Alternate Semantics of the Guarded Conditional

Nachum Dershowitz

School of Computer Science, Tel Aviv University, Ramat Aviv, Israel

Abstract

The various possible semantics of Dijkstra's guarded commands are elucidated by means of a new graph-based low-level, generic specification language designed for asynchronous nondeterministic algorithms. The graphs incorporate vertices for actions and edges indicating control flow alongside edges for data flow. This formalism allows one to specify, not just input-output behavior nor only what actions take place, but also precisely how often and in what order they transpire.

1 Introduction

The behavior of asynchronous algorithms can be specified using various formalisms, some more precise than others. There may be events that must occur in a particular order; others may be such that their relative order is immaterial; there may be sterile events along execution paths that turn out to be unproductive. We propose a generic graph-based low-level specification language for such algorithms and a graph-based operational semantics for describing their computations. This allows one to infer, not just input-output behavior, nor only what actions take place, but also how often and in what order they occur.

We require the ability, for example, to formally express the intention that if $y = 0$ and the nondeterministic guarded conditional **if** $y \geq 0 \rightarrow z := 1 \; [\!] \; y \leq 0 \rightarrow z := -1$ **fi** is executed, then one or both of the guards are evaluated, in either order or simultaneously, whereas exactly one of the assignments is executed. We want to contrast the computation of the parallel assignments $y := 1 \parallel z := -1$ for which the order of execution matters not, and $z := 1 \parallel z := -1$ for which the outcome depends on the order. Our interest is in operational semantics, so there is a difference between **if** $odd(x)$ **then** $x := 0$ **else** $x := x - x$ **fi** and **if** $odd(x)$ **then** $x := x - x$ **else** $x := 0$ **fi** despite the fact that the outcomes are identical.

Our goal is a more precise, more flexible, and more general generic language for describing asynchronous computations and algorithms than currently available. More precise, on account of providing exact control over the order in which values are accessed and over the number of times they each are. Flexible, in that it allows arbitrary sequences of assignments and other operations as part of a single step. More general, because it is designed to incorporate a range of modes of choice and nondeterminism.

We use directed graphs to describe computations by setting down the actions performed and the *quasi*-order in which they transpire. We work with the most basic, atomic actions, namely, reads, writes, tests, and choices—some labeled with uninterpreted operation symbols. Actions that occur "simultaneously" sit in the same equivalence class of this quasi-ordering. The consecutive states of distributed processes might not be well defined, as the precise order of actions on the state may be under-determined. Hence, the ordering may be partial. But the graph records the effects of actions, and it preserves the order in which they occur to the extent that subsequent behavior might depend on the order.

An algorithm will be presented as a finite directed graph whose vertices are actions on the state. The initial state determines the memory locations accessed, their values, the outcomes of tests, and any state changes. In addition to the usual edges describing control flow, there will be data edges showing whence values are produced and where they may be used by subsequent operations. Examples of such algorithms and some partial computations are given in the appendix.

2 Motivation

Consider the simplest case of Dijkstra's guarded commands [20], **if** $p \rightarrow s \;[\!]\; q \rightarrow t$ **fi**. The intention is as follows: (*a*) If only condition p holds true, then p must be evaluated, command s is then executed, and the whole statement succeeds. It may be that the other guard q is also evaluated before, after, or at the same time (and found false); however, t is definitely not to be executed. (*b*) Likewise, if only q holds true, then q is evaluated, t is executed, and the statement succeeds. Even if p is also evaluated, s is not executed. (*c*) If both p and q are true, then at least one—but maybe both—of p and q are evaluated but only one of s and t is executed, and the whole statement succeeds. (*d*) If neither p nor q is true, then both p and q must be evaluated to determine that that is the case, but neither s nor t is executed, and the whole statement fails (aborts). Explanations, such as the following, in which *all* guards are *always* evaluated, are misleading: "Upon execution of a selection all guards are evaluated. If none of the guards evaluates to true then execution of the selection aborts, otherwise one of the guards that has the value true is chosen non-deterministically and the corresponding statement is executed" [29]. A similar understanding is formalized in [13]. It is more than plausible, however, that an implementation would actually evaluate only one guard if it turns out that the first guard considered evaluates to true.

Now consider the possibility (not entertained by Dijkstra) that the evaluation of a guard (p and/or q) can fail or might not terminate. Different possible behaviors—now with potentially different outcomes—may be understood; the question is which is intended, and how to make that precise. Suppose a branch enters into a nonhalting computation, as in **if** TRUE $\rightarrow$ **loop** $[\!]$ TRUE $\rightarrow$ **abort** $[\!]$ TRUE $\rightarrow$ **skip fi**. We might want for the other branches eventually to be attempted, as though all options are explored in parallel. This gives rise to three options: (1) Any one branch is taken, with possible outcomes: looping, failing, succeeding—as in the guarded command language [20]. (2) In case of explicit failure, an alternate possible branch is chosen, leaving two possibilities: looping or succeeding, as in Prolog [12] or Icon [22]. (3) All branches are attempted, so success is the only possible eventuality, as in Dynamic Logic [28]. In short, the need to make the intended semantics of programming instructions precise and explicit is inescapable. As we will see, incorporating constructs that signal success or failure explicitly, for situations such as angelic choice, can help resolve such ambiguities. See [28] for an analysis of Dijkstra's intent and of some of the alternatives.

3 Control Scenarios

In an asynchronous system, some details regarding the order of actions during execution may be deemed irrelevant and should not be nailed down more than necessary by the semantics. Other aspects of the order of actions may affect the outcome and need to made precise. Algorithms specify the behavior of computations under different initial circumstances. Before we look at algorithms, however, we need to understand the behavior of individual asynchronous computations.

A computation may be thought of as traversing a finite or infinite directed graph—the *control graph*, expressing the control order in which *events* transpire. So, its vertices are events, each with a label drawn from some set of possible actions. A *scenario* comprises a (finite or infinite) set V of event vertices, labeled by event types L, and a multiset E of control edges over V. This defines a *control* (multi-)graph

$\langle V, E, \ell \rangle$, with labeling $\ell : V \to L$. We'll draw a red control edge, $\alpha \longrightarrow \beta$ from event $\alpha \in V$ to event $\beta \in V$ if β is an *immediate* successor of α. This means that β requires α, so it cannot occur before α. Thus, computation proceeds in steps, each of which involves one or more events.

