Research Prospectus

Decomposing Digital-system Specifications into Interacting Sequential Processes

December 1991

Abstract

This 2-year project description is part of the *Digital Design Derivation Project* of the Hardware Methods Laboratory, Computer Science Department, Indiana University. Participating researchers, students, and staff include Prof. Steven D. Johnson, Prof. David E. Winkel, Bhaskar Bose, Zheng Zhu, Kamlesh Rath, Esen Tuna, C. David Boyer, and Willie Hunt.

The 2-year project investigates *sequential decomposition*: a process by which higher specifications of digital-system behavior are broken down into interacting sequential subsystems. We will develop a treatment in the framework of functional algebra, defining formal representations for relevant abstractions, building a set of transformations for manipulating these representations, and incorporating these transformations in a mechanical reasoning system for hardware derivation. The aim is to develop a "correct-by-construction" method of verification that is on an equal footing with deductive formalisms (theorem provers) in use today.

An *interaction scheme* $P[f]$ is a formula expressing a protocol between two processes but abstract with respect to function f. Given a procedure F we want to construct an instance $P[F]$ of the interaction. For an expression $E[F]$ that uses F, the goal is to factor E into a complementary instance, $E[\bar{P}[F]]$, in a manner that preserves a suitable notion of equivalence. That is, we want

$$E[F] \approx E[\bar{P}[F]] \circ P[F]$$

A number of demonstration experiments will be performed to illustrate results of the theoretical work. They will include at least two full-system prototypes, to be developed in the Indiana University Hardware Methods Laboratory.

1 Project Description

1.1 Sequential Decomposition

Digital systems are composed of interacting sequential processes, but the description of sequential interaction is not well captured by current hardware description languages. In an article on interface synthesis (which has great bearing on the research proposed here) Borriello states,

> ..., the interface component has received very limited attention [in high-level synthesis research] even though it is crucial to integrating the circuit into an environment that will put it to use [2].

This situation will change over the next five years, as high-level synthesis continues to emerge in practice. We will begin to see greater interest in and support for procedural and protocol abstraction at the specification level, as CAD research moves in the direction of "system synthesis" [10].

There has been dramatic progress in in finite-state machine verification [5], and steady progress in the use of process formalisms to reason at higher levels of specification (see Section 4.3). However, nearly all of this research involves post-design verification. As elements of system synthesis comes on line, it will become more difficult and less useful to reason in this fashion. The time has arrived to lay semantic foundations for constructive design processes at the system level.

This project investigates *sequential decomposition*: a formal process by which digital-system descriptions are broken down into interacting sequential subsystems. We will develop a treatment in the framework of functional algebra, defining formal representations for relevant abstractions, building a set of transformations for manipulating these representations, and incorporating these transformations in a mechanical system for hardware derivation.

This is an investigation of formalized reasoning rather than automatic design. We use the term *design derivation* to differentiate from synthesis on the one hand, and post-design verification on the other. Although it is valid to say that formal derivation is a species of formal verification, that would add confusion to an already overloaded vocabulary. As we discuss in briefly Section 4.1.2, derivation oriented methods should not be regarded as antithetical to deduction oriented methods. To the contrary, these reflect alternate modes of reasoning that should be integrated. Toward this end, we want to develop a system that is on an equal footing, mathematically and mechanically, with deductive systems in use today. Unlike a good CAD tool, it trades proficiency for generality.

Our prior research has produced an interactive transformation system, described in Section 4.2. that has been used successfully in moderate-scale applications. A significant portion this project will be devoted to the further enhancement of this system, to incorporate new techniques resulting from the research, and to make general improvements to the system. We will apply this system to a range of new examples, as described in Section 4.6.2.

Of course, “sequential process” can mean anything from a delay element to a microprocessor to a telephone network. We shall focus on a limited range of devices, such as simple arithmetic units, memory systems, and interfaces, that arise in digital-ASIC design. We are specifically concerned with adapting our formalism to control-dominated problems. Although this is merely a step in the direction of true system specification, we believe it reflects new trends in automated synthesis; and we are certain that it contributes to the current limited understanding found in higher-level formal research.

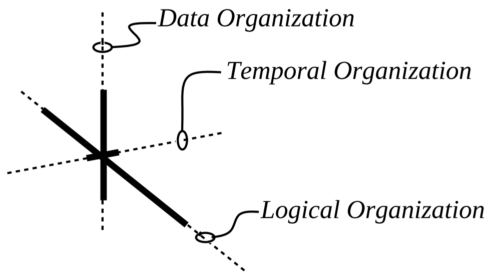
Section 4.4 introduces enough of the existing formalism to make sense of later discussions. The approach we will take to sequential decomposition is described in Section 4.5. Briefly, an *interaction scheme* $P[f]$ is a formula expressing a protocol between two processes but abstract with respect to function f. Given a procedure F we want to construct an instance $P[F]$ of the interaction. For an expression $E[F]$ that uses F, the goal is to factor E into a complementary instance, $E[\bar{P}[F]]$, in a manner that preserves a suitable notion of equivalence. That is, we want

$$E[F] \approx E[\bar{P}[F]] \circ P[F]$$

Our notion of interaction scheme is quite preliminary. Section 4.6.1 reviews the research topics that will make it more concrete.

