To appear in *Science of Computer Programming*, 2002.

The Semantics and Execution of a Synchronous Block-Diagram Language

Stephen A. Edwards [1]

Columbia University, 1214 Amsterdam Avenue, New York, NY, 10027, USA [2]

Edward A. Lee [3]

University of California, 518 Cory Hall, Berkeley, CA 94720, USA

Abstract

We present a new block diagram language for describing synchronous software. It coordinates the execution of synchronous, concurrent software modules, allowing real-time systems to be assembled from precompiled blocks specified in other languages. The semantics we present, based on fixed points, is deterministic even in the presence of instantaneous feedback. The execution policy develops a static schedule—a fixed order in which to execute the blocks that makes the system execution predictable.

We present exact and heuristic algorithms for finding schedules that minimize system execution time, and show that good schedules can be found quickly. The scheduling algorithms are applicable to other problems where large systems of equations need to be solved.

Key words: Heterogeneity, Synchronous, Software Modules, Execution, Fixed points, Embedded systems, Coordination Language, System of Equations, Relaxation, Chaotic Iteration

1 Introduction

The need for new techniques for designing software in embedded systems continues to grow as hardware costs plummet. Software is coming to dominate these systems, yet most of it is still written using ad hoc techniques in languages designed for batch processing systems. Such techniques do not address concurrency and real-time constraints, two of the more challenging aspects of much embedded

[1] *E-mail address:* sedwards@cs.columbia.edu
[2] This work was done while the author was at the University of California, Berkeley
[3] *E-mail address:* eal@eecs.berkeley.edu

software. In this paper, we present a new coordination language better tuned to the problem of assembling efficient, predictable software for these systems.

Most embedded systems operate in real time, so *when* they perform a task is as important as the task itself. Digital logic designers have long built clocked systems to control when events occur, but only recently has this paradigm become available to software designers in the form of the so-called synchronous languages [4,27], which include Esterel [11], Argos [36,37], Lustre [19,28], and Signal [31]. These provide powerful primitives for synchronizing parts of a system to external inputs.

This paper proposes a synchronous coordination language that allows systems to be assembled from pieces specified in different languages. This allows each system design problem to be solved using the language best-suited for it, and improves reuse possibilities. Furthermore, the coordination language and its scheduling techniques can be used as a foundation for designing other languages.

Our systems consist of synchronously communicating blocks. Like all synchronous languages, it adopts a model of time like that used in synchronous digital circuits: in each cycle of the global clock, the system examines inputs from the environment, evaluates the function of the system (which depends both on those inputs and the system's state), and produces outputs that are sent to the environment or used to determine the state of the system in the next cycle. Within each cycle, the blocks communicate instantaneously (i.e., information generated in a cycle can be observed in the same cycle), and no restrictions are placed on their topology. In particular, instantaneous feedback is allowed.

Each block must be able to both evaluate its outputs based on its inputs and advance its state. Splitting these is necessary because a block may need to be evaluated more than once per cycle if it appears in a feedback loop. Blocks must not make any assumptions about the number of times they are evaluated since it is an unpredictable function of the whole system and the scheduling algorithm. In contrast, the state of each block is advanced exactly once per cycle after its outputs have been determined. As mentioned earlier, this may make the block compute a different function in each cycle.

The main contribution of this coordination language is the ability to execute such systems without the compiler requiring information about the contents of the blocks, allowing them to be described in different languages. Provided each block uses the same communication protocol and behaves monotonically (never "changes its mind" when presented with additional information), the systems are deterministic, deadlock-free, and can be executed efficiently and predictably.

The remainder of the paper is divided into two parts. The first half formally defines the semantics of these systems as the unique least fixed point of the function of all the blocks, thus proving the language is deterministic. The second half introduces techniques for scheduling and ultimately executing these systems in compliance with the semantics. Experimental results and a discussion of future work concludes the paper.

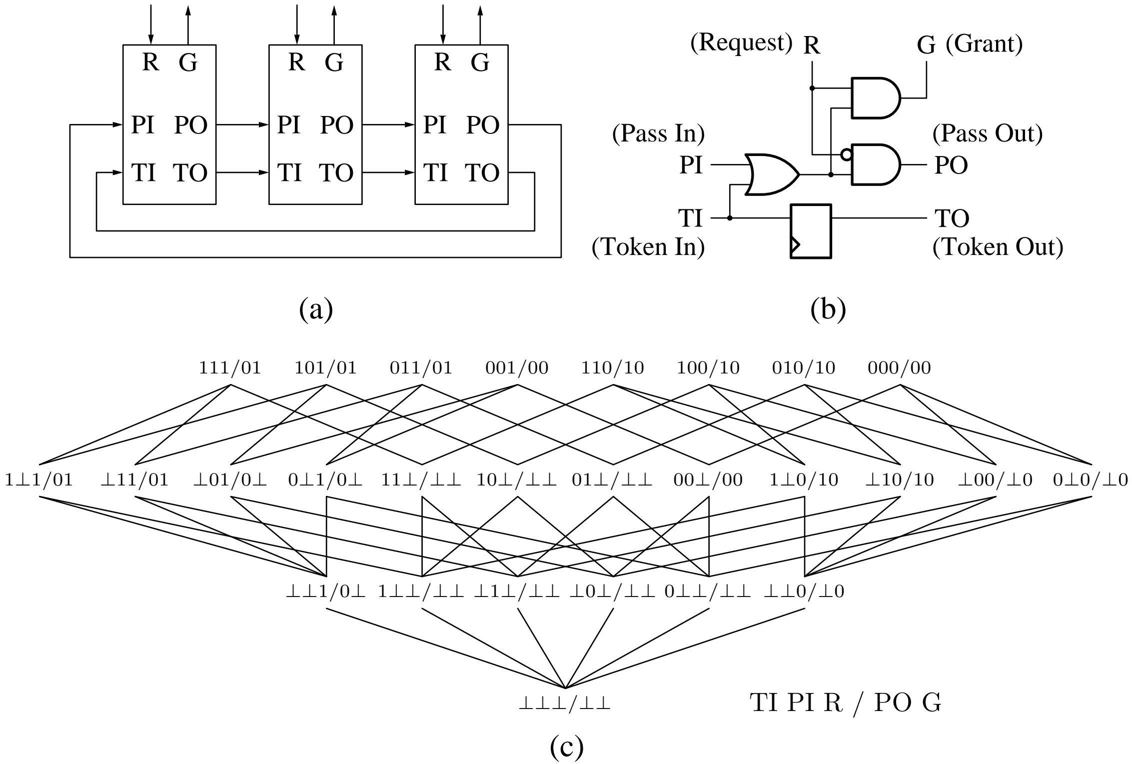

Fig. 1. (a) A cyclic token-ring arbiter composed of three blocks. (b) The function within each block. (c) A truth table for the function arranged according to the number of defined inputs. A line indicates a single input becoming defined.

2 Synchronous Block Diagrams

Our block diagram language is based on ideas from traditional zero-delay three-valued circuit simulation. Blocks compute a function of their inputs and communicate through zero-delay "wires" that convey values such as 0, 1, or undefined (we use $\perp$ to represent this unknown value). Time is divided into a sequence of clock cycles, and in each cycle each block sets the values on its outputs depending on the value it sees on its inputs and the state of the system. Wires communicate instantaneously, i.e., when a block sets the value of an output wire, all blocks connected to that wire see the new value in the same clock cycle. The number and connectivity of the blocks does not change while the system runs.

The cyclic token-ring arbiter in Fig. 1 is typical of the systems that can be described with our block-diagram language.[4] This system arbitrates fairly among requests for exclusive access to a shared resource by marching a token around a ring. In each cycle the arbiter grants access to the first requestor to the right of the block with the token. Fig. 1b shows the function of each block, which passes the token around the ring and either responds to a request or passes its opportunity to its right neighbor. At all times, exactly one of the latches stores a 1; the rest contain a 0.

[4] Berry [9] attributes this example to Robert de Simone.

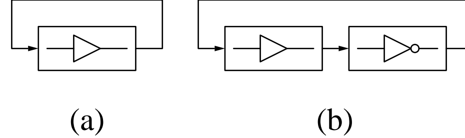

Fig. 2. Semantic challenges: (a) An ambiguous system. (b) A paradoxical system.

It appears this system might deadlock since the PO output depends on the value of PI, which comes from the PO output of the block to its left, and so on around the ring. The presence of the "token"—actually a 1 value on one of the latches—breaks this deadlock by setting to 1 the TI input of the block immediately to the right of the token. The presence of this 1 establishes the output of the OR gate independently of the value of PI, breaking the deadlock situation.