A scenario proceeds along the (red) control edges of the control graph. Events on a cycle of control edges are *simultaneous*. A *stage* in a scenario is a directed cut, a "dicut", of the scenario's control graph. Each cut divides the graph into an *past segment* of the scenario—what has been done, and a *future segment*—what still needs doing. (This is akin to the cuts in a distributed system that give consistent global states; see [34], for instance.) A directed cut cannot cut a cycle, so simultaneous events must all be on the same side.

Each step takes the scenario from one stage to the *next*, that is to another stage whose additional events are wholly in the future of the first, current stage and between which there is no other stage. Each stage may be followed by any number of next stages. A *move* is a finite sequence of (one or more) transitions from one stage to a possible next stage, such the transitions in an iteration of a loop. In this way a (partial or complete) scenario may be decomposed into a sequence of moves.

A *component* in a scenario is a (nonempty) maximal strongly connected component of simultaneous events in the control graph. The events in a component must all be part of the same step, or else the order would be violated. Independent events, on the other hand, may belong to the same step or to different steps. Thus, a single step will necessarily consist of one or more pairwise incomparable components. There are no two strictly ordered events in a single step.

A scenario may have finitely or infinitely many stages. Regardless, we require confluence, insisting that "all roads lead to Rome": For any two stages in a scenario, there is always at least one subsequent stage reachable from both, as will be shown in Lemma 3.1 below. This is necessary for our notion of a scenario to be well defined. When there are more than one next step, it should matter not which one is pursued. Hence, the specific choices taken to reach any particular stage will not affect the ultimate result. In other words, a single scenario may embody many different sequences of steps, but the outcome—to the extent there is any outcome—of the scenario is independent of the choice of sequence.

The transitive closure of the relation $\to$ between events in a scenario is the *control* quasi-ordering $\precsim$, corresponding to reachability in the graph. For control (quasi-) ordering $\precsim$, event α *(strictly) precedes* event β and β *(strictly) succeeds* α, written $\alpha \prec \beta$, if $\alpha \precsim \beta$ but $\beta \not\precsim \alpha$. Events are equivalent, $\alpha \simeq \beta$, when both $\alpha \precsim \beta$ and $\beta \precsim \alpha$, and incomparable, $\alpha \perp \beta$, when neither holds.

Events that are equivalent in this ordering occur *simultaneously*—understood literally or figuratively. If they are incomparable in the ordering, we'll refer to them as *independent*. Events are said to be *sequential* or *ordered* if one strictly precedes the other in this quasi-ordering. Two events are *nonsequential* if they are either simultaneous or independent. The execution order of independent events is unknown or unknowable, but in any case must be irrelevant to the outcome.

Formally, a *dicut* (directed cut) A/B is a cut of a graph $G = (V, \to)$, that is, a bisection of its vertices ($V = A \uplus B$), in which all edges $\to$ of G go in the same direction, from *preceding* events A to *succeeding* events B: $\forall a \in A, b \in B.\ b \not\to a$.

Consider a control graph $(V, \to)$. A cut A/B, where $A, B \subseteq V$, can be represented by A alone since $B = V \setminus A$. Such an A must be (downward) closed under $\precsim$ (predecessor). The control relation $\to$ induces a *stage relation* on cuts: $A \rightsquigarrow A'$ if $A \subsetneq A'$ and $\forall a' \in A'. \forall v \in V \setminus A.\ (v \not\to a' \vee v \simeq a')$. This is the "next stage" relation. It respects the direction of control edges: from events in the previous stage to all new events in the next stage. Simultaneous events stay together; independent events need not. From one stage to the next, a set of nonsequential events transpire.

Strong confluence (see [16]) follows from the definitions:

Lemma 3.1. *If $A \rightsquigarrow B$ and $A \rightsquigarrow C$, for stages A, B, C of some computation, then $B \cup C$ is also a stage of the scenario and $A \rightsquigarrow B \cup C$, $B \overset{=}{\rightsquigarrow} B \cup C$, and $C \overset{=}{\rightsquigarrow} B \cup C$, where $\overset{=}{\rightsquigarrow}$ is the reflexive closure of $\rightsquigarrow$.*

This lemma is crucial. It means that the graph of a scenario has all the information needed to determine its outcome, regardless of the precise sequence of steps that may have transpired behind the scenes—provided that the order of independent actions does not matter. In the final analysis, the same actions will have taken place, though not necessarily in the same sequence. See Theorem 6.1 below.

Stages are naturally partially ordered by the reflexive-transitive closure $\leq$ of the stage relation $\rightsquigarrow$. If $A \leq B$ for stages A, B, then stage B is *reachable* from stage A in the sense that there is a sequence $A = A_0 \rightsquigarrow A_1 \rightsquigarrow \cdots \rightsquigarrow A_n = B$, $n \geq 0$, of stages such that each A_i has A_{i+1} as one of its possible next stages. A sequence of consecutive stages like $A_0 \rightsquigarrow A_1 \rightsquigarrow \cdots \rightsquigarrow A_n$ is a *run*.

Given a graph quasi-ordering its events, and identifying a stage with its preceding events, the stage order may be defined as follows:

Definition 3.2 (Stage order)**.** Given a control order $\lesssim$ on events V, two stages, $A, A' \subseteq V$, are ordered $A \leq A'$ in the *stage order* if $A \subseteq A'$ and $\forall a' \in A'.\, \exists a \in A.\, a \lesssim a'$.

(This is a special case of the Egli-Milner powerset construction; see [1, Def. 6.2.2].)

A full scenario begins with an *initial* stage. The empty cut $\varnothing / V$ is the unique lower bound of the stage relation $\leq$, which is the initial stage. The full cut $V / \varnothing$ is its unique upper bound. When finite, a scenario has a single outcome, the unique *final* stage, with no stages following it. However, in the case of an infinite graph that upper bound $V / \varnothing$ is unreachable.