1.1.1 Background

We use a functional modeling theory to represent the behavior and structure of digital systems. Reasoning in this framework is formalized by classes of equivalence preserving transformations.[1] Intuitively, we are concerned with three dimensions of a formalized design space:



Logical organization refers to translations from behavior to architecture. It is the most highly developed aspect in our formalism. *Data organization* refers to the hierarchy of type abstractions employed during design. We have limited methods to move between levels of data definition. This is significant because while current synthesis systems support hierarchical structural description, they do not support data abstraction.

In order to expand in either of these two dimensions, we must incorporate algebra for manipulating *temporal organization*—encompassing synchronization and data-communication

[1] “Equivalence preserving” is a contingent notion. Transformations can synthesize logical side-conditions, just as in ordinary algebra (for example) division by a variable x generates the side condition “$x \neq 0$.” This is one way to integrate deductive reasoning with derivational reasoning. See Section 4.1.2.

protocols. With this capability, complex entities can be encapsulated as non-trivial sequential processes; and we will be able specify above the register-transfer level.

A typical first example involves implementation of memory. Suppose we specify at a level that views memory as an array, so `read` and `writes` are viewd as combinational operations. We often want to realize this abstraction using dynamic RAMs, so that *either* the circuit must be built from basic components that make the DRAM interactions transparent *or* the specification must be transformed to account for refresh cycles, address multiplexing, and so on. Either alternative is valid, so the formalism should not dictate a choice.

The standard treatment of this problem is to revise the specification and then prove the revision correct. However, in cases like this a systematic transformation is possible, and would be preferable verifying a revision.

On the other hand, we do not believe a comprehensive construction is attainable. A general method would have to include programmable devices, and hence be higher order. Even if it did not, many real devices are too sophisticated to be handled in way (See Section 4.3.2). Our research will address a limited domain of decomposition, including simple memory structures, ALUs, interfaces, etc., that commonly arise in ASIC design. This would permit significantly higher levels of specification than are presently available but, as just noted, would not eliminate the need for correctness proofs when formal correctness is desired.

1.1.2 Formalized Reasoning and Heterogeneous Verification

We shall define "formal reasoning" as the representation of reasoning processes in a formal system. A formal system is defined by rules of syntax, saying how formulas are built, and rules of reasoning, saying how formulas are manipulated. This terminology applies to research in which theorem provers, proof assistants, rewriting systems, transformation systems, *etc.*, are employed to reason about hardware. It does not apply—at least not directly—to model based formal-methods work in boolean and finite-state machine verification, high-level synthesis, and symbolic simulation.

In this context, *verification* refers to a formal proof relating two expressions, S (the specification) and I (the implementation). In a logical framework proofs are often manifested as deductions, the intuition being that both S and I are known in advance. In an algebraic formalism, proofs are manifested as derivations. The contrasting impression is one of reasoning *from* S *toward* I; there is a sense of using the algebra to discover an implementation.

While these intuitions are misleading (see [24] or [35] for example) because proof *construction* is non-linear in either case, it is nevertheless evident that designing something involves both before-the-fact and after-the-fact reasoning. The underlying motive of our research is to explore this relationship formally, in particular, to look for ways to support multiple modes of reasoning mechanically. We believe this to be a fundamental aspect of automated design that has not been widely investigated in a formal setting.

1.2 The DDD Transformation System

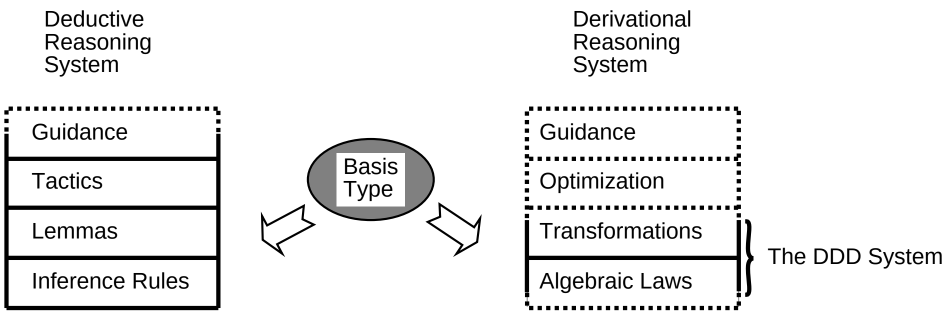
DDD is a mechanized algebra for deriving synthesizable hardware descriptions from higher-level functional expressions [19]. The primary inputs to DDD are:

- A system of function definitions serving as a behavioral specification.
- A declaration relating two or more levels of data abstraction.
- A *derivation script*, or sequence of transformations.