The way the cyclic arbiter breaks its deadlock is typical: the feedback loop contains non-strict blocks that can be "short-circuited" to ignore inputs from the loop when certain patterns are applied to their inputs, thus breaking the deadlock. For a system to be deadlock-free it is necessary (but not sufficient) for each feedback loop to contain at least one non-strict block—one that can produce a partially-defined output in response to a partially-defined input. A three-valued OR gate is typical of a non-strict block: if one of its inputs is 1, its output is 1 regardless of the value on the other input. Similarly, AND gates, multiplexers, and delay elements are all non-strict. Functions that always require all their inputs, such as inverters and exclusive-OR gates, are strict. A feedback loop containing only exclusive-OR gates will always deadlock.

The main objective of our block-diagram language is to handle systems like the cyclic arbiter that may appear to deadlock but do not because of behavioral details. Specifically, we are able to define the semantics of and simulate such systems without detailed knowledge of the functions computed by each block. This is useful in software systems linked together from pieces that are compiled separately or whose blocks are specified using different languages. We used a circuit diagram to define the function of the blocks in this example, but could just have easily used a synchronous language such as Esterel.

Systems with paradoxes and ambiguity, such those in Fig. 2 have a natural interpretation in this framework: the undefined value $\perp$ appears on wires participating in unbroken feedback loops. For example, the system with the single buffer in Fig. 2a appears to be satisfied with any value on its single wire, but our deterministic semantics declare the undefined value $\perp$ to be the only correct solution. Similarly, the paradoxical system in Fig. 2b seems to have no satisfying assignment. However, since the inverter must produce $\perp$ in response to a $\perp$ input, our semantics say both wires take the value $\perp$.

In the remainder of this section, we put the semantics of our language on firm mathematical ground by defining it as the least fixed point (LFP) of the function of all the blocks and using a well-known fixed point theorem to show that this is unique. The second half of the paper discusses how to efficiently evaluate this LFP.

2.1 Semantics

We base the semantics of our systems on three main concepts. First, the values passed through wires are taken from a complete partial order—a set whose elements are ordered by how much "information" each contains. Second, the blocks are restricted to compute functions that are monotonic with respect to this order, so they never decrease or change existing information when presented with additional information. Finally, a well-known theorem guarantees that such monotonic functions have a unique least fixed point, which we define as the behavior of the system in each clock cycle.

Our coordination language permits the unknown value, written $\bot$, on a wire, which is used to represent wires in feedback loops with ambiguous or contradictory values. Formally, each wire takes a value from a partially-ordered set V with a binary relation $\sqsubseteq$ that is reflexive ($x \sqsubseteq x$), antisymmetric (if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$), and transitive (if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$). We construct such sets by "lifting" a set. Starting with a set V' of defined values such as $\{0,1\}$ or the integers, lifting V' adds the undefined element $\bot$ (i.e., $V = \{\bot\} \cup V'$) and imposes the order $\bot \sqsubseteq \bot$, $\bot \sqsubseteq v'$, and $v' \sqsubseteq v'$ for all $v' \in V'$. This order leaves distinct members of the set V' incomparable, e.g., neither $0 \sqsubseteq 1$ nor $1 \sqsubseteq 0$.

The $\sqsubseteq$ relation naturally extends to vectors ($(a_1,\ldots,a_n) \sqsubseteq (b_1,\ldots,b_n)$ iff $a_1 \sqsubseteq b_1, a_2 \sqsubseteq b_2, \ldots,$ and $a_n \sqsubseteq b_n$) and imposes an information ordering in the sense that if $a \sqsubseteq b$, then there are only two possibilities for corresponding elements of a and b: they can be identical, or $a_k = \bot$ and $b_k \in V'$.

To ensure deterministic semantics, we require that each block compute a monotonic function of its inputs (i.e., a function F for which $x \sqsubseteq y$ implies $F(x) \sqsubseteq F(y)$). This has a natural interpretation: presenting a block with a more-defined input always produces a more-defined output or the same value.

Fig. 1c is an oddly-drawn truth table for the function of an arbiter block that shows it is monotonic. Input/output pairs are separated by a slash and arranged such that following a line upward always leads to a more defined input. Careful inspection of the diagram will show that the outputs also always become more defined along an upward path, implying the function is monotonic. For example, the rightmost path is the sequence $\bot\bot\bot/\bot\bot \to \bot\bot 0/\bot 0 \to 0\bot 0/\bot 0 \to 000/00$.

The fixed-point theorem operates on a totally-ordered sequence called a chain, i.e., a set $C \subseteq V$ such that $x \sqsubseteq y$ or $y \sqsubseteq x$ for all $x, y \in C$. The maximum length of these chains is important, so we define the *height* of a partially-ordered set V as the size of the longest chain in V. A lifted set that represents the value on a single wire has height two, since the longest chains all look like $\{\bot, v'\}$ for some $v' \in V'$. The height of an n-valued vector of elements of V is $n+1$ (vectors in the longest chain have between 0 and n $\bot$ elements).

The fixed-point theorem we use also applies to sets with infinite-height chains, but this requires chains and functions to stay bounded. An upper bound $b \in V$ of a set $S \subseteq V$ satisfies $s \sqsubseteq b$ for all $s \in S$. The least upper bound, if it exists, is the unique element $\operatorname{lub} S$ such that $\operatorname{lub} S \sqsubseteq b$ for all upper bounds b. A complete partial order (bounded on infinite chains) is a set V that is partially ordered, has a distinguished

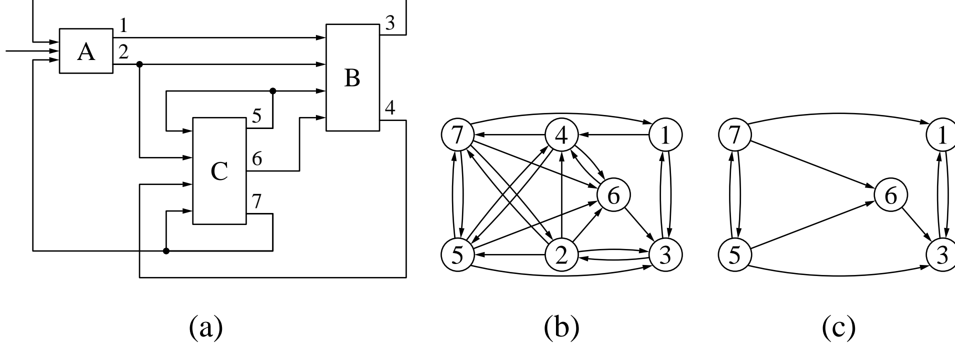

Fig. 3. (a) A system, (b) its dependency graph (each output is a node, an arc is drawn from each output driving an input on the block, self-loops are omitted), and (c) the dependency graph after removing nodes 2 and 4. Each node corresponds to an output and each arc represents a functional dependence.

bottom element $\perp$ such that $\perp \sqsubseteq v$ for all $v \in V$, and all chains $C \subseteq V$ have a least upper bound. A function $F : V \to V$ is continuous (bounded on infinite chains) if for all chains $C \subseteq V$, $F(\text{lub}\, C) = \text{lub}\{F(c) : c \in C\}$.

Since all the chains in our domains (i.e., finite vectors of lifted sets) are finite, our partial orders are complete because finite chains always have a least upper bound. Furthermore, it is not difficult to show that our monotonic functions, since they are defined on sets with only finite chains, are also continuous.

We define the semantics of our systems as the least fixed point of a function. A fixed point of a function F is simply a value x satisfying $F(x) = x$. For block diagrams, this corresponds to the case where the output of each block produces matching inputs. In short, the inputs and outputs of each block are consistent. A function may have many fixed points, but we are interested in the least-defined fixed point $\text{lfp}\, F$, which by definition satisfies $F(\text{lfp}\, F) = \text{lfp}\, F$ and $\text{lfp}\, F \sqsubseteq x$ for all fixed points x of F. A fixed point is attractive as a definition of behavior because it corresponds to the intuitive notion of a consistent state of the system. Furthermore, the least fixed point can be reached purely through deductive reasoning (i.e., it is unnecessary to make and test hypotheses to compute the least fixed point, which is not the case with other fixed points), and it is unique, making systems deterministic.

The key theorem that defines the semantics and guarantees determinism is a folk theorem variously attributed to Knaster, Tarski, and Kleene [30].

Theorem 1 *The least fixed point of a continuous function F on a complete partial order is unique and equal to the least upper bound of the chain* $\{\perp, F(\perp), F^2(\perp), \ldots\}$.