The central precept of the formalism espoused here is that an event only proceeds after it receives control from all its immediate predecessors, much like (simple) Petri nets [36]. We have no explicit notion of process or processor. The number of events that take place simultaneously may vary arbitrarily during a computation. Control flow is an abstraction for both allocation of processes as well as data dependencies.

4 State Semantics

A *computation* captures the salient aspects of the behavior of a single execution of a computational process. The behavior of a computation involves state changes. Thus, a *computation* is a scenario wherein events are actions that act upon states. We wish to analyze concrete computations in detail, be they algorithmic or not. By "algorithmic" we mean that they are finitely describable. Later on, we will explore how one might specify algorithmic computations by means of graphical programs.

The stages of a computation are associated with *(global) states*. At each stage, there is a specific *current* state. The initial state is the current state of the initial stage; likewise, the final state of a finite computation is the current state of its final stage. In this way, a computation transforms the given initial state step by step.

But what do states look like? What exactly transpires during computation? And how can we faithfully describe that? The state of a standard Turing machine is composed of three parts: (a) the contents of the tape; (b) the positions of the reading heads on the tape; and (c) the internal state (Turing's "state of mind"). Similarly, the *state* of the computations that we are envisioning consists of three components: (a) a memory *store*, containing both the definitions of all builtin operations as well as the current *values* of all variables and additional mutable data; (b) a *service panel*, which holds values that have been retrieved from the store and may be needed further down the line—like values held temporarily in processor registers; and (c) a *control panel*, indicating where in the process control currently lies—like the internal state of a Turing machine, or the program counter of von Neumann machines, or the multiple program counters of multithreaded parallel execution. The service and control panels are referred to together as the process' "panel".

The store comprises a set of *locations* in which values are stored, values that may be changed along the way. *Actions* are designed to retrieve a stored value or to update it. Locations that are not actively modified by a computation should hold invariant. States are global, as we impose no restrictions on which values held in the state may be modified by which actions. Only the control panel determines the next upcoming actions. The values in the service panel determine the outcomes of those actions. Those outcomes produce values that show up in the subsequent panel.

Let $\mathcal{S}$ be the set (or class) of states of some process and $\mathcal{A}$, the set of allowed (atomic) actions on stores. Generically, an action requires some incoming control lines to be live for it to execute. It may access various locations in the store, use various values available in the service panel, modify the values of various locations in the store, produce values for the subsequent service panel, and transfer control to a subset of the outgoing control lines.

Events in a control graph are instances of actions, which act upon states during computation and come in different flavors. Actions $\tau \in \mathcal{A}$ are state-transition functions $\tau : \mathcal{S} \to \mathcal{S}$. (In a more general framework than the one dealt with in this paper, events may be partial or multivalued functions.) Some actions change the store. Other actions might have no impact on the store, but instead may provide data needed for decisions or affect only the panel to be used by subsequent events.

The graph formalism applies equally well regardless of the internal complexity of actions. But, as we are interested in foundational issues, we turn to basic, atomic actions. As Alan Turing observed in the pencil and paper domain [41], there are three kinds of basic actions of importance in computations: reads, writes, and choices. *Read*: Query and retrieve a value from the state, or—put another way—apply some operation known to the state to some arguments. *Write*: Request and store a value within the state, or—viewed alternatively—update the way future queries are to be answered. *Decide**: Choose what to do next on the basis of a retrieved value. Read and write actions act on states. Reads access values known to the state, but leave the store as is. In the Turing-machine framework, only a bounded number of cells are available for inspection in any one step, and each cell contains one of a given *finite* set of symbols. In more general algorithmic frameworks, stored values may come from an infinite set, such as the natural numbers. Accessed values may become part of the panel part of the state, as we will see. Writes modify values; so, they change the store. Reads also serve the purpose of guiding decisions as to how to proceed; the next action depends on what was read.

Considerations of what constitute universal atomic actions underlying generic algorithmic computation led to Gurevich's design of *abstract state machines (ASMs)*. These serve as a most generic model of sequential computation, able to precisely describe arbitrary classical (i.e. deterministic, non-parallel, non-distributed, non-interactive) algorithms, whether effective or not [24, 15]. The reading and writing of a tape of symbols in which a Turing machine engages are generalized to accessing and storing values $f(a_1, \ldots, a_n)$ for a set Σ of operations f. We build upon this insight.

The salient aspects of stores, as in the ASM framework [24], are captured by (first-order) logical structures. Let Σ be the vocabulary of stores—including all the symbols in reads. writes, and choices—and let $\mathcal{U}$ be the domain (universe) of values contained in locations of the store and panel. The stores of states interpret each of the symbols f in Σ as an operation $[\![f]\!]$ over $\mathcal{U}$. These operations may have fixed or variable arity. The value $[\![f]\!](\bar{a}) = [\![f(\bar{a})]\!] \in \mathcal{U}$ of $f \in \Sigma$ at $\bar{a} \in \mathcal{U}^*$ (U^* are the sequences of elements taken from $\mathcal{U}$), as interpreted by a state $\sigma \in \mathcal{S}$, resides at *location* $f(\bar{a})$. Let the set of all locations be $\mathcal{L} = \{f(\bar{a}) \mid f \in \Sigma, \bar{a} \in \mathcal{U}^*\}$, where the length of the sequence of elements $\bar{a}$ of $\mathcal{U}$ matches the arity of the operation symbol $f \in \Sigma$, if its arity is fixed. Nullary operations are scalars with arity 0, so we traditionally write c rather than $c()$. When an operation f, such as addition, is of variable arity, we may treat it as a family of operations, $f_0, f_1, f_2, \ldots$, one per possible arity. With the above in mind, stores may be viewed as specific assignments of values to locations: $\mathcal{S} = U^{\mathcal{L}}$. We also include data channels in the set $\mathcal{L}$ of locations.