From the designer's point of view, the task is to develop a derivation script that produces an acceptable implementation, usually a boolean network for logic synthesis. DDD outputs

- A series expressions that can be executed to explore design properties.
- A collection of auxiliary definitions maintaining the meaning of the initial specification. Often these are interpretable as constraints on the implementation environment.

Although DDD has sometimes been classified as a synthesis system, it is more accurate and meaningful to compare it to a an theorem proving environment:



A typical theorem prover is built on a basic inference engine, providing the elementary mechanisms to construct proofs. These are generalized over types structures and so can be applied at arbitrary levels of description. Most systems can also package theorems for later use. There may be provisions for proof automation, through built-in tactics (e.g. induction) or a degree of programmability (e.g. HOL *tacticals* [12]). Mature systems have a top-level proof management environment.

DDD is also an automated formal system. In place of an inference engine, it provides elementary algebra from which courser-grained transformations are composed. At its present stage of development however, DDD lacks both the tactical and management levels of a well developed theorem prover. These should develop over time and experience.

As evidenced by this proposal, we make no completeness claims for the elementary level.[2] In fact, we do not even claim that the basic algebra is elementary. Many of DDD's transformations could be further decomposed, and it remains a topic of research to find a good stratification of its reasoning rules.

[2]although it is complete in the sense of [33], for example

We have used DDD to build several working digital systems. Most of these have been data dominated special-purpose processors [20, 3]. In [34] we produced a whole computer system, but DDD was applied only to individual components (CPU, garbage collector, I/O interface, memory subsystem). An experimental goal of the proposed project is to specify and derive similar systems entirely within DDD.

1.3 Related research

1.3.1 Heterogeneous Verification

Combining deductive verification with other modes of reasoning is an emerging topic in formal methodology. Hanna, Fourman, and others have considered interactive proof/synthesis in higher order logic [13, 9, 6].

The deductive counterpart for our functional formalism is the Boyer-Moore logic [4]. In a preliminary case study [21] we applied DDD to Hunt's FM8501 microprocessor description [14]. Naturally enough, our conclusion was that the formal exercise of proving FM8501 would be more effectively done with a combination of deductive and derivational steps. Except for two features, we found that a formal derivation of the FM8501 was attainable.

First, we could not derive Hunt's correctness results for binary representations of arithmetic. Although this can be done an algebraic formalism (but not in DDD) we are convinced that deductive reasoning is more natural. This thesis is developed in [38].

Second, we could not derive the portion of Hunt's architecture that performs synchronization with memory. Although it was encoded by the logical representations, Hunt's proof involved a translation of the functional abstraction of memory used in the specification to to the process abstraction used in the implementation. This was not a particularly complex entity and we felt that DDD should be capable of exposing it. That observation led to the research of this proposal.

1.3.2 Process Formalisms

Process algebras, such as Milne's CIRCAL [26] and Gopalakrishnan's HOP [11] as well as synchronous calculi like LUSTRE [32] and SBL [30], are natural candidates for modeling sequential interaction. Even though these systems are algebraic [27], the research using them deals almost exclusively with post-design verification. The algebra is used to compose processes rather than to decompose them.

The literature in this area is full of complex examples that would be unreasonable to obtain by derivational methods. In [11] for example, Gopalakrishnan considers a pipelined memory in which a cache allows reads to overtake writes. In [1], Bickford and Srivas use a functional formalism to verify a pipelined microprocessor. Their problem was to identify the microstate in which an individual instruction completes, given that the next instruction has already started. It is unlikely that these examples could be systematically derived from a naive memory model. Each involves a nontrivial relationship between event structures at two levels of abstraction.

These examples illustrate that constructive methods cannot subsume ingenious design. But our experience is that most proof tasks are not ingenious; they could have been developed through mundane constructions.

1.3.3 Processes in Synthesis

The mainstream of research in high-level synthesis (HLS) considers compilation of architecture from single-threaded specifications of control [7]. However, elements of sequential decomposition impose themselves at this level. Input-output events may be bound to fixed steps of a DFG schedule; some HLS systems understand operations that span time steps; optimization algorithms are being adapted to multiple computation branches. Nevertheless, HLS systems—and their languages—address design at the individual-process level with marginal consideration of procedural abstraction (subroutines, coroutines, etc.).

In the introduction we mention Borriello's recent work in interface specification [2]. Our research seeks to integrate a level of data abstraction with the notion "interface."

1.3.4 Self Timing, Etc.

Various methods exist to deal transparently with the process synchronization. Thankfully, a review of literature is not appropriate for this proposal; but we have found cogent discussions and surveys in [8] and [31]. These schemes simplify sequential interaction but do not eliminate the need to specify and reason about coordinated systems. For example Martin's self-timed processor [23] must still be proved safe, live, and correct with respect to the instruction set it implements. Our research assumes a level of atomic communication exists and considers concurrency beyond that level.