Finally we are in a position to define the semantics of our systems. The function for the system is derived from the functions of the blocks and their connectivity.

Consider the system in Fig. 3a. The functions of its three blocks are

$$A : I \times S \times V^2 \to V^2$$
$$B : I \times S \times V^4 \to V^2$$
$$C : I \times S \times V^4 \to V^3$$

where I is the set of all possible inputs to the system and S is the set of all possible states of the system. Although block A is drawn with three inputs, the A function is only defined on V^2 because only two of its inputs are connected to internal wires. The effect of the external input is felt through A's dependence on I; B and C are probably independent of I. This very abstract model of both inputs and system state is sufficient for our purposes. The semantics treats environment inputs and system state equivalently: they simply select the functions computed by the blocks. The only requirement is that A, B, and C be monotonic with respect to outputs.

The function of this system $G : I \times S \times V^7 \to V^7$ maps the input, state, and 7 current output values to a new set of 7 output values. We will define the semantics as the least fixed point of this function.

Each component of the vector-valued G function is an output of one of the blocks and is a component of one of the block functions. For example, G_1 is the function for output 1, which is the first output of block A. The two non-external inputs of block A are driven by outputs 3 and 7, so

$$G_1(i, s, v_1, \ldots, v_7) = A_1(i, s, v_3, v_7)$$

where A_1 is the function for the first output of block A.

The other component of the G function are defined similarly:

$$\begin{aligned}
G_2(i, s, v_1, \ldots, v_7) &= A_2(i, s, v_3, v_7) \\
G_3(i, s, v_1, \ldots, v_7) &= B_1(i, s, v_1, v_2, v_5, v_6) \\
G_4(i, s, v_1, \ldots, v_7) &= B_2(i, s, v_1, v_2, v_5, v_6) \\
G_5(i, s, v_1, \ldots, v_7) &= C_1(i, s, v_5, v_2, v_4, v_7) \\
G_6(i, s, v_1, \ldots, v_7) &= C_2(i, s, v_5, v_2, v_4, v_7) \\
G_7(i, s, v_1, \ldots, v_7) &= C_3(i, s, v_5, v_2, v_4, v_7)
\end{aligned}$$

In general, an n-output system implies a system function $G : I \times S \times V^n \to V^n$ constructed in this way. The behavior of the system in a cycle in state $s \in S$ with inputs $i \in I$ is the least vector $x \in V^n$ that satisfies

$$G(i, s, x) = x, \tag{1}$$

Theorem 2 *There is always a unique least x that satisfies (1), so these systems are deterministic.*

Proof. This follows from Theorem 1 because V^n is a complete partial order and $G(i,s,x)$ is continuous w.r.t. x because it is a vector-valued combination of the monotonic (and hence continuous because chains in V are finite) block functions. ∎

3 Execution

In each cycle, the semantics of our block diagram language requires us to find the least fixed point of G, the monotonic function describing the composition of all the blocks in the system. We compute this fixed point by evaluating the functions of the blocks in a particular order—a schedule—that guarantees that the final result is the least fixed point.

We obtain these schedules through a divide-and-conquer approach. The "conquer" part comes from the iteration in Theorem 1, which says the LFP of a function G can be computed by taking the least upper bound of the chain $\{\bot, G(\bot), G^2(\bot), \ldots\}$. Because chains in our domain (the vector of all block outputs) are finite, this reduces to evaluating G until a fixed point is reached. Specifically, an n-output system has chains of height $n+1$, so we are guaranteed to reach the LFP after n evaluations of G.

The "divide" part of our divide-and-conquer algorithm comes from Bekić's Theorem [3]:

Theorem 3 (Bekić) *Let $X : V^m \times V^n \to V^m$ and $Y : V^m \times V^n \to V^n$ be continuous functions on a complete partial order. Then the least fixed point of $X \times Y : V^m \times V^n \to V^m \times V^n$ is $(\hat{x}, \hat{y})$, where*

$$\hat{x} = \mathrm{lfp}_x X(x, \mathrm{lfp}_y Y(x,y)), \tag{2}$$

$$\hat{y} = \mathrm{lfp}_y Y(\hat{x}, y), \tag{3}$$

and $\mathrm{lfp}_x f(x,y)$ is a function of y, say $g(y)$, that is the least function that satisfies $f(g(y),y) = g(y)$.

This provides a mechanism for evaluating the least fixed point of a vector-valued function by breaking it into two, evaluating the least fixed point of the first half, then using the result to evaluate the second half. At first glance, this is not helpful since evaluating the LFP of the first half requires evaluating the LFP of the second half along the way. However, the computation does become substantially simpler when X does not depend on its second argument:

Corollary 1 *If $X(x,y)$ does not depend on y, then the least fixed point of $X \times Y$ is $(\hat{x}, \hat{y})$ where $\hat{x} = \mathrm{lfp}_x X(x,z)$, $\hat{y} = \mathrm{lfp}_y Y(\hat{x}, y)$, and z is an arbitrary vector in V^n.*

This implies that the LFP of a system with no feedback can be evaluated by calculating the LFP of the blocks in topological order, i.e., by evaluating the blocks that depend only on external inputs first, then by evaluating blocks that only depend on that set of blocks, and so forth.

To illustrate our scheduling and execution procedure, consider the three-block

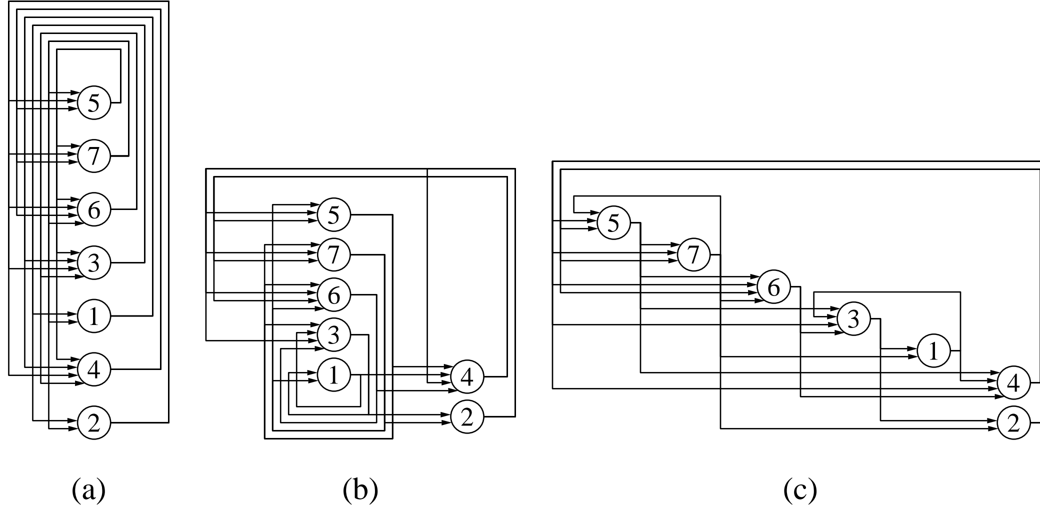

Fig. 4. Decomposing the dependency graph in Fig. 3b using Bekić's theorem. (a) A brute-force evaluation using Theorem 1 involves evaluating a feedback loop with seven wires. Cost: $7^2 = 49$ (b) Splitting it two using Bekić's theorem (the X function contains nodes 2 and 4, the others are part of the Y function) transforms the graph into an inner feedback loop with five wires and an outer loop with two. Cost: $2^2 + (2+1)5^2 = 79$ (c) Further decomposing the Y function transforms the five-element feedback loop into two loops (5 and 7, 3 and 1) of one wire each. Cost: $2^2 + (2+1)(3+1+3) = 25$

system in Fig. 3a. We represent its communication dependencies with the graph in Fig. 3b, whose nodes represent outputs. An arc is drawn from node x to node y if output y is on a block with an input connected to output x, e.g., there is an arc from node 5 to node 3 because output 5 drives block B, which produces output 3. Self-loops are omitted because they do not affect how rapidly a least-fixed-point computation converges. Specifically, if an output is $\perp$ when the input is $\perp$, the output is consistent. If the output is not $\perp$ when the input is $\perp$, the output must remain at that non-$\perp$ value when the feedback is taken into account because the functions are monotonic.

One way to evaluate the LFP of the system is to directly apply the iteration of Theorem 1. If we assume evaluating a single block output has a cost of 1 (which we do throughout this paper), then because the height of the set with all vectors of length 7 is 8 (these vectors represent the values on all the wires) we need to evaluate all block outputs 7 times. There are 7 block outputs, so evaluating the system function once has cost 7. Thus the total cost of evaluating the LFP of the system using a brute-force iteration is $7^2 = 49$.