Reads are labeled by a function symbol $f \in \Sigma$, take values $\bar{a}$ for the coördinates of the location being explored, and produce a value $b = [\![f(\bar{a})]\!]$. We can represent the read vertex fully by indicating the action and its outcome as follows: $f(\bar{a}) \vdash b$. Writes are also labeled by a function symbol $g \in \Sigma$, take values

$\bar{a}$ for the coördinates of the location being explored, plus a value $b \in \mathcal{U}$ to be assigned, and change the value of f at $\bar{a}$ in the current state to be b, whenceforth $[\![f(\bar{a})]\!] = b$. Thus, writes can be to arbitrary locations, not just to scalars. We represent the write vertex by $f(\bar{a}) \mapsto b$. Choices are labeled by a scalar symbol c and take one value, which is compared to the value of c.

To summarize, the set of all possible actions on stores, reads and writes, respectively, are

$$\mathcal{A} = \left\{ f(\bar{a}) \vdash b \mid f \in \Sigma,\ \bar{a} \in \mathcal{U}^*,\ b \in \mathcal{U} \right\} \cup \left\{ f(\bar{a}) \mapsto b \mid f \in \Sigma,\ \bar{a} \in \mathcal{U}^*,\ b \in \mathcal{U} \right\}$$

We assume a set $D \subseteq U$ of *designated* domain elements, such as a truth value T (standing for "true"), and dedicated constant symbols in the vocabulary for each. We will also insist that the values assigned to those dedicated symbols are immutable (never changed by an assignment). These are needed for conditionals.

It is advantageous to add the possibility of a *junction* action $\bullet$ that is just a "no-op"; it does nothing more than pass control to all outgoing edges after receiving control from all incoming edges. It serves as a "handshake": before proceeding, all the incoming processes must complete. In addition, we need to consider that algorithms make nondeterministic choices, which leave their mark on computations. Accordingly, we incorporate circles $\circ$ in computation graphs to indicate the points at which such choices occurred. As it turns out, it is crucial to also have choices ◆ that depend on whether a control line-is active (has already received power from its predecessors) or inactive (has not). We will call this a *probe*; see Fig. 3(a), with its thick green test line.

5 Flow of Values

The arrows of control graphs only indicate the order in which actions are performed. But each of the values used by those reads and writes must come from somewhere—not pulled magically out of a hat. We can just say that any value that is consumed by a read or a write must of necessity have been produced earlier—in the control order—by a read. See Section 6.

At this point, we have not yet insisted that the connection be explicit between the read that produces a value and the action that uses it. The complete description of a process should also indicate where each consumed value was produced, since any particular value could have been read from many different locations. For this reason purpose, we will be employing "channels". These may be named or numbered for convenience, in which case an algorithm can only use a finite vocabulary to refer to them.

To cope with the dataflow aspect of computation, we need to add another layer of edges to the graph, drawn from the producer of a value to each of its uses. If event α produces a value and a subsequent event β uses it, then we draw a (dashed) blue edge $\alpha \rightarrow \beta$. These edges form a dag, as they must be acyclic. We will refer to these data-edges as *channels*; they represent conduits for the transmission of produced values for subsequent use. They may correspond to messages passed from one process to another, or to some other mechanism of data transfer. So, they serve as a form of short-term memory. The number of channels coming in to a read action matches the operation's arity—if it has one—and determines the location that is looked up in the operation's table of values (its graph). A write action has one more channel, for the newly assigned value.

Channels are necessary for indicating how values move about during a computation. They cannot be supplanted by writes and reads. Whereas programming languages involve terms, graphical programs connect the function symbols in a term with each other to indicate how subterms contribute values to larger terms.

We will draw round read vertices, square writes, and rhombic choices, each labeled with the relevant operation. For example, here is a read of the value of the (unary) operation f (which requires one argument), a write to some (binary) operation g (which requires two arguments plus a new value for the

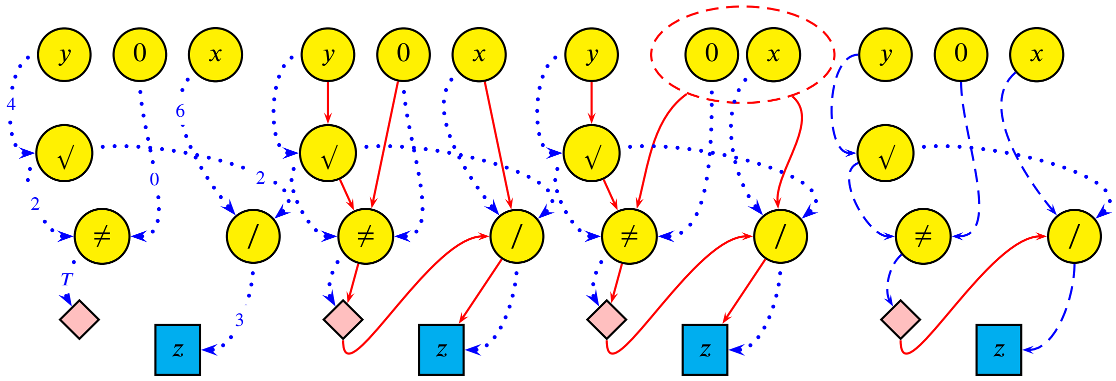
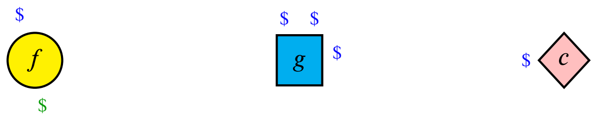


Figure 1: From left to right: (a) Graph with (dotted blue) channels and their values; (b) behavior graph with both (solid red) control and (dotted blue) data edges; (c) variant with simultaneous reads encircled; (d) simplified version, with dual-purpose dashed blue edges.

location), and a choice point that depends on an incoming value being (that of nullary operation) c:



Here, the (blue) dollar signs signify incoming operands and the green one, an outgoing value.