1.4 Foundations of the Proposed Research

This section outlines how we represent hardware description and design in a first-order functional algebra. We review only enough of the notation and methodology for the explanation of proposed research that follows in the next section.

1.4.1 Type Structures

Any first-order system represents a family of languages, each built from an underlying vocabulary called the *basis*, or *underlying type*. Rules of reasoning partition into metarules, which hold irrespective of the basis, and base laws, which depend on it. While support of the metarules is built in, the system is, in effect, programmable with respect to the base laws.

The basis refers to a *type system*, a collection of value domains together with constants, operations, and tests on these domains; and a collection of equational laws relating the operations. For example, a parameterized description of "memory," in terms of addresses and contents, is:

SORTS	OPERATIONS	LAWS
adr	$\mathsf{rd}\colon (\mathsf{mem}, adr) \to con$	$\mathsf{rd}(\mathsf{wt}(m, a, c), a) = c$
con	$\mathsf{wt}\colon (\mathsf{mem}, adr, con) \to \mathsf{mem}$	$\mathsf{rd}(\mathsf{wt}(m, a, \mathsf{rd}(m, a)) = m$
mem		*etc.*

The use of data abstraction involves *both* reasoning within the basis *and* translation between bases. We might represent Stacks with address-memory pairs and implement stack operations Memory *combinators*:

$$\begin{array}{lcl} top\langle a, m\rangle & \Leftarrow & \mathsf{rd}(m, a) \\ push(c, \langle a, m\rangle) & \Leftarrow & \langle a+1, \mathsf{wt}(m, a, c)\rangle \\ pop\langle a, m\rangle & \Leftarrow & \langle a \dot{-} 1, m\rangle \\ & etc. & \end{array}$$

In the general case, combinators would be replaced by recursive expressions. We should be able to prove that the Stack identities, such as "$pop(push(c, s) = s$" are preserved by the translation. The subtle semantic issues are summarized in [25], with more recent developments in [28].

However, our research is more concerned with *using* a correct implementation than with proving it correct. A specification in terms of Stack must translate to an expression in terms of Memory for a derivation of hardware. We consider the problems of implementing type-translation in [38]?

1.4.2 Control Oriented Expressions

The behavioral aspect of specification is represented by *recursion equations:* simultaneous systems of function definitions, $\{F_i(\bar{x}) \Leftarrow \mathcal{E}_i\}$. Defining expressions $\mathcal{E}_i$ are composed of base terms, recursive applications, and simple conditional forms, such as *if-then-else* and *case* expressions. An algorithm to compute *factorial* will be used as an example throughout:

$$\boxed{\begin{array}{l} F(n,1)\ \textit{where} \\ \quad F(u,v) \Leftarrow (u=0)\ \to\ v, \\ \qquad\qquad\qquad\qquad\quad F(u-1, u*v) \end{array}} \qquad (1)$$

A tail-recursive call, $F_j(t_1, \ldots, t_n)$ is a concurrent update of the data state $\bar{x} = (x_1, \ldots, x_n)$ by the base terms $\bar{t} = (t_1, \ldots, t_n)$, with a transfer to control-state F_j. We call $\bar{t}$ a *computation step.*

Control manipulations are formalized as *fold/unfold* transformations. Applications to hardware derivation are described in [3] and [36]. In [37], we also consider the interplay of control manipulation and data translation.

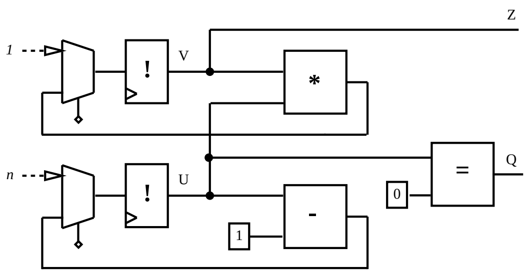
1.4.3 Structure Oriented Expressions

The structural aspect is represented by *sequential systems*: simultaneous systems, $\{X_i = \mathcal{S}_i\}$, where the expressions $\mathcal{S}_i$ denote infinite sequences. We use D^* to denote the computable sequences of D. Sequence-defining expressions, $\mathcal{S}_i$, are base terms over the variables X_j in which operations extend to element-wise maps. There is an delay operator, '!', which plays the role of a register.

$$v \,!\, \langle s^0,\ s^1,\ \ldots\rangle = \langle v, s^0,\ s^1,\ \ldots\rangle$$