Bekić's Theorem and Corollary 1 allow us to evaluate the LFP more cheaply. We cannot apply Corollary 1 at first because the dependency graph is strongly connected, i.e., there is a directed path in both directions between every pair of nodes, implying every function is dependent on every other. So we use Bekić's Theorem to evaluate the LFP.

Bekić gives us the freedom to divide the system function any way we like, so we will choose a decomposition with the goal of reducing the number of func-

tion evaluations. The algorithm we describe later (Section 3.3) suggests we first choose the X function to consist of nodes 2 and 4, and Y to contain the rest. This decomposition is shown in Fig. 4b, which is drawn to suggest the application of Bekić's theorem. First, we calculate $\hat{x} = \text{lfp}_x X(x, \text{lfp}_y Y(x,y))$ as follows. Starting with $\hat{x}_0 = \bot$, we evaluate $\hat{x}$ in two iterations using $\hat{x}_{k+1} = X(\hat{x}_k, \text{lfp}_y Y(\hat{x}_k, y))$, so $\hat{x} = \hat{x}_2$. In this computation, $\text{lfp}_y Y(\hat{x}_k, y)$ is calculated by taking five iterations of $y_0 = \bot$, $y_{j+1} = Y(\hat{x}_k, y_j)$, and $\text{lfp}_y Y(\hat{x}_k, y) = y_5$. Once $\hat{x}$ is evaluated, the final value of $\hat{y}$ is evaluated by five more iterations of $\hat{y}_0 = \bot$, $\hat{y}_{j+1} = Y(\hat{x}, \hat{y}_j)$, and $\hat{y} = \hat{y}_5$.

We assume evaluating each output has unit cost, so we compute the cost of this computation as follows. There are five outputs in the Y function, so Y costs 5 to evaluate. It takes 5 iterations to evaluate $\text{lfp}_y Y(x,y)$, and this is done three times: twice to evaluate $\hat{x}$, and once more to evaluate $\hat{y}$. The X function is evaluated twice. Thus the total cost is $2^2 + (2+1)5^2 = 79$, unfortunately higher than the cost of evaluating the LFP using brute force.

We can do better. Evaluating $\text{lfp}_y Y(x,y)$ is the most costly part of this computation because we evaluated it using brute force. But it can be further decomposed and evaluated more cheaply. Fig. 3c shows the dependency graph for the Y function consists of three strongly-connected components (an SCC is a maximal set of nodes with directed paths between all pairs)—nodes 5 and 7, node 6, and nodes 1 and 3—whose least fixed point can be evaluated more efficiently using Corollary 1. Furthermore, it is more efficient to use Bekić's Theorem than brute force to evaluate the LFP of a two-output function.

To more succinctly represent these iterations, we introduce notation for representing a schedule, which indicates a sequence of nodes to evaluate. Juxtaposition is the fundamental part of the notation: writing *a b* means evaluate *a*, then evaluate *b*, each of which may be single nodes or more complicated schedules. A group of nodes surrounded by square brackets $[n_1\ n_2\ \cdots]$ is evaluated in parallel. Note that the order of nodes within brackets is irrelevant since they are all evaluated at once. The most complicated notation in our schedules describes an evaluation according to Bekić's Theorem and consists of two sub-schedules separated by a dot and surrounded by parenthesis with a superscript $(\ s_1\ .\ s_2\ )^n$, corresponding to n iterations of the sequence $s_2\ s_1$ followed by a single evaluation of s_2. So $(\ s_1\ .\ s_2\ )^1$ expands to $s_2\ s_1\ s_2$, $(\ s_1\ .\ s_2\ )^2$ expands to $s_2\ s_1\ s_2\ s_1\ s_2$, and so forth. In the language of Bekić's Theorem, s_1 evaluates X and s_2 is Y. In this notation, the brute-force, single decomposition, and multiple decomposition schedules for the example system are

$([5\ 7\ 6\ 3\ 1\ 4\ 2]\ .)^7$
$([4\ 2]\ .\ \ ([5\ 7\ 6\ 3\ 1]\ .)^5\ )^2$
$([4\ 2]\ .\ \ (5\ .\ 7)^1\ 6\ (3\ .\ 1)^1\ )^2$

This last schedule implies the following sequence of node evaluations:

7 5 7 6 1 3 1 [4 2] 7 5 7 6 1 3 1 [4 2] 7 5 7 6 1 3 1

which has cost 25 (each node evaluation has unit cost), substantially better than the brute-force cost of 49.

Our schedules are a generalization of those proposed by Bourdoncle [13], which always remove exactly one node from an SCC. This can lead to less efficient schedules for certain graphs, such as Fig. 3a. Furthermore Bourdoncle's scheduling algorithm is heuristic and can miss the optimal schedule, although it runs faster.

3.1 Merging Block Evaluations

These schedules describe evaluating nodes, yet in our language only blocks can be evaluated as a whole. The simple-minded approach of evaluating a whole block when the schedule calls for a single output on that block (i.e., a node) still produces the least fixed point because the blocks are monotonic. (It is easy to show that the sequence of intermediate results produced by evaluating nodes is a lower bound for the sequence produced by evaluating blocks.)

However, this approach is wasteful because it may perform more evaluations than necessary. To eliminate some (but not all) of this inefficiency, we propose the following algorithm that can reorder a schedule to take into account block evaluations.

First, consider the following rewrite rules. Written in a deductive style, they imply the subexpression above the bar can be rewritten as the subexpression below the bar when the condition on the right is true. Two helper functions simplify the conditions: $O(s)$ is the set of all indices that appear in subexpression s, and $I(i)$ is the set of predecessors of node i, i.e., all the nodes that directly affect output i.

$$\frac{s\ i}{i\ s} \quad \text{when } I(i) \cap O(s) = \emptyset \tag{4}$$

$$\frac{(s_1\ .\ s_2)^n i}{(s_1\ .\ s_2\ i)^n} \quad \text{always} \tag{5}$$

$$\frac{(s_1\ .\ s_2)^n i}{(s_1\ i\ .\ s_2)^n} \quad \text{when } I(i) \cap O(s_2) = \emptyset \tag{6}$$

$$\frac{(i\ s_1\ .\ s_2)^n}{(s_1\ .\ s_2\ i)^n} \quad \text{always} \tag{7}$$

$$\frac{i_1\ \cdots\ i_n}{[i_1\ \cdots\ i_n]} \quad \text{when } \forall j < k, O(i_j) \cap I(i_k) = \emptyset \tag{8}$$

$$\frac{[i_1\ \cdots\ i_n]}{i_1\ \cdots\ i_n} \quad \text{always} \tag{9}$$

The first rule, (4), follows from noting that i can be evaluated before s if i does not use any of the outputs from s. Note that s may depend on i, but evaluating i earlier only means the least fixed point may be reached sooner because the functions are monotonic.

Rule (5) actually increases the number of evaluations since

$$(s_1 \; . \; s_2)^n i = s_2 \; s_1 \; s_2 \; \cdots \; s_1 \; s_2 \; i \text{ and}$$
$$(s_1 \; . \; s_2 \; i)^n = s_2 \; i \; s_1 \; s_2 \; i \; \cdots \; s_1 \; s_2 \; i.$$

Rule (6) requires that i does not use any outputs of s_2. Examining the bottom sequence shows why this restriction is necessary: the final i is moved to just before the final evaluation of s_2:

$$(s_1 \; i \; . \; s_2)^n = s_2 \; s_1 \; i \; s_2 \; s_1 \; i \; \cdots \; s_1 \; i \; s_2.$$

Rule (7) just adds a trailing evaluation of i.

$$(i \; s_1 \; . \; s_2)^n = s_2 \; i \; s_1 \; s_2 \; i \; s_1 \; i \; \cdots \; i \; s_1 \; s_2$$
$$(s_1 \; . \; s_2 \; i)^n = s_2 \; i \; s_1 \; s_2 \; i \; s_1 \; i \; \cdots \; i \; s_1 \; s_2 \; i$$

Rule (8) says that any sequence of nodes that do not depend on partial results may be evaluated in parallel. And rule (9) says nodes evaluated in parallel can always be evaluated in sequence: the parallel result is always a lower bound for the series result because the functions are monotonic.

Together, these rules suggest an algorithm for rewriting schedules to better suit block evaluations. The goal of the algorithm is to move outputs on the same block together so they can be coalesced into a single parallel (block) evaluation by (8). It considers each node in turn, applying (4)–(7) repeatedly to find a position where the node can be merged with another on the same block using (8).