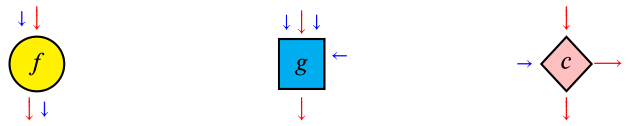
Reads $f(\bar{a})$ with result b have incoming channels for each of the components of $\bar{a}$ and optional outgoing channels carrying b. Scalar (zeroary) reads have no inputs. Writes $f(\bar{a}) := b$ have incoming channels for the $\bar{a}$ and also for b, but no outgoing channels. Incoming channels to reads and writes are fixed in number and ordered. Outgoing channels can be manifold and are not ordered (as we are dealing, for now, with the single-output case); the same output value goes down all the output channels.

In Fig. 1(a), one can see (dotted blue) channels showing the flow of data from vertex to vertex. The result of the square-root is produced once but used twice, once by its direct successor and again by its grandchild, the division. The outcome of the test is transmitted to the choice node but goes no further. (When the test is for truth, T, we omit the designated constant.) The division awaits the choice. The channel label is the value at its two ends, which must agree for the edge to be sensible (see Section 6). Since the order of operands matters (as in division here), channel edges must connect to specific *ports* of vertices. We should think of each vertex $v \in V$ as being a finite set of indexed ports v_i to which channels are connected. Accordingly, an edge e in the data graph can be represented as (u_j, ℓ, v_i), where u_j is a port of vertex u, v_i is a port of v, and $\ell \in \mathcal{V}$ is the value of the channel. Both u and v are also vertices of the control graph. (See, for example, [2] for graphs with ports.) The second subfigure has control edges included, but omits channel values.

Significantly, the value sent along a channel might no longer exist in stored memory by time it is used. On the other hand, one cannot consume a value before it is produced. So there must be a (red) control path (consisting of one or more edges) wherever there is a (blue) channel. Therefore, it will be frugal to combine data and control and omit the control edge when both are present. Henceforth, we use dashed blue for double-duty control-cum-data edges and solid red for control-only edges. Dotted blue will remain data sans control. So instead of the full diagram of Fig. 1(b), we can use the simpler one on the right. The middle graph shows how we handle simultaneous operations.

The *behavior graph* for a computation is the combination of control edges from the control graph and the labeled data edges of the dataflow graph. It describes what happens and why: on which prior actions does a particular action depend; whence does an action obtain the values it requires. The vertices are those of the control graph, which is a superset of those in the data graph, but with ports as in the data graph. In other words, $G = (V_c', E_c \cup E_d)$, where G is the behavior graph, (V_c, E_c) is the control multigraph, (V_d, E_d) is the labeled multigraph of the data, $V_d \subseteq V_c$ (ignoring ports), and V_c' is V_c enhanced with the ports of V_d. We point out that the cuts that defined stages (in Section 3) are also directed cuts of these enhanced graphs, since data edges always respect the control order.

A metaphor: The red, control arrows are like electric wiring; they are needed to power the actions. We will sometimes call them wires and use on/off or live/dead to refer to the two possible states of wire. The blue, data arrows are like plumbing; they supply the necessary resources. We call them "channels". Showing both types of arrows (long red controls; short blue channels), the central three action vertices look like this:



An action can be taken only after control reaches the incoming red arrow. This can be long after the input values are ready on the channels.

6 Constraints on Computations

The following assumptions are needed for computation graphs to be sensible:

A. Ordered reads of the same location produce the same value, unless a write intervenes.
B. After a write, all reads of that location result in the value that was written, unless another write intervenes.
C. The value of a location does not change unless there is a write to that location.
D. The values of designated symbols are constant.
E. Nonsequential reads of the same location retrieve the same value.
F. Nonsequential writes to the same location store the same value.
G. The value retrieved by a read is the same as the value stored by a nonsequential write.
H. A channel has matching values at its two ends.
I. Every input value of a read or a write must be an output value of some preceding read.
J. All outgoing channels of an action carry the same value.
K. The two ends of each channel are ordered by control.
L. All cuts cross a bounded number of control lines and channels.
M. There are only finitely many function symbols appearing in an algorithm.
N. Actions have bounded indegree.

We are insisting that if a read and write for the same location are nonsequential, the value retrieved by the read is the same as the value stored by the write. This demand is somewhat weaker than what is usually assumed, namely that reads and writes to the same location are always sequential, since only contradictory simultaneous behavior is precluded by the postulate: "Writing a word into or nondestructively reading a word from this store are undividable operations; i.e., when two or more computers try to communicate (either for reading or for writing) simultaneously with the same common location, these communications will take place one after the other, but in an unknown order" [19].

Given an initial state and a graphical algorithm, one can extract all possible computation graphs; see the example in the appendix. Given an initial state and a finite sequence of slices of a computation, it is straightforward to compute the final post-state. Thanks to Lemma 3.1, we have this:

Theorem 6.1. *For a given pre-state and finite computation meeting the above conditions, the identical post-state is obtained regardless of the choices of next stage taken along the way.*

7 Compositions

For composition and decomposition, we make use of the *drag* model of graph rewriting [17, 18], in which directed graphs also have partial, dangling edges coming in and out of vertices. Data and control flow from the roots towards the sprouts (the reverse of the convention in [17]).

Scenarios can be composed to form larger scenarios. To *compose* two scenarios, we require a *switchboard*, which is a mapping that specifies which sprouts connect with which roots. For our purposes, roots and sprouts have identifiers (such as the edge they derive from when decomposed), so the switchboard matches them accordingly.

Moves (single slices or more) of a scenario can have both roots and sprouts, as does the slice on the right of the figure. And a switchboard can connect sprouts to roots of more than one scenario. Thus, in general, composition may create new cycles in the graph. A *sequential composition* is one in which the switchboard connects in one direction only, as in our example. A special case is when no connections are made and the two scenarios lie side by side, all events in one being incomparable control-wise to all events in the other.

This all gives us three modes of composition: (1) If S and T are scenarios, then $S\ ;\ T$ is their sequential composition, with sprouts of S hooking up with roots of T. (2) When there are no connections, and the events of S are independent of those in T, we write $S \parallel T$, which is identical to $T \parallel S$. (3) When S and T are each strongly-connected, meaning that the events in each are all simultaneous, then $S\ \&\ T$ $(= T\ \&\ S)$ is their strongly-connected composition, with some sprout of S connected to any root of T and vice-versa, making all the events in both simultaneous. Which sprout and which root matters not.