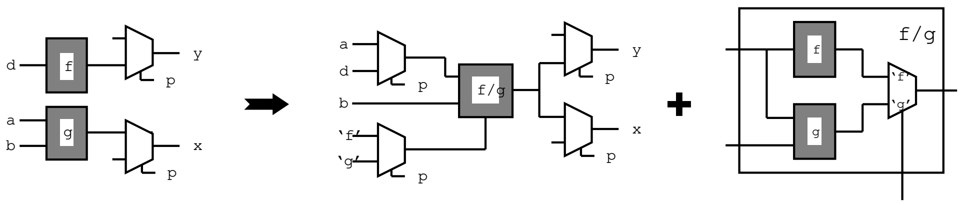
The *factorial* algorithm (1) corresponds to a sequential system composed of exactly the same terms:

$$\boxed{\begin{array}{l} (Q, Z)\ \textit{where} \\ \quad U = n \,!\, U - 1 \\ \quad V = 1 \,!\, U * V \\ \quad Q = \quad\ (U = 0) \\ \quad Z = \quad\ V \end{array}} \tag{2}$$



The equation for V gives a trace of parameter v, $V = \langle 1, n, n(n-1), \ldots, F(n,1), 0, 0, \ldots\rangle$. So (2) computes (1) in the sense that sequence Z has value $F(n,1)$ as soon as Q becomes true. Such a system can be mechanically constructed from any *iterative* (i.e. fully tail-recursive) recursion equation and conversely [16].

Structural manipulations under fixed control are composed of elementary transformations described in [18]. A *system factorization* combines two or more operations into a single module by distributing selection and control through them. As illustrated below, a factorization produces a modified system *and* a new module definition, consistent with the original configuration.



In application, this factored module is generally abstract, relative to the residual system, and so acts as a behavioral specification. Factorizations develop implementation architecture [20]; provide a way to encapsulate complex data objects [17, 15]; and can generate certain kinds of environmental constraints [19]. Incorporating registers in a factorization gives a trivial form of sequential decomposition.

1.4.4 Communication

Hardware behaviors are not just functions from values to values. The canonical specification has signature:

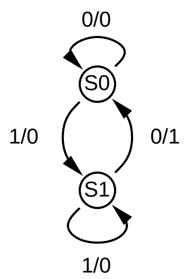
$$\mathcal{F}\colon (D^*)^p \times D^q \to (D^*)^r$$

mapping p input sequences, call them *ports*, and an initial data state, to r output ports. We can still use recursion equations for specification, incorporating ports as a standard extension of the basis [20]. For brevity we will use a diacritical convention for input ports. If $\hat{p}$ names a port, then $\acute{p}$'s stands for its current value and $\grave{p}$ its future; that is, $\hat{p} = (\acute{p} \,!\, \grave{p})$. For example, a *single pulser* outputs a unit pulse for every input pulse [29]. Such a device could be specified as

$$\begin{array}{rcl} SP_0(\hat{p}) \Leftarrow (\acute{p} = 0) & \rightarrow & 0 \,!\, SP_0(\grave{p}) \\ & & 0 \,!\, SP_1(\grave{p}) \\ SP_1(\hat{p}) \Leftarrow (\acute{p} = 0) & \rightarrow & 1 \,!\, SP_0(\grave{p}) \\ & & 0 \,!\, SP_1(\grave{p}) \end{array}$$

Thus, $\mathrm{SP}_0\langle 0,0,1,1,1,0,0,\ldots\rangle = \langle 0,0,0,0,1,0,0,\ldots\rangle$. If we restrict our attention to iterative processes that communicate in every computation step, we can abbreviate further, replacing $\hat{p}$ and $\grave{p}$ by *input* declaration:

$$\begin{array}{l} \hat{p}\text{: } \mathit{input} \\ \hat{q}\text{: } \mathit{output} \\ S_0() \Leftarrow (\acute{p} = 0) \;\rightarrow\; 0 \,!\, S_0() \\ \qquad\qquad\qquad\qquad\;\; 0 \,!\, S_1() \\ S_0() \Leftarrow (\acute{p} = 0) \;\rightarrow\; 1 \,!\, S_0() \\ \qquad\qquad\qquad\qquad\;\; 0 \,!\, S_1() \end{array}$$



We might do something similar for output, but it would be too suggestive of asynchronous process languages, such as CIRCAL [26] and HOP [11]. What we want is much closer to the value-passing form of *synchronous CCS* [27]. At low levels of data abstraction, there is also an obvious resemblance *RTFSM*s [22] and similar automata models.

1.5 An Approach to Sequential Decomposition

We have been able to discover no standard vocabulary for talking about the communicative relationship between processes. We shall use the term *interaction* for the whole pattern of communication, *synchronization* for the coordination of control, and *transaction* for the data exchange protocol. We use "protocol" informally.

We will continue to use *factorial* as an illustration. Let $\mathcal{B}$ and $\mathcal{N}$ denote the binary and natural numbers, respectively. Recall that the original expressions was

$$\boxed{\begin{array}{rcl} F(u,v) \Leftarrow (u = 0) & \rightarrow & v, \\ & & F(u-1, u*v) \end{array}} \tag{1}$$

1.5.1 Interaction Schemes

Expression (1) would be more meaningful if, in addition to telling how to compute the function, it also described an interaction with the environment. We should be able to express this interaction independently of the computation performed.