For the schedule

$$([4\;2] \; . \; (5 \; . \; 7)^1 \; 6 \; (1 \; . \; 3))^2$$

of the system in Fig. 3, applying (9) to node 4 then applying (7) gives

$$(2 \; . \; (5 \; . \; 7)^1 \; 6 \; (1 \; . \; 3) \; 4)^2.$$

Next, applying (5) and (8) gives

$$(2 \; . \; (5 \; . \; 7)^1 \; 6 \; (1 \; . \; [3\;4]))^2.$$

Replacing nodes with blocks produces the final schedule

$$(\text{A} \; . \; (\;\text{C} \; . \; \text{C}\;)^1 \; \text{C} \; (\;\text{A} \; . \; \text{B}\;)^1\;)^2,$$

which corresponds to the sequence of block evaluations

C C C C B A B A C C C C B A B A C C C C B A B.

1: **function** schedule(G,b) **returns** (s,c)
2: Decompose G into strongly-connected components $G_1,\ldots,G_n$
3: $b = \min\{b, \sum_{i=1}^{n} |G_i|^2 - |G_i| + 1\}$ *Bound to meet*
4: $(s,c) = (\text{the empty schedule}, 0)$ *Initialize the schedule and its cost*
5: **for all** strongly-connected components G_i in topological order **do**
6: $b' = b - c - \sum_{j=i+1}^{n} |G_j|$ *Rough bound for this component*
7: **if** $b' < |G_i|$ **then**
8: **return** $(\emptyset, \infty)$ *Scheduling is impossible in sub-linear cost*
9: **if** $|G_i| = 1$ **then**
10: Append G_i to s *Schedule for a single node is the node*
11: $c = c + 1$ *Schedule a single node*
12: **else**
13: $s' =$ the empty schedule *Best schedule for this component*
14: **for all** subsets X of G_i **do**
15: $(s'', c'') = \text{schedule}(G_i - X, \left\lfloor \frac{b' - |X|^2}{|X|+1} \right\rfloor)$ *Schedule rest of component*
16: **if** $|X|^2 + (|X|+1)c'' < b'$ **then**
17: $s' = ([G_i] \,.\, s'')^{|G_i|}$ *Evaluate this component using Bekić*
18: $b' = |X|^2 + (|X|+1)c''$ *Evaluation cost*
19: **if** s' is the empty schedule **then**
20: **return** $(\emptyset, \infty)$ *No schedule for this SCC met the bound*
21: Append s' to s
22: $c = c + b'$ *Add the cost of scheduling this component*
23: **if** $c \leq b$ **then**
24: **return** (s,c) *Met the bound*
25: **else**
26: **return** $(\emptyset, \infty)$ *Could not find a low-cost schedule*

Fig. 5. The branch-and-bound algorithm for finding the optimal schedule. Capital letters denote sets of nodes. Primed variables are associated with a strongly-connected component, double-primed variables with part of a component. Selecting which subsets to consider and in what order in line 14 is the crux of the algorithm. Choosing these judiciously is the subject of Section 3.3.

3.2 Scheduling

Fig. 5 shows the recursive branch-and-bound algorithm we use to compute schedules. It takes a dependency graph and a bound and attempts to find a schedule for computing the least fixed point of the system represented by the dependency graph that meets the bound. The core of the algorithm decomposes the graph into strongly-connected components (line 2) and then attempts to find a schedule for each SCC by further decomposing it (lines 5–22).

The algorithm always produces correct schedules. The schedule for a graph is a sequence of schedules for each SCC in topological order, since this is the order in which the algorithm considers them (line 5), and if a schedule for the SCC that meets the bound is found, it is added to the returned schedule in line 10 or 21.

Corollary 1 tells us that evaluating SCCs in topological order is correct. Each SCC is decomposed and scheduled using the computation in Bekić's Theorem, which appears in line 17.

The function begins by attempting to lower the given bound (line 3). It is always possible to evaluate an SCC by using Bekić's Theorem with a Y function containing exactly 1 node. For an n-node SCC, this costs $(n-1)^2+n = n^2-n+1$: the estimate used in line 3. This upper bound is tight because a fully-connected graph can require this many evaluations.

To schedule each SCC, the main loop (lines 5–22) begins by computing a bound for the SCC (line 6) by assuming all the remaining SCCs can be scheduled with linear cost. If this optimistic bound still requires the SCC to be scheduled in less than linear cost, then the function returns failure (line 8).

There are two ways an SCC can be scheduled. A trivial SCC consisting of a single node can always be scheduled with cost one, handled by lines 10–11. Handling nontrivial SCCs, done in lines 12–22, is the main challenge.

The branching in this branch-and-bound algorithm comes from different decompositions of each SCC. The most cost-effective decomposition is rarely obvious, so the algorithm tests many different ones, bounding the cost of the worst schedule it is willing to consider as better schedules are found.

Decomposing an SCC amounts to choosing the X function in Bekić's Theorem—a set of nodes that will become an outer loop in the schedule. The next section discusses how different subsets of each SCC are chosen in line 14; for the moment assume the algorithm considers all $2^n - 1$ possibilities.

The inner loop (lines 14–18) begins by calling the scheduler recursively to schedule the rest of the SCC (line 15). The bound deserves explanation. The cost of evaluating the entire SCC using Bekić's Theorem is $|X|^2 + (|X|+1)c''$ (the expression in lines 16 and 18), where $|X|$ is the number of nodes in the removed subset (the dimension of Bekić's X function) and c'' is the cost of evaluating the nodes that remain. The bound in line 15 must be met to make this expression less than b', the maximum cost allowed to evaluate the SCC.

If the cost c'' of evaluating the rest of the nodes in the SCC is sufficiently low, the schedule that produced it is used to schedule the SCC (line 17) using the computation in Bekić's Theorem, and its cost becomes the new bound.

If the algorithm finds a schedule for the SCC that meets the bound, the schedule for the SCC is appended to the schedule for the graph (line 21) and the cost of the component's schedule is added to the total schedule cost c (line 22). Note that since the SCCs are considered in topological order (line 5) and the schedule for each is appended to the schedule for the graph (line 10 and 21), the SCCs are scheduled in topological order in accordance with Corollary 1.

Finally, if the cost of all the SCCs did not exceed the bound (line 23), the function returns a pair consisting of the schedule for the graph and its cost (line 24), otherwise the function returns failure (line 26).

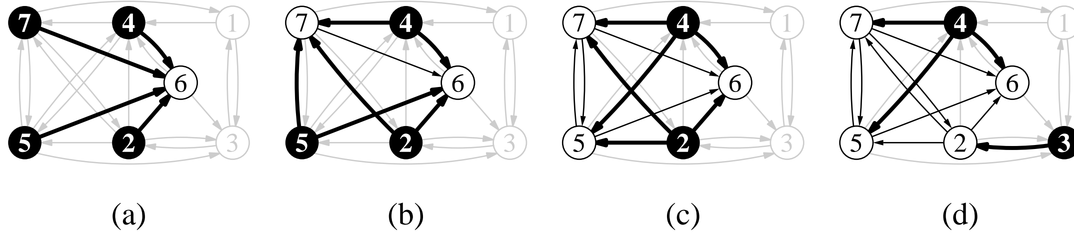

Fig. 6. The heuristic for partitioning an SCC in action. Nodes in K, the candidate kernel, are drawn normally. In each step, the heuristic returns the predecessors of K nodes, drawn in black, as a candidate subset X. (a) The initial $K = \{6\}$ gives $X = \{2,4,5,7\}$. (b) Adding node 7 to K gives $X = \{2,4,5\}$. (c) Node 5 added ($X = \{4,2\}$). (d) Node 2 added ($X = \{3,4\}$, $K = \{2,5,6,7\}$).

3.3 Finding Efficient Schedules

The branch-and-bound algorithm in Fig. 5 will always find the most efficient schedule if it considers all $2^n - 1$ subsets in line 14, but this makes the algorithm very costly (exponential) to run. The branch-and-bound technique does tend to reduce the number of possibilities considered by attempting to trim the search space as much and as soon as possible, but the asymptotic running time remains exponential.

If we are resigned to considering all possible subsets in line 14, which appears to be necessary to find the optimal schedule, we consider them in an order that attempts to find tighter bounds more quickly to reduce the number that must be considered. We consider all subsets of size 1 first, then all subsets of size 2, and so forth. This order should lower the bound more quickly because the cost of evaluating an SCC using Beckić's Theorem rises with the square of the size of the subset, as reflected in the cost computation in line 18.