8 Graphical Programming

Until now, we have concentrated on the graphical description of individual computations. In reality, each computation is guided by some process that determines how it is to proceed depending on the particulars of the state and the vagaries of choices. Not every computational process, however, is algorithmic in the sense that the process is amenable to a finite and complete description.

The notion that algorithmic computations can be described as one big loop is one of the "folk theorems" of computer science [27], harking back to Kleene's normal form for recursive programs [31]. It is also central to the ASM formalism [24]. And it serves us here.

Given the description of an algorithm or a non-algorithmic procedure plus a starting state, we would want to be able to construct the corresponding computation. So, how shall we express algorithms and procedures? What better way than graphs that reflect all possible computations?

Processes may be described by graphs, which may be finite or infinite. Unlike computation graphs, alternative computation paths are included in process graphs, with the addition of vertices for conditional choice, as well as for the various forms of nondeterministic choice. Whereas an algorithm describes the choices to be made, behavioral graphs describe the consequences of each of those choices.

The program schemata we have seen so far are choiceless, and loopless, corresponding to straight-line programs. More general programs or procedures involve choices. Choice may be deterministic—involving tests and leading to only one possibility in any given state. Choice may also be nondeterministic, in which case there may be more than one possible outcome, as we will see.

Consider the case where a program segment is prefaced by a conditional. Then we must first of all include the graph of the evaluation of the conditional. Depending on the resultant value, the execution

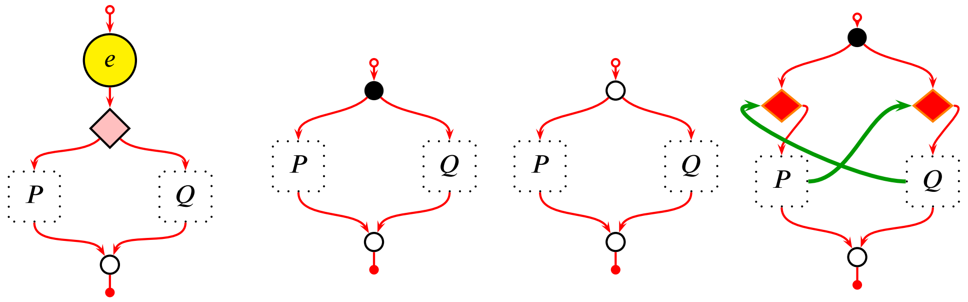


Figure 2: Choice constructs: (a) conditional; (b) angelic; (c) demonic; (d) angelic with bypass.

of a particular branch is included in the resultant computation. In any case, control passes from the conditional to the chosen branch.

The algorithm itself needs to include all alternatives, only one of which is taken each time that choice point is encountered during a computation. We are using a diamond for conditional choice. There should be control exits for each designated value, plus one for all other domain values.

The standard conditional programming construct, **if** e **then** P **else** Q **fi**, for Boolean-valued expression e, translates into the diagram in Fig. 2(a). The dotted rectangles represent subgraphs for algorithm segments, P and Q, only one of which is performed.

The conditional action described above decides which outgoing wire gets control based on the value of an incoming channel. In addition, we need to be able to probe a wire to see if it is live (hot) or not (cold) and decide how to proceed accordingly. We use a simple red diamond for this *probe* vertex, shown in Fig. 3(a). After control arrives on top, the (green-colored) control wire coming in from the left is probed. If the probe is active, then control continues below; if it's off, then control passes to the right. Thus, probes act like binary-valued channels. However, control need not have reached the probe for an off value to be seen.

It bears stressing that once probed, the outcome is fixed and won't change even if the probed wire changes status. So, only when the algorithm is ready to perform the test should power be provided on the upper incoming wire.

Since it is not a control wire that is used for sequencing, we color the probe green. In computations, we use dashed arrows for the wire when it is hot and dotted when it is cold. It is a mechanism that allows for lookahead to check if control already reached some point via another route. Indeed, a probe can create a loop. In a computation graph, we can use a dotted arrow to indicate that it was not live when probed.

We need logical connectives to obtain all manner of dependencies in computations. We have already seen the black-dot *junction*, with its all-in, all-out behavior. When it has only one exit, we call it a *conjunction*, since it acts like logical-and. For *disjunction*, we use a simple circle $\circ$, as in the previous diagram. If any of its incoming control edges is active, then control passes to the outgoing edge. Note that this is unlike anything we have seen till now, since the action does not require all incoming control lines to be active. There is an element of choice here: Control may choose to proceed through this vertex at any time as long as there is at least one live incoming wire. Bear in mind, that if—at a subsequent stage—control comes again to the disjunction along some other incoming wire, the disjunction is not taken again.

Negation is a simple instance of the probe with only the false exit. If the incoming probe is hot, the outgoing wire is cold; and vice versa. The probe is made after control reaches the vertex.

A conditional test chooses how control proceeds based on the value in some channel. The dual notion is to choose a value based on a probe of control. Consider the following assignment with a conditional

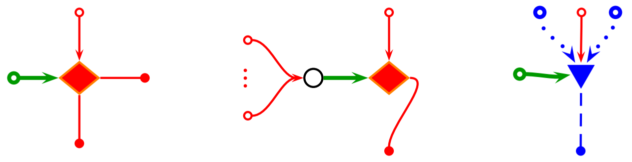


Figure 3: Gates: (a) control probe, (b) bypass probe, and (c) shuttle valve.