Let $f\colon\mathcal{N}\to\mathcal{N}$ be a unary function; let '?' designate an indeterminate condition; let 'ϕ' stand for an undetermined value (don't-care) of any sort; and consider the scheme[3] $P[f]$ mapping $\mathcal{B}^*\times\mathcal{N}^*$ to $\mathcal{B}^*\times\mathcal{N}^*$

$$\boxed{\begin{array}{l} P(f)(\hat{g},\hat{n}\colon input,(\hat{z},\hat{q})\colon output)\Leftarrow\\ \quad \begin{array}{lll} IDLE(v)\Leftarrow(\acute{g}=0) & \to & (\phi,0)\;!\;BUSY(\acute{n}),\\ & & (\phi,1)\;!\;IDLE(\phi)\\ BUSY(v)\Leftarrow\,?\!\to & & (f(v),1)\;!\;IDLE(\phi),\\ & & (\phi,0)\;!\;BUSY(v)\end{array}\end{array}} \tag{2}$$

IDLE waits for a "go" signal on $\hat{g}$, reads a value v from port $\hat{n}$, eventually computes $f(v)$, and signals completion on port $\hat{q}$. Our goal is to instantiate (2) using (1). A possible instance of $P[F]$ is

$$\boxed{\begin{array}{l} \quad\hat{g},\hat{n}\colon input\\ (\hat{z},\hat{q})\colon output\\ \quad \begin{array}{lll} G(u,v)\Leftarrow(\acute{g}=0) & \to & (v,0)\;!\;H(\phi,v),\\ & & (v,1)\;!\;G(\acute{n},1)\\ H(u,v)\Leftarrow(u=0) & \to & (v,1)\;!\;G(\phi,v),\\ & & (v,0)\;!\;H(u-1,u*v)\end{array}\end{array}} \tag{3}$$

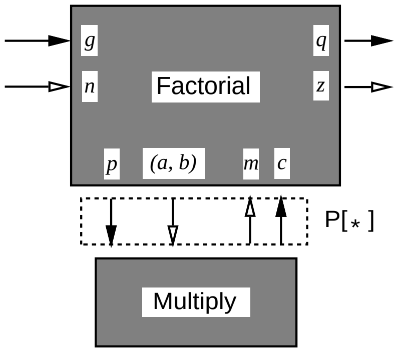
Note that this is not the minimal instance because v is displayed in every state. Of course, this is just one side of the interaction. Suppose a sequential multiplier is incorporated using the same interaction scheme, with operands passed in parallel. A *complementary* interaction must be performed:

$$\boxed{\begin{array}{l} \bar{P}(f)(\hat{z},\hat{q}\colon input,(\hat{g},\hat{n})\colon output)\Leftarrow\\ \quad \begin{array}{lll} \overline{IDLE}(v)\Leftarrow\,? & \to & (\phi,0)\;!\;\overline{IDLE}(v)\\ & & (v,1)\;!\;\overline{BUSY}(\phi)\\ \overline{BUSY}(v)\Leftarrow(\acute{q}=0) & \to & (\phi,0)\;!\;\overline{IDLE}(f(v)),\\ & & (\phi,1)\;!\;\overline{BUSY}(\phi)\end{array}\end{array}} \tag{4}$$

Assuming below that $\mathsf{Multiply}$ an an instance of $P[*]$, $\mathsf{Factorial}$ should be an instance of $\bar{P}[*]$,

[3] Actually, the $P[f]$ maps a pair-of-streams to a stream-of-pairs, $\mathcal{B}^*\times\mathcal{N}^*$ to $(\mathcal{B}\times\mathcal{N})^*$. A coercion is needed to resolve the inconsistency.

with respect to its single occurrence of '$*$':



The expression below is one possible solution.

$$\boxed{\begin{array}{l} g, n, p, m\text{: } input \\ z, q, (a, b), c\text{: } output \\ G(u,v) \Leftarrow (\acute{g} = 0) \;\rightarrow\; (v, 0, \phi, 0)\; !\; H(\phi, v), \\ \qquad\qquad\qquad\qquad\quad (v, 1, \phi, 0)\; !\; G(\acute{n}, 1) \\ H(u,v) \Leftarrow (u = 0) \;\rightarrow\; (v, 1, \phi, 0)\; !\; G(\phi, v), \\ \qquad\qquad\qquad\qquad\quad (v, 0, (u,v), 1)\; !\; J(u,v) \\ J(u,v) \Leftarrow (\acute{p} = 0) \;\rightarrow\; (v, 0, \phi, 0)\; !\; H(u-1, \acute{m}), \\ \qquad\qquad\qquad\qquad\quad (v, 0, \phi, 0)\; !\; J(u,v) \end{array}} \tag{5}$$

In summary, we have illustrated a process of factorization in which objects are transformed according to interaction schemes. The schemes are abstractions of sequential computations describing synchronization and value-passing protocol, while suppressing other details. The object is to incorporate complementary views of the interaction in two cooperating processes. The system should guarantee that if the two views are, in fact, complementary, the resulting interaction is correct. Each of the factors is a refined behavioral description, and hence can be manipulated by the derivation algebra described earlier do derive hardware implementations.

