no means unique to PVS; to the contrary, PVS was fairly adept for working around some of them. However, each work-around involves a semantic embedding with strategies tailored to make the embedding transparent. This may be possible on a case-by-case basis, but in combination the embeddings are certain to interfere with each other and break that transparency, leaving the user to untangle the consequences.

The most immediate problem is a lack of support for mutual recursions. This alone makes embedding DDD in PVS untenable, but that wasn't our goal anyway. We wanted instead to export verification conditions from DDD to PVS in order to verify ingenious design optimizations, especially those done on stream networks. However, streams domains are not well founded, raising a significant problem in the PVS logic. Miner was able to work around this problem, developing a *co-algebraic* theory of streams, based on their conventional representation as functions on the natural numbers, and a collection of higher order predicates and PVS strategies to mask that representation.

To illustrate the issue, consider the functions zip and map defined for streams:

$$\begin{aligned}\mathsf{zip}(a:A, b:B) &\stackrel{\text{df}}{=} a : b : \mathsf{zip}(A, B)\\ \mathsf{map}(f, a:A) &\stackrel{\text{df}}{=} f(a) : \mathsf{map}(f, A)\end{aligned}$$

Proving that map distributes over zip is intuitively straightforward.

$$\begin{array}{ll}\mathsf{map}(f, \mathsf{zip}(a:A, b:B)) & \\ = \mathsf{map}(f, a : b : \mathsf{zip}(A, B)) & (\text{defn. } \mathsf{zip})\\ = f(a) : f(b) : \mathsf{map}(f, \mathsf{zip}(A, B)) & (\text{defn. } \mathsf{map})\\ \stackrel{\text{H}}{=} f(a) : f(b) : \mathsf{zip}(\mathsf{map}(f, A), \mathsf{map}(f, B)) & (\textit{induction})\\ = \mathsf{zip}(f(a) : \mathsf{map}(f, A), f(b) : \mathsf{map}(f, B)) & (\text{defn. } \mathsf{zip})\end{array}$$

Although true, the '$\stackrel{\text{H}}{=}$' step is not a logically valid induction in PVS because streams are not well founded (There is no base case in zip's defintion, for example.) This inconvenience can be circumvented in a number of ways including (See [5] for one entry point into the literature on these matters).

(a) It is simply an abbreviated mathematical induction based on a representation of streams as functions over the natural numbers and extensionality.
(b) It is a recursion induction over the domain $S[\alpha] \stackrel{\text{df}}{=} \alpha \times S[\alpha]$, or (or a similar directed-complete partial order).
(c) It is an stylized proof that the relation:

$$\sim \; = \; \{(\mathsf{map}(f, \mathsf{zip}(s, s')), \mathsf{zip}(\mathsf{map}(f, s), \mathsf{map}(f, s'))) \mid s \text{ and } s' \text{ streams}\}$$

is a bisimulation. We should be using '$\sim$' rather than '$\stackrel{H}{=}$' in the third step.
(d) It is a co-induction.

Interpretation (a) is still fairly common, but it reminds me of the Greek Method of Exhaustion: cumbersome and unnecessarily indirect. Streams and other infinite objects are fundmental in computing theory, and should not be bound to a particular model [1]. Furthermore, the argument is a "forward" induction, not a standard one. Miner's PVS formalization is essentially (c) as justified by (a) and extensionality, but without the implicit assumption of equality. He codes strategies to perform bisimulation proofs to give the appearance of (d).

More direct mathematical foundations have recently appeared. Figure 2 shows a complete [10] logic for recursive equations with first-order substitution, due to

Hurkens, McArthur, Moschovakis, Moss, and Whitney [5]. *First-order substitution* means that the recursive equations define non-parameterized objects, like streams. The generalization to first-order systems of recursive functions is in progress.

Figure 3 shows five transformation rules forming a basis for DDD architectural (stream) transformations. Introduction and identification are derived instances of the recursion rule in Figure 2. The replacement rule extends equality to the underlying data type. Grouping is a structural rule allowing for the manipulation of multivalued functions. The collation rule, originally published as a multiplexing rule, is essentially a version of specialization.

Thus, the logic in Figure 2 is much better suited to the DDD formalism than the PVS sequent calculus, suggesting that an "deep embedding" of DDD in PVS is not appropriate, even if it were practical to do so.

Interactions between Interactive Formalisms. From the discussion above, the idea of a single logical framework serving as the umbrella environment for formalized analysis raises foundational problems. However, this is applied research. A derivation is the proof of a theorem about equivalence, and so can be stated in a logic whether proven or not. Conversely, I have already acknowledged that useful transformation system requires the support of a logic, for instance, to validate conditional transformations.

We have done a lot of work trying to integrate DDD with PVS (and some preliminary work to integrate with ACL2). We have found the software engineering hurdles to be more vexing than the semantic problems. To some extent, these problems mirror the conceit that all reasoning should be embedded in a one particular logic. Theorem provers, and DDD is no exception, are simply not designed to talk to other theorem provers as peers.