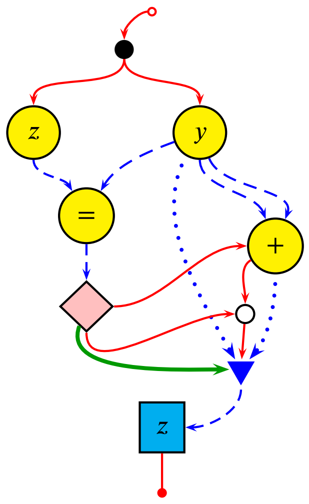


Figure 4: Graphical Python program with a conditional value and shuttle valve.

value—in Python syntax: $z = t$ **if** $z = (t := y)$ **else** $t + t$. We need a way to model the conditional choice of the value (y or $y + y$) to assign to z. We call this choice node a *(shuttle) valve*, depicted in Fig. 3(c). After (red) control arrives on top, the (green) control line on the left is probed. If it is on, then the value in the left incoming (blue dotted) channel is passed to the outgoing channel on the bottom; if off, then the value in the (blue dotted) right channel is passed on. In either event, control is passed on along with the chosen value. Of course, the relevant value must be available; the other value need not. With this type of action, the graph corresponding to the above Python program with its conditional value is as shown in Fig. 4. Only after the conditional test and the addition, if needed, is the valve used to channel the chosen value to the assignment.

Loops in a process may be indicated by a (thick dotted) dangling sprout at the bottom of program graph that connects back to a root at the top to form the loop. See examples in the appendix.

9 Nondeterministic Choices

We are particularly interested in modeling multiway nondeterministic choice, where there are more than one possible next state. Unlike probabilistic choice, which we ignore, nondeterministic choice is capricious with no assumed distribution for the alternatives.

Traditionally, two flavors of nondeterminism are considered: demonic and angelic. See [7], for instance, where the angelic/demonic terminology is attributed to Tony Hoare [33]. Choices can be made both between paths of action or between flows of values. In its rawest form, "angelic" means that a successful choice is made if there is one, while "demonic" means that a failing choice, when there is one, is always a viable option.

In *demonic* choice (this is the mild sense of "don't know"), one branch or the other is chosen for execution, a priori, oblivious of the possible outcomes of either choice. Only the chosen branch contributes to the actual behavioral graph. If any branch can go on forever, then the composite computation might, too. See Fig. 2(c).

In the alternate, *angelic* (don't care, erratic) version of behavior under choice, the "right" choice—if any—is always taken somehow. To implement such a choice, one can try alternatives until at least one completes. In that case, the control graph includes one complete computation and other (perhaps partial) ones—in parallel. If any one terminates, then the combined computation does, too. Control, in this case, passes from the successful alternative to the continuation of the computation. This kind of choice requires an infrastructure for communication if we don't want the computation to continue down useless, and potentially infinite, paths. If any interim results do not agree along the alternate paths, then provision must also be made for the chosen path's effects to win out.

Both varieties, demonic and angelic, of choice can be expressed in our framework. In the demonic case, one outgoing branch is chosen for the computation, regardless of what may happen as a consequence. In the angelic case, it may be that some branches are sterile, or it may be that different branches yield different outcomes. We model this by means of multiple incoming edges and/or channels. If control arrives via at least one, then control will pass onward. If multiple values arrive via channels, then exactly one is chosen to be passed on. See Fig. 2. (If part of the computation languishes, the exact form of the behavioral graph may depend on the vagaries of the execution mechanism, an eventuality we ignore here.)

So one should assume the worst when programming demonically. If some path leads to failure, that could be the one chosen, in which case the computation as a whole would fail. In any event, the computation needs only reflect the fact that an arbitrary choice was made. This is just a switch with a single root.

Angelic choice is something very different. The underlying idea is that a successful path is always taken, as long as any one of the choices leads to success. It is in this sense, for example, that a nondeterministic Turing machine accepts an input if any sequence of choices ends in acceptance. For angelic nondeterminism to work as advertised, all branches ought to be actively pursued (unless curtailed explicitly by the success or commitment of another branch; see below). This may be reflected simply as in Fig. 2(b). Both P and Q are started, and when either concludes, control passes to the exit on bottom. Nothing here dictates the speed in which the computations of the two alternatives proceed. However, there must not be any potential conflicts between P and Q, or else synchronization between them needs to be added.

The problem with plain angelic choice is that it isn't very angelic: unnecessary work is performed along alternate paths. In reality, traversing all paths simultaneously is often uncalled for. In Fig. 2(b), both P and Q will proceed uninterrupted until completion, even though once one completes, the other serves no purpose and could have been abandoned. Avoiding duplication can be achieved by means of a *bypassing probe*, as illustrated in Fig. 2(d). More generally, multiple success signals (control reaching the bottom) may be collected as in Fig. 3(b). If there are any, then computation along the branch with the probe is interrupted and instead it terminates (with no downward exit from the probe). Otherwise, computation proceeds downward.

One could, instead, entertain the idea of trying alternatives in some order, one by one, and backtracking to the next alternative only upon failure. This only works if no path leads to an infinite computation that does not succeed, while another succeeds, since then this "depth-first" approach can miss its objective. Assuming this is not the case, some management is still required. Should some store-changing actions along the failing or unnecessary path be undesirable, then the program must undo them.

In angelic contexts, explicit failure means that other alternatives—if any—are to be explored. Failure,

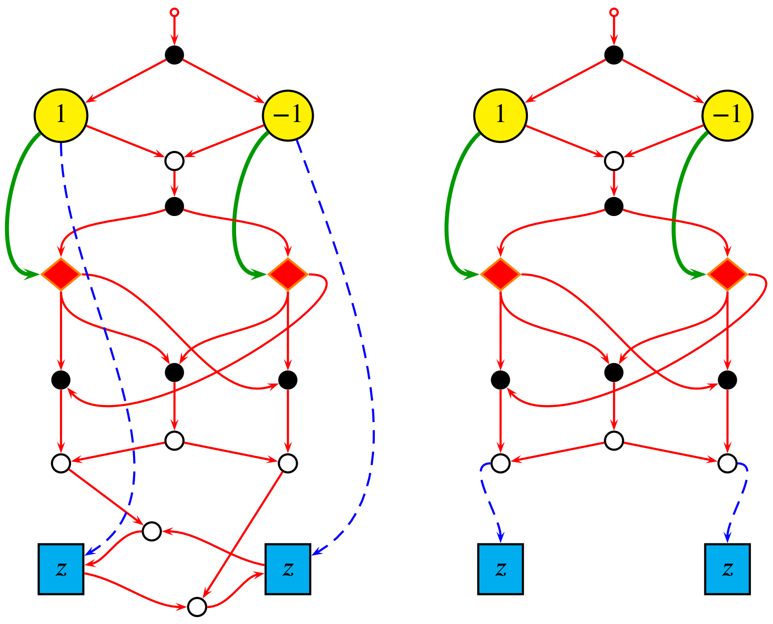


Figure 5: Two types of choice: (a) first come, first serve; (b) winner takes all.

however, is not a primitive, since it requires communication. Nonetheless, we want to ensure that we can implement failure as well as explicit success (or commitment). Signalling failure should cause another alternative to be pursued. It should also terminate the failing path. This can be achieved easily by a disjunction of failure points, as shown above. If failure is detected at any point in the path of one alternative, then the next alternative obtains control. Success, on the other hand, means that no other alternative needs not be pursued.