1.6 Research Topics

In this section we outline specific research topics implied by the approach just described. Our twofold objective is to automate useful techniques while at the same time developing a broader theoretical characterization of what is going on. We discuss the theory first and then describe implementation and experimentation.

1.6.1 Formalization of Interaction

Semantics of Interaction. We have only an informal notion of what it means to be an instance of a interaction scheme. Intuitively, it is an embedding of one control/communication structure in another. One of our first tasks is to define this notion rigorously. The conventional approach is to develop a semantics of event structures. It is clear that neither complementing nor instantiating an interaction scheme gives a unique expression. The semantics must give a way to verify this property.

Complexity of Factorization. It is intractable to determine how an arbitrary interaction scheme embeds in an arbitrary algorithm. To have any hope of mechanical support, we must restrict to something less general, and perhaps subregular. Since a limited number of distinct handshakes are used in practice, such a restriction would not be a hardship for two-party synchronization. However, manipulating interactions will entail some notion of composition (as discussed next). The research must therefore identify classes of schemes with desirable closure properties.

Parallel Decomposition. The *factorial* example illustrates a simple coroutine decoupling technique. In actual design, sequential decomposition is usually done for the purpose of increasing parallelism. Of the many facets of this problem, the following three are worthy of individual study:

- *Concurrent computation*: overlapped execution of coroutines.
- *Collateral computation*: parallel operations in a computation step.
- *Time-space coercion*: e.g. serial *vs.* parallel data transactions.

We believe that each of these aspects can be formalized using the approach we have described; However, the integration of these and other techniques is probably beyond the reach of a two-year proposal. By the end of the grant period, we will have explored transformations for classes of parallel decomposition and expect to have a proposal for a more comprehensive treatment.

Constraints. Interaction schemes will not be sufficient for all classes of timing constraints. It is a well known problem to incorporate constraint specification in a behavioral specification dialect. Our current thinking is that constraint specification and should be orthogonal to behavior description, just as termination is separated from correctness in program proofs.

Clearly, adding an interaction affects both local and global event structures. It is expecting too much for a system of transformations to preserve all possible properties of such structures. On the other hand, we have shown that transformations can generate useful side conditions for post-design verification [19]. This is the approach we will explore first.

Resource Sharing. In control dominated designs it is common for the data state to be shared among processes. For example, a typical communications interface consists of several processes operating on a queue. Our preliminary notion of interaction must be extended to account for this quality.

Alternative Approaches. The proposed model of interaction fixes a level of atomic communication. Interaction schemes transmit the same kinds of values that their instances do. Another possibility is to allow certain structures in value sequences to represent interactions. For example, the "value" oriented description of a multiplier would take finite sequences ($a\phi^k$, $b\phi^k$) to $\phi^k(a * b)$ *for some k*.

Under this representation, the goal of transformation would be to flatten the value hierarchy by introducing synchronization. Currently, this approach is more intriguing than promising, but it has some attractions over the control oriented specifications used earlier. For example, it seems better at reflecting the concept of self-timed communication. We will devote some time in our research to exploring it.

1.6.2 Implementation and Experimentation

Results having to do with interaction schemes will be incorporated in the DDD transformation system. We will use exercise these enhancements by applying DDD at the system-specification level.

Since this research branches for work in semantics and programming methodology, and is still affiliated with it, our experimentation usually involves language oriented processor design. The processing unit itself is usually the simplest component in such a processor. The more interesting exercises involve storage and device management.

For example, one of our ongoing projects is to develop a self managing multiported heap, based on reference counting. This design, which involves dozens of processes operating on a heap organization, is a nontrivial example of control dominated design.

However, these proof-of-principle experiments will probably not result in hardware products. During the period of this project, we hope to demonstrate that DDD is of general value as a system for ASIC design.

References

[1] Mark Bickford and Mandayam Srivas. Hardware verification using clio. In M. Leeser and G. Brown, editors, *VLSI Specification, Verification and Synthesis: Mathematical Aspects*, New York, July 1989. Proceedings of Mathematical Sciences Institute work shop, Cornell University, Springer. Lecture Notes in Computer Science, Vol-408.

[2] Gaetano Borriello. Specification and synthesis of interface logic. In R. Camposano and W. Wolf, editors, *High-level synthesis*, chapter 7, pages 153–176. Kluwer, Boston, 1991.

[3] C. David Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In John A. Darringer and Franz J. Rammig, editors, *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages*, pages 235–246. North-Holland, 1990.

[4] Robert S. Boyer and J. Struther Moore. The user's manual for a computational logic. Technical Report 18, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, Texas 78703, February 1988.

[5] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. Proceedings of the 1991 ACM/IFIP Workshop on Formal Methods for VLSI, in preparation.