There are differences in look and feel between building a deductive proof and building a derivation. Taking PVS as an example, the two main proof windows contain the text of relevant theories (definitions and theorems), and the proof in the form of a sequent. Commands to the prover are mainly actions performed on the sequent, although the process typically induces revisions to the theories. In DDD, there are also two main windows, one the text of the expression being transformed, the other a history of the transformations that have been applied. The user focuses on the expression, not the derviation, so that interaction has the feel of a syntax directed expression editor.

Thus, DDD and PVS interactions are complementary. One can imagine a single environment supporting both, but such an environment tool would have two distinct modes of interaction. It being unlikely that a consolidated interface will appear any time soon, progress depends on the two systems talking to each other. Unfortunately, neither DDD nor PVS, nor any of the interactive tools we have used, has provisions for back-channel communication. These systems assume a single user and also impose strong restrictions on their file system.

Whether these problematic qualities are a consequence of a a formalist's pecking order, or lack of software engineering resources, they need to be repaired.

4 Conclusions

In formal methods research there is a moving boundary between what is practical and what is productive. Productivity is affected by both automation and expertise. The successful industrial users of formal verification are beginning to recognize "verification engineering" as a distinct skill set, to which design processes need to adapt. While it is important to focus on the transition from practical to productive, there is considerable danger in extrapolating from the past. It is certain that the best methods will eventually prevail and be reflected in automated interactive reasoning. Hastening that eventuality requires a healthy competition between "top-down" and "bottom-up" approaches, not only in theory, but in applied research and exploratory practice. What I have found, and have tried to illustrate in this paper, is that this dialectic does not arise in just one place but pervades every level of abstraction.

References

1. Jon Barwise and Lawrence Moss. *Vicious Circles*. CLSI Publications, Stanford, California, 1996.
2. Carl B. Boyer. *The History of the Calculus and its Conceptual Development*. Dover, New York, 1959. Republished 1949 edition.
3. Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.
4. C. A. R. Hoare. Theories of programming: Top-down and bottom-up meeting in the middle. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods*. LNCS 1708.
5. A. J. C. Hurkens, Monica McArthur, Yiannis M. Moschovakis, Lawrence S. Moss, and Glen T. Whitney. The logic of recursive equations. *The Journal of Symbolic Logic*, 63(2):451–478, June 1998.
6. Steven D. Johnson. The Indiana University System Design Methods Laboratory home page. `http://www.cs.indiana.edu/hmg/hmg.html`.
7. Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 260–281. LNCS 408, 1989.
8. Steven D. Johnson. A workshop on formal methods education: an aggregation of opinions. *International Journal on Software Tools for Technology Transfer*, 2(3):203–207, November 1999.
9. Steven D. Johnson and Paul S. Miner. Integrated reasoning support in system design: design derivation and theorem proving. In Hon F. Li and David K. Probst, editors, *Advances in Hardware Design and Verification (CHARME'97)*, pages 255–272. Chapman-Hall, 1997.
10. Lawrence S. Moss. Recursion and corecursion have the same equational logic. *Theoretical Computer Science*, to appear in 2002. `http://math.indiana.edu/home/moss/home.html`.
11. Amir Pnueli. These quotations are extracted from transparencies for invited talks at PODC'90, FM'99, and CAV'00. They can be found at `//www.wisdom.weizmann.ac.il/TEamir/invited-talks.html`.

BEHAVIOR:

$$\begin{aligned}\mathsf{gcd}_b(x,y) \stackrel{\text{df}}{=} \mathit{if}\ \ & x = y\ \mathit{then}\ \ x \\ \mathit{else}\ \mathit{if}\ \ & x < y\ \mathit{then}\ \ \mathsf{gcd}_b(x, y-x) \\ & \mathit{else}\ \ \mathsf{gcd}_b(x-y, y)\end{aligned}$$

ARCHITECTURE:

$$\begin{aligned}&\mathsf{gcd}_a(\check{x},\check{y}) \stackrel{\text{df}}{=} x\ \mathit{where} \\ &\quad x = \check{x} : \mathit{sel}(x \geq y, x, y-x) \\ &\quad y = \check{y} : \mathit{sel}(x > y, x-y, y)\end{aligned}$$

COORDINATION: (request-acknowledge synchronization)

$$\begin{aligned}&\mathsf{gcd}_c(x,y,\mathit{req}) \stackrel{\text{df}}{=} (z, \mathit{ack})\ \mathit{where} \\ &\quad u = \mathit{sel}(\mathit{req}, x, \mathit{sel}(u \geq v, u-v, u)) \\ &\quad v = \mathit{sel}(\mathit{req}, y \mathit{sel}(u > v, v, v-u)) \\ &\quad z = u \\ &\quad \mathit{ack} = u = v\end{aligned}$$