If, however, we are willing settle for merely a good schedule instead of the optimum, we can greatly reduce scheduling time by using a heuristic that only considers some of the possible subsets. The motivation for this heuristic comes from the observation that an optimal partition of an SCC always breaks its strong connectivity, and that breaking strong connectivity requires making sure some subset of nodes in the resulting graph has no predecessors outside that set (Frank [25] calls this a well-known result). If partitioning an SCC did not break strong connectivity, the remainder would consist of a single SCC that would have to be evaluated using Bekić's Theorem. It would have been more efficient to have combined the two X functions rather than nesting the evaluation.

The heuristic tries to find a small subset of nodes to remove from the SCC to break strong connectivity. It does this by adding each node in the SCC one at a time to a set K. In each step, the heuristic returns a set X (used in line 14) which contains the predecessors of K. The branch-and-bound algorithm removes the X nodes from the graph, which usually makes K a kernel and thus breaks the strong connectivity of the graph. This does not break strong connectivity if $X \cup K$ is the whole SCC, e.g., in a fully-connected subgraph.

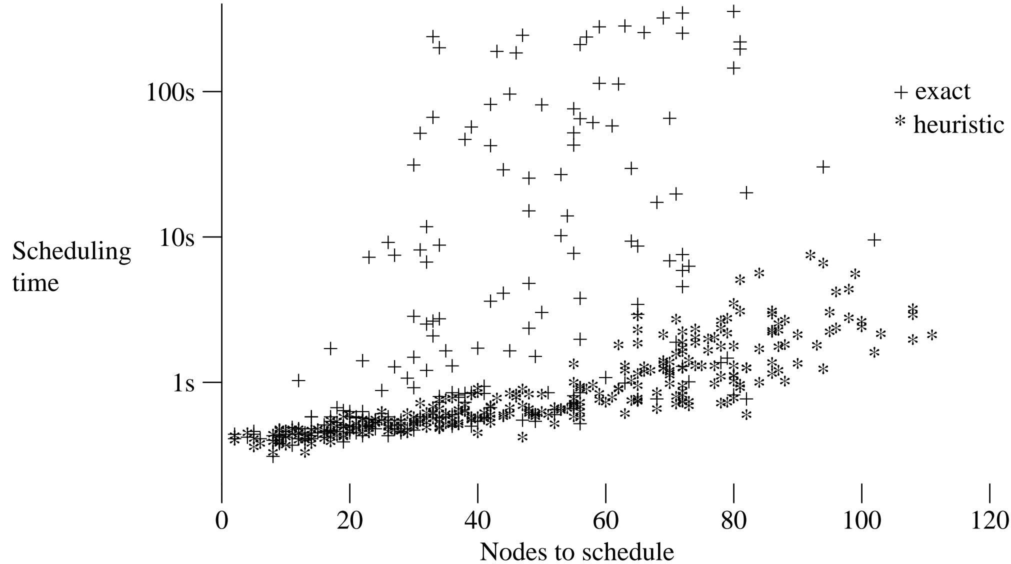

Fig. 7. Scheduling times for the exact and heuristic algorithms. Times are on a SPARCStation 10.

Fig. 6 illustrates how this heuristic chooses partitions (i.e., subsets of a strongly-connected component) for the dependency graph in Fig. 3b. Initially, K is seeded with a single node, 6 (Fig. 6a). Its predecessors are nodes 2, 4, 5, and 7, so the first subset $X = \{2,4,5,7\}$. Next, the algorithm adds one of the nodes in X to K, choosing the one that will produce the smallest X in the next step. Adding nodes 2 or 4 would add node 3 or 1 to X, but adding node 5 or 7 removes one node from X. The algorithm chooses 7 arbitrarily, returning $X = \{2,4,5\}$ (Fig. 6b). The next step adds 5 to K, again because adding nodes 2 or 4 would add a node to X and adding 5 reduces the size of X. This produces $X = \{2,4\}$, which turns out to be the best choice; however, the algorithm continues to produce new X sets until every node has been added to K.

This heuristic may not locate the optimal subset for two reasons. First, certain kernels are missed because only one node is ever added to the kernel set, even when there is a choice. Second, the optimal subset may be larger than the minimum—it may include nodes that are not required to break strong connectivity.

4 Experimental Results

To test the efficiency of the scheduling algorithm and the partition selection heuristics, we generated 304 block diagrams at random and found the minimum cost schedule for each using both the exact and heuristic scheduling algorithms. The exact algorithm considers all possible subsets by first considering all subsets with one vertex, then all with two, and so forth: an exponential number of possibilities. The heuristic variant uses the algorithm described in the last section to choose small subsets within the branch-and-bound technique.

To create the random block diagrams, we generated sixteen systems with two

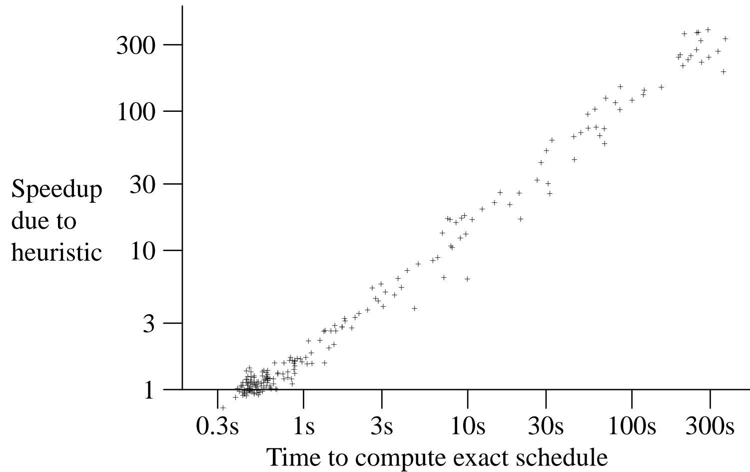

Fig. 8. Scheduling time speedup due to the use of the heuristic as a function of exact times.

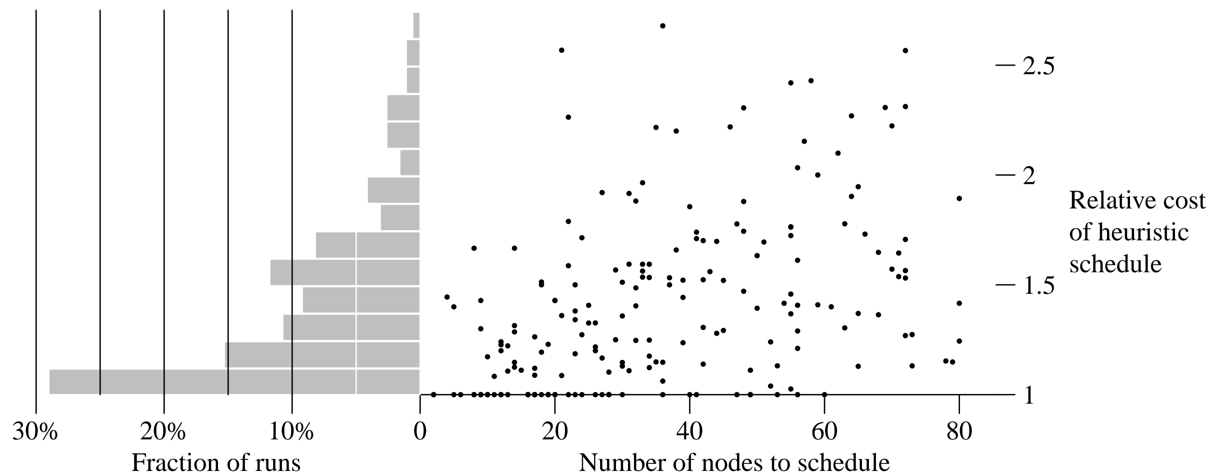

Fig. 9. The cost of running the heuristic as a function of the problem size.

blocks, sixteen with three blocks, etc., up to twenty blocks. For each block in a system, we independently selected a number of inputs and outputs at random from one to ten, uniformly distributed. Then, we connected each block's input to a block and output chosen at random.

All data were collected on a SPARCStation 10 with 96MB of main memory, although the program never consumed more than about 4MB. All times include the time to initialize the program and load the system, typically a few hundred milliseconds.

Fig. 7 shows the time it took to schedule each system using the exact and heuristic algorithm. The number of outputs in the system is plotted horizontally (the sum of the number of outputs on each block—exactly the number of vertices in the dependency graph). The times are plotted vertically on a logarithmic scale. The exact algorithm required over 500 seconds to compute a schedule for 98 systems (out of 304), but the heuristic always took less than eight seconds.