[6] Holger Busch. Proof based transformation of formal hardware models. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990*, pages 271–296, London, 1991. Springer.

[7] Raul Camposano and Wayne Wolf, editors. *High-level* VLSI *Synthesis*. Kluwer, Boston, 1991.

[8] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, Department of Computer Science, Stanford CA 94305 USA, October 1984. Report No. STAN-CS-84-1026.

[9] S. Finn, M. P. Fourman, M. D. Francis, and R. L. Harris. Formally based system design – interactive synthesis based on computer assisted formal reasoning. In L.J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, volume 1, pages 139–152. North Holland, 1990.

[10] Daniel D. Gajski. Essential issues and possible solutions in high-level synthesis. In R. Camposano and W. Wolf, editors, *High-level synthesis*, chapter 1, pages 1–26. Kluwer, Boston, 1991.

[11] Ganesh Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In J. Darringer and F. Ramming, editors, *Proceedings of the IFIP WG 10.2 Ninth*

International Symposium on Computer Hardware Description Languages, Amsterdam, 1989. Elsevier. in preperation.

[12] M. Gordon. HOL - a machine oriented formulation of higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1987.

[13] F. K. Hanna, N. Daeche, and M. Longley. Veritas$^+$: A specification language based on type theory. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 358–379, New York, July 1989. Springer. (LNCS 408).

[14] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985. Also published as Technical Report 47 (December, 1985).

[15] Steven D. Johnson. Applicative programming and digital design. In *Proceedings Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 218–227, 1984.

[16] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, Cambridge, 1984.

[17] Steven D. Johnson. Digital design in a functional calculus. In G. Milne and P. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–178. North-Holland, Amsterdam, 1986.

[18] 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. Proceedings of the Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag, July 1989.

[19] Steven D. Johnson and Bhaskar Bose. DDD – a system for mechanized digital design derivation. Proceedings of the 1991 ACM/IFIP Workshop on Formal Methods for VLSI, in preparation.

[20] Steven D. Johnson, Bhaskar Bose, and C. David Boyer. A tactical framework for digital design. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988.

[21] Steven D. Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In L.J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, volume 1, pages 117–136. IMEC, North-Holland, 1990.

[22] Miriam Leeser and Wayne Wolf. Behavior FSMs for high-level verification and synthesis. In P. A. Subrahmanyam, editor, *Proceedings of the ACM/SIGDA International Workshop on Formal Methods in VLSI Design*. Springer, in preparation.

[23] Alain J. Martin. The design of a delay-insensitive microprocessor: An example of synthesis by program transformation. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 244–259, Berlin, 1990. Springer.

[24] Thomas F. Melham. *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic*. PhD thesis, University of Cambridge, 1990. published as University of Cambridge Computer Laboratory Technical Report No. 201.

[25] José Meseguer and Joseph A. Goguen. Initiality, induction, and computability. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 14, pages 459–542. Cambridge University Press, Cambridge, 1985.

[26] G.J. Milne and M. Pezze. Typed CIRCAL: A high level framwork for hardware verification. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 117–138. North-Holland, 1988.

[27] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, 1989.

[28] Lawrence S. Moss, José Meseguer, and Joseph A. Goguen. Final algebras, cosemicomputable algebras, and degrees of unsolvability. preprint series IULG-91-10, Indiana University Logic Group, Department of Philosophy, Bloomington, Indiana, July 1991.

[29] Franklin P. Prosser and David E. Winkel. *The art of digital design*. Prentice-Hall, Englewook Cliffs, 2 edition, 1987.

[30] R. C. Sekar and M. K. Srivas. Formal verification of a microprocessor using equational techniqes. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 4, pages 171–217. Springer, New York, 1989.

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

[32] Ghislaine Thuau and Daniel Pilaud. Using the declarative language LUSTRE for circuit verification. In G. Jones and M. Sheeran, editors, *Designing Correct Circuits, Oxford 1990*, pages 313–331, London, 1991. Springer.

[33] Ranga Vermuri. A formal model for register transfer level structures and its application to verification and synthesis. In L.J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, volume 1, pages 61–82. North Holland, 1990.

[34] Robert M. Wehrmeister. Derivation of an SECD machine: Experience with a transformational approach to synthesis. Technical Report 290, Computer Science Department, Indiana University, September 1989.

[35] Daniel Weise. constraints, abstraction and verification. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 25–39, Berlin, 1990. Springer.

[36] Zheng Zhu and S. D. Johnson. An algebraic characterization of structural synthesis for hardware. In L.J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, volume 1, pages 261–269. North Holland, 1990.

[37] Zheng Zhu and S. D. Johnson. An algebraic framework for data abstraction. In Geraint Jones and Mary Sheeran, editors, *Proceedings of the International Workshop on Designing Correct Circuits*, 1990.

[38] Zheng Zhu and S. D. Johnson. An example of interactive hardware transformation. In *Proceedings of the ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. To Appear.