DATA: ($\langle Bit^2, +, \cdot, \ldots\rangle$ implementing $\langle Int, -, >\rangle$)

$$\begin{aligned}&\mathsf{gcd}_d((\check{x}_1,\check{x}_0),(\check{y}_1,\check{y}_0)) \stackrel{\text{df}}{=} (x_1, x_0)\ \mathit{where} \\ &\quad x_0 = \check{x}_0 : \mathit{sel}(ge?, x_0, d_0) \\ &\quad x_1 = \check{x}_1 : \mathit{sel}(ge?, x_1, d_1) \\ &\quad y_0 = \check{y}_0 : \mathit{sel}(gt?, d_0, y_0) \\ &\quad y_1 = \check{y}_1 : \mathit{sel}(gt?, d_1, y_1) \\ &\quad gt? = x_1\overline{y_0} + x_0\overline{y_0}(x_1 + \overline{y_1}) \\ &\quad ge? = (x_0 \odot y_0)(x_1 \odot y_1) + gt? \\ &\quad (d_1, d_0) = \mathsf{sub}(\ \mathit{sel}(ge?, (x_1, x_0), (y_1, y_0)) \\ &\qquad\qquad \mathit{sel}(ge?, (y_1, y_0), (x_1, x_0)))\end{aligned}$$

Fig. 1. Four facets of system design represented as recursive equations. All but gcd_b are stream networks. The expression $\check{v} : v$ denotes a stream whose head is the value $\check{v}$ and whose tail is the stream v; *sel* is a two-way selector; the operations are boolean in gcd_d and arithmetic elsewhere. The tetrahedron symbolized the thesis that one expression cannot simultaneously reflect all design aspects, but must suppress one or more.

TAUTOLOGY: $\phi \vdash \phi$

EQUALITY: $\vdash A = A$, $A = B \vdash B = A$, and $A = B$, $B = C \vdash A = C$.

REPLACEMENT: $A = B \vdash E[A/x] = E[B/x]$, provided the substitutions are free.

SPECIALIZATION: $\forall x(\phi(x)) \vdash \phi(E)$, provided the substitution is free.

WEAKENING: If $\Gamma \vdash \phi$ then $\Gamma \cup \Delta \vdash \phi$.

CUT: If Γ, $\psi \vdash \phi$ and $\Gamma \vdash \psi$ then $\Gamma \vdash \phi$.

GENERALIZATION: If $\Gamma \vdash \phi(x)$ then $\Gamma \vdash \forall x(\phi(x))$ provided x is not free in Γ.

HEAD: $\vdash A(x_1, \ldots, x_n)$ `where` $\{\vec{x} = \vec{B}\}$
$= A(x_1$ `where` $\{\vec{x} = \vec{B}\}, \ldots, x_n$ `where` $\{\vec{x} = \vec{B}\})$

BEKIČ-SCOTT: $\vdash A$ `where` $\{\vec{y} = \vec{C}, \vec{x} = \vec{B}\}$.
$= (A$ `where` $\{\vec{y} = \vec{C}\}$ `where` $\{\ldots, x_i = B_i$ `where` $\{\vec{y} = \vec{C}\}, \ldots\}$

FIXPOINT: $\vdash A$ `where` $\{x = A\} = x$ `where` $\{x = A\}$.

RECURSION:
Given $\mathsf{A} \equiv A_0$ `where` $\{x_1 = A_1, \ldots, A_n = A_n\}$,
$\mathsf{B} \equiv B_0$ `where` $\{y_1 = B_1, \ldots, y_n = B_m\}$,
and a set, Σ of equations of the form $(x_i = y_j)$,
if $\Gamma, \Sigma \vdash A_0 = B_0$, and $\Gamma, \Sigma \vdash A_i = B_j$for each $(x_i = y_j) \in \Sigma$
then $\Gamma \vdash \mathsf{A} = \mathsf{B}$.
provided No x_i/y_j occurs in B/A and no x_i or y_j occurs free in Γ.

Fig. 2. A complete logic for recursive systems with first-order substitution [5]

INTRODUCTION:
A `where` $\{\vec{x} = \vec{B}\} \Leftrightarrow A$ `where` $\{\vec{x} = \vec{B}, y = C\}$
Provided y is a new variable, and the r.h.s. is well formed.

IDENTIFICATION:
A `where` $\{\ldots, x = B, y = C, \ldots\} \Leftrightarrow A$ `where` $\{\ldots, x = B[C/y], y = C, \ldots\}$.

REPLACEMENT:
A `where` $\{\ldots, x = B[C/y], \ldots\} \Leftrightarrow A$ `where` $\{\ldots, x = B[D/y], \ldots\}$
provided $\Delta \models C = D$.

SIGNAL GROUPING:
A `where` $\{x = B, y = C, \ldots\} \Leftrightarrow A$ `where` $\{(x, y) = (B, C), \ldots\}$.
with a suitable generalization of substitution.

COLLATION:
$E[\natural/x] \Rightarrow E[B/x]$
a one-way rewriting version of specialization, $\natural$ a generic "don't-care" constant.

Fig. 3. Adequate rules for DDD transformations on stream networks [7]. The Δ in the replacement rule stands for the underlying type theory.