From Fig. 7, it appears the time to run the exact algorithm varies substantially and grows quickly with problem size. The heuristic algorithm's run time also appears

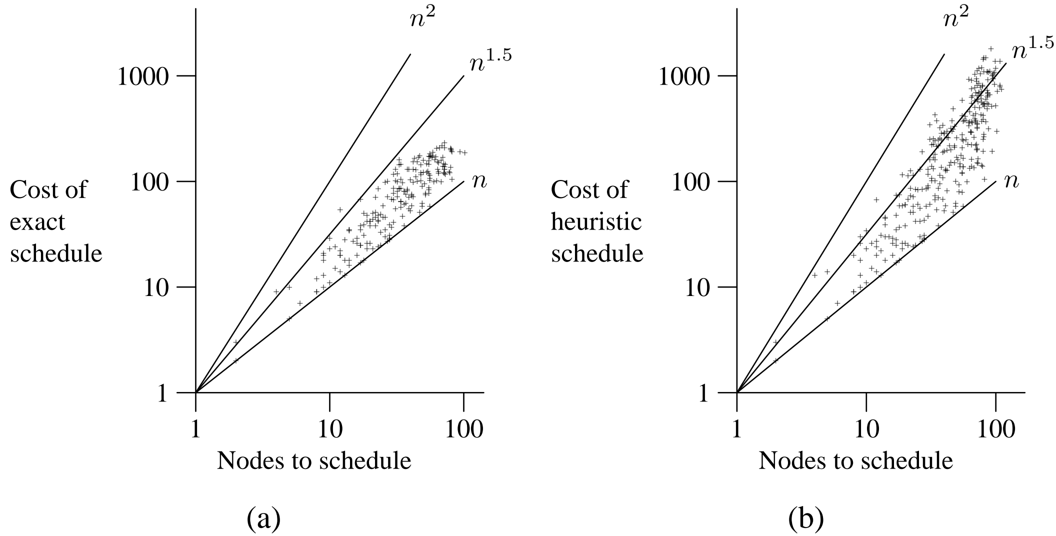

Fig. 10. The cost of the schedule generated by the exact (a) and heuristic (b) schedulers as a function of problem size.

to be growing exponentially, but much more slowly and predictably.

Fig. 8 shows the heuristic is exponentially more efficient than the exact algorithm. Although the speedup is between $1\times$ and $2\times$ about 40% of the time, and the heuristic is actually slower in about 20% of the cases, this is only the case when both the exact and heuristic times are fairly small. For longer times (e.g., one second or more), the heuristic partitioner is the clear winner by an exponentially growing margin.

To save time, the heuristic partitioner considers only a subset of all possible partitions. Unfortunately, it can miss the optimal partition, leading to the cost increases shown in Fig. 9. But this penalty is not devastating: the increase is less than 12% for more than an quarter of the cases. Interestingly, the additional cost does not appear to be related to the problem size, suggesting the heuristic will continue to work well for larger problems.

The cost of an optimal schedule for an n-node graph ranges from n to $n^2 - n + 1$. The graphs in Fig. 10 bear this out—the cost of all schedules falls between the n and n^2 lines. However, more interestingly, the asymptotic bound appears to be closer to $n^{1.5}$. Of course, this is a function of the systems we chose to schedule, and there are systems whose optimal schedule costs $n^2 - n + 1$, but there do not appear to be many of them.

From these results, we conclude both the exact and heuristic partitioning schemes have merit. In many cases, finding the exact answer is computationally feasible, but when it is not, the heuristic scheme is much faster and produces comparable results—half of the time within 25% of the optimal schedule, and rarely more than twice as bad.

5 Related Work

This work arose from a desire to marry the heterogeneous philosophy of the Ptolemy project [16] with the synchronous semantics of the Esterel language [11]. The Ptolemy system consists of domains that each implement a particular block-diagram language. The structure of these domains has followed Lee and Messerschmitt's Synchronous Dataflow [32,33] block diagram language, which can be executed by a very efficient scheduler that needs little information about the blocks apart from their connectivity. This approach of allowing blocks to be black boxes enables heterogeneous systems to be modeled hierarchically by nesting systems described in other domains within blocks. When we began the work presented in this paper (c. 1996) it was not clear that this same philosophy could be applied to the synchronous languages, whose execution then required detailed understanding not just of the components of the system but also of the system's state space.

Although Benveniste and Le Guernic's synchronous dataflow language SIGNAL [6] also has provisions for dealing with instantaneous feedback, its solution is very language-specific. Instead, the semantics presented here follow from Berry's more general constructive semantics for Esterel [8,10], which also address the instantaneous feedback problem. This grew out of his work on implementing Esterel programs in hardware [7,8], which could produce irksome cyclic circuits. Malik's procedure for analyzing the meaning of such circuits [34,35] provided a solution and connected the synchronous semantics of Esterel with the fixpoint semantics long used in the denotational semantics community pioneered by Dana Scott and Christopher Strachey in the early 1970s [39]. Textbooks such as Gunter [26] or Winskel [43] describe this philosophy in detail.

Shiple, Berry, and Touati [40] describe the procedure the Esterel V5 compiler uses to handle programs with instantaneous feedback: the program is first translated into a netlist using Berry's procedure [7,8]. We took the semantics of our block diagram language from the semantics of these netlists. Next, any strongly-connected components in the netlist are unrolled using Bourdoncle's algorithm [13] and the state space of the program explored symbolically [21] using Binary Decision Diagrams [15,14].

Our execution procedure amounts to using chaotic iteration to find the least fixed point. Chaotic iteration has been widely studied as a method for finding solutions to systems of equations [38]. One of its main advantages, which we do not exploit, is its ability to be used on parallel hardware without the need for synchronization [2]. A series of researchers have shown that chaotic iteration schemes converge under successively weaker conditions [41,42]. Wei notes that the computation will converge even if older intermediate results (say, those that might not have yet come from another processor running in parallel) are used. This result, stronger than we need, confirms our ability to evaluate blocks even though our analysis is done on a per-output basis.

Our scheduling technique builds on Bourdoncle's work [13], which comes from the field of abstract program interpretation pioneered by Cousot and Cousot [23,24,22]. Our schedules are a strict superset of Bourdoncle's because we are able to remove

more than one node at a time from strongly-connected components, which can be a great advantage for highly connected graphs. Furthermore, our algorithm, when run in exact mode, can guarantee an optimal (lowest cost) schedule, whereas Bourdoncle's algorithm is a heuristic.

Berry and Sentovich [12] present another technique for executing systems with constructive semantics (e.g., our block diagrams). Their goal, however, is execution within the asynchronous domain of the POLIS project's CFSMs [1,20] which has no explicit scheduler, something that could be found in a distributed system. Thus, while their semantics are very similar to ours, their execution environment is far less disciplined and hence less predictable. It does, however, have the novel ability to pipeline the execution of a synchronous system. Caspi, Girault, and Pilaud [17,18] propose a more disciplined mechanism for distributed implementations of synchronous systems, although theirs does not directly implement constructive semantics, instead assuming a system's behavior has been coded into a finite automaton.

The key difference between Esterel's constructive semantics [8,10,12] and ours is the admission of $\perp$ values on signal values. For a program to be considered correct, Esterel's constructive semantics specifically prohibits the appearance of $\perp$ on any signal, internal or otherwise, whereas our semantics permits this. As a result, our approach does not require systems to undergo the extensive analysis necessary to prove a program constructive to run it, unlike Esterel. While this does appear to permit seemingly erroneous systems under our scheme, it has the benefit of allowing heterogeneity, i.e., our systems can execute without the compiler/scheduler knowing details about the contents of the blocks.

Berry's group has an experimental system for performing separate compilation of Esterel modules. Each module is compiled into a separate netlist. Before the system runs, a linking phase wires these netlists together, which sometimes requires unrolling to address Esterel's reincarnation problem (an idiosyncrasy due to preemption constructs in Esterel, not shared by our block diagram language). Finally, the resulting netlist is simulated using three-valued logic by traversing the in-memory network. This technique does not allow compile-time scheduling, and is probably not very efficient. To our knowledge, this work has not been published.

The Lustre language [19,28] explicitly prohibits zero-delay feedback loops and the compiler requires detailed understanding of the program. The compilation technique [29] explores the state space of the system to build an automaton. A simple-minded search would produce many equivalent states, so the compiler employs a clever state minimization technique that removes these redundant states on the fly.

Benveniste et al. [5] propose another approach to separate compilation of synchronous specifications based on characterizing functional dependencies among inputs and outputs on a single block. Rather than completely characterizing the I/O behavior of a block, they abstract it either structurally ("this output depends on this input") or functionally ("this output depends on this input when the block is in state A"). They still require, however, that the system have an acyclic computation order in every cycle, something they admit may not be trivial to prove.

6 Conclusions

We have presented the semantics of a coordination language for synchronous software systems along with a practical implementation policy. It is novel in its ability to handle heterogeneous zero-delay software blocks with feedback and remain deterministic. The formal semantics for these systems is based on the least fixed point of a monotonic function on a finite partial order, and we presented an execution scheme that finds this least fixed point by executing the blocks in a fixed order according to a schedule.

The schedules are derived from a recursive strongly-connected component decomposition of the system. Any schedule so derived is correct, but the cost of executing a particular schedule depends on the choice of nodes to remove from each SCC. We use a branch-and-bound algorithm to make good choices, and have both an exact way to develop the set of choices and a heuristic that greatly reduces the number of choices to consider at the expense of sometimes returning a non-optimal schedule.

The language and scheduler have been implemented as the SR Domain, part of the "Ptolemy Classic" environment available from http://ptolemy.eecs.berkeley.edu/. There, blocks written in the other Ptolemy domains, including dataflow and discrete-event, can be imported into SR block diagrams to build heterogeneous simulations.

Almost certainly there are more sophisticated algorithms for choosing the nodes to remove from an SCC. It is an open question whether this scheduling problem is NP-hard, but we suspect it is due to the subtle relationship between a graph and the optimal schedule for it. However, since determining the minimum number of nodes required to break the strong connectivity of a graph can be done in polynomial time with network flow algorithms, there is still hope for a polynomial-time algorithm.

References

[1] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, K. Suzuki, Hardware-Software Co-Design of Embedded Systems: The POLIS Approach, Kluwer, Boston, Massachusetts, 1997.

[2] G. M. Baudet, Asynchronous iterative methods for multiprocessors, Journal of the Association for Computing Machinery 25 (2) (1978) 226–244.

[3] H. Bekić, Definable operations in general algebras, and the theory of automata and flowcharts, in: C. B. Jones (Ed.), Programming Languages and Their Definition, Vol. 177 of Lecture Notes in Computer Science, Springer-Verlag, 1984, pp. 30–55.

[4] A. Benveniste, G. Berry, The synchronous approach to reactive real-time systems, Proceedings of the IEEE 79 (9) (1991) 1270–1282.

[5] A. Benveniste, B. Caillaud, P. Le Guernic, Compositionality in dataflow synchronous languages: Specification & distributed code generation, Information and Computation 163 (2) (2000) 125–171.

[6] A. Benveniste, P. L. Guernic, Hybrid dynamical systems theory and the SIGNAL language, IEEE Transactions on Automatic Control 35 (5) (1990) 535–546.

[7] G. Berry, Esterel on hardware, Philosophical Transactions of the Royal Society of London. Series A 339 (1992) 87–103, issue 1652, Mechanized Reasoning and Hardware Design.

[8] G. Berry, The constructive semantics of pure Esterel, draft book, see http://www.esterel.org (1999).

[9] G. Berry, The Esterel v5 Language Primer, Centre de Mathématiques Appliquées, part of the Esterel compiler distribution from http://www.esterel.org (Jul. 2000).

[10] G. Berry, Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, 2000, Ch. The Foundations of Esterel.

[11] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152.

[12] G. Berry, E. Sentovich, An implementation of constructive synchronous programs in POLIS, Formal Methods in System Design 17 (2) (2000) 165–191.

[13] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Formal Methods in Programming and Their Applications: International Conference Proceedings, Vol. 735 of Lecture Notes in Computer Science, Springer-Verlag, Novosibirsk, Russia, 1993.

[14] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient implementation of a BDD package, in: Proceedings of the 27th Design Automation Conference, Orlando, Florida, 1990, pp. 40–45.

[15] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers C-35 (8) (1986) 677–691.

[16] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, Ptolemy: A mixed-paradigm simulation/prototyping platform in C++, in: Proceedings of the C++ At Work Conference, Santa Clara, CA, 1991.

[17] P. Caspi, A. Girault, D. Pilaud, Distributing reactive systems, in: Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94, ISCA, Las Vegas, 1994.

[18] P. Caspi, A. Girault, D. Pilaud, Automatic distribution of reactive systems for asynchronous networks of processors, IEEE Transactions on Software Engineering 25 (3) (1999) 416–427.

[19] P. Caspi, D. Pilaud, N. Halbwachs, J. A. Plaice, LUSTRE: A declarative language for programming synchronous systems, in: ACM Symposium on Principles of Programming Languages (POPL), Munich, 1987.

[20] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli, A formal specification model for hardware/software codesign, in: Proceeding of the International Workshop on Hardware-Software Codesign, 1993.

[21] O. Coudert, C. Berthet, J. C. Madre, Verification of synchronous sequential machines based on symbolic execution, in: J. Sifakis (Ed.), Proceedings of the Workshop on Automataic Verification Methods for Finite State Systems, Vol. 407 of Lecture Notes in Computer Science, Springer-Verlag, Grenoble, France, 1989, pp. 365–373.

[22] P. Cousot, Program Flow Analysis: Theory and Applications, Prentice Hall, Upper Saddle River, New Jersey, 1981, Ch. Semantics Foundations of Program Analysis, pp. 303–346.

[23] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: ACM Symposium on Principles of Programming Languages (POPL), Association for Computing Machinery, Los Angeles, California, 1977, pp. 238–252.

[24] P. Cousot, R. Cousot, Automatic synthesis of optimal invariant assertions: Mathematical foundations, in: Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages, ACM SIGPLAN Notices 12(8), Rochester, New York, 1977, pp. 1–12.

[25] A. Frank, How to make a digraph strongly connected, Combinatorica 1 (2) (1981) 145–153.

[26] C. A. Gunter, Semantics of Programming Languages, MIT Press, Cambridge, Massachusetts, 1992.

[27] N. Halbwachs, Synchronous Programming of Reactive Systems, Kluwer, Boston, Massachusetts, 1993.

[28] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language LUSTRE, Proceedings of the IEEE 79 (9) (1991) 1305–1320.

[29] N. Halbwachs, P. Raymond, C. Ratel, Generating efficient code from data-flow programs, in: Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming (PLILP), Vol. 528 of Lecture Notes in Computer Science, Passau, Germany, 1991.

[30] J.-L. Lassez, V. L. Nguyen, E. A. Sonnenberg, Fixed point theorems and semantics: A folk tale, Information Processing Letters 14 (3) (1982) 112–116.

[31] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, Programming real-time applications with SIGNAL, Proceedings of the IEEE 79 (9) (1991) 1321–1336.

[32] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Transactions on Computers C-36 (1) (1987) 24–35.

[33] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.

[34] S. Malik, Analysis of cyclic combinational circuits, in: Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD), Santa Clara, California, 1993, pp. 618–625.

[35] S. Malik, Analysis of cyclic combinational circuits, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13 (7) (1994) 950–956.

[36] F. Maraninchi, The Argos language: Graphical representation of automata and description of reactive systems, in: Proceedings of the IEEE Workshop on Visual Languages, Kobe, Japan, 1991.

[37] F. Maraninchi, Operational and compositional semantics of synchronous automaton compositions, in: CONCUR '92. Third International Conference on Concurrency Theory., Vol. 630 of Lecture Notes in Computer Science, Springer-Verlag, Stony Brook, NY, 1992, pp. 550–564.

[38] F. Robert, Discrete Iterations: A Metric Study, Vol. 6 of Springer Series in Computational Mathematics, Springer-Verlag, 1986.

[39] D. Scott, C. Strachey, Toward a mathematical semantics for computer languages, in: Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971, pp. 19–46.

[40] T. R. Shiple, G. Berry, H. Touati, Constructive analysis of cyclic circuits, in: Proceedings of the European Design and Test Conference, Paris, France, 1996, pp. 328–333.

[41] A. Üresin, M. Dubois, Parallel asynchronous algorithms for discrete data, Journal of the Association for Computing Machinery 37 (3) (1990) 588–606.

[42] J. Wei, Parallel asynchronous iterations of least fixed points, Parallel Computing 19 (8) (1993) 886–895.

[43] G. Winskel, The Formal Semantics of Programming Languages: An Introduction, Foundations of Computing, MIT Press, Cambridge, Massachusetts, 1993.