In this way, if a program is trying several alternatives at the same time—as in an angelic setting—and at some point one alternative succeeds, then the rest of that computation may proceed, while the other contingent actions should be blocked. Moreover, at that point, unexecuted preparatory steps of the disregarded alternatives can also be abandoned.

Choice can be combined with other atomic actions to provide various nondeterministic constructs that are of use. Consider our erstwhile incompatible assignments $z := 1 \parallel z := -1$ from the Introduction. We have disallowed any possibility of parallel components with conflicting writes (Section 6). So, they cannot be performed together; one must precede the other. Hardware normally precludes both assignments executing simultaneously. Simultaneous accesses to a single location are eschewed; an order on competing reads and writes involving the same memory location will be imposed at the hardware level. All the same, the behavioral graph needs to reflect the intended semantics, by forcing an ordering of the writes one after the other. But which of the two possible orders need not be specified. This is depicted in Fig. 5(a).

Another type of choice is provided by a gate that lets only one control line through even if both are active. When only one is active when probed, the corresponding action is chosen, and the other is abandoned. This is shown in Fig. 5(b).

All said and done, given a graphical algorithm and an initial state, it is straightforward to derive the actual computation and the associated sequence of state changes.

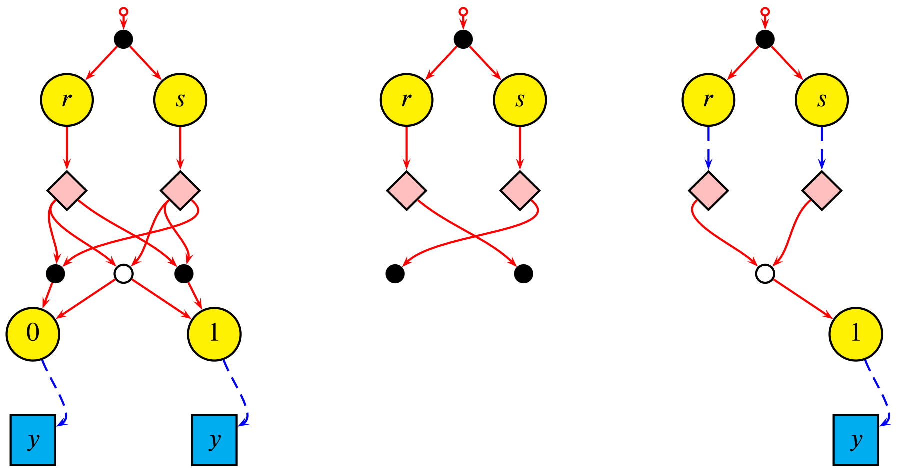


Figure 6: Guarded conditional. (a) Program on the left; (b) computation when both guards are false; (c) when both are true and the right path is taken.

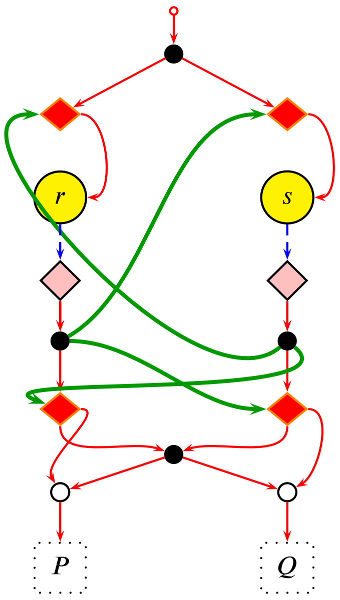


Figure 7: Avoiding unnecessary tests using probes. If r (s, resp.) has tested true before s (r, resp.) is examined, then the latter test is skipped. In any case, only one of P and Q obtains control.

10 Guarded Commands Revisited

Specifying the intent of Dijkstra's guarded commands [20] requires a mix of conditional, sequential, parallel, and demonic composition.

An implementation of the guarded conditional **if** $r \rightarrow y := 0 \mathbin{[\!]} s \rightarrow y := 1$ **fi**. might test r before s, or vice-versa, or both simultaneously. The latter case (common in the literature), namely, that first all the conditions are evaluated and then an oblivious choice is made among the enabled branches, translates into the procedure on the left of Fig. 6. No choice is needed when only one branch is enabled; a conjunction suffices. If neither r nor s is true, then only the control lines in the middle subfigure are taken. If both are true, then a choice must be made, as in the computation on the right.

In general, the number of different orders in which the guards may be examined is counted by the exponentially-growing Fubini numbers [39, #670]. It is infeasible to program each alternative separately, and choose among them. The nondeterministic paradigm proposed here obviates that issue.

If one wants more realistically to allow for the possibility that only one of the tests, r and s, needs to be executed when it is found to succeed, then probes can be brought into play, as shown in Fig. 7. Once one of them succeeds, the other is blocked—provided it hasn't already been taken. When both are tested and found true, then a nondeterministic choice is made as to how to continue.

Consider now a three-case conditional **if** $p \to s \ [\!] \ q \to t \ [\!] \ r \to u$ **fi**. Suppose p and q are true but r diverges. With a breadth-first, angelic policy, exactly one of s and t is executed, and control passes on to the next statement. With a depth-first, non-angelic policy, an additional possibility is that neither is executed, and the whole statement diverges on account of r. In either case, failure of the chosen branch, be it s or t, would mean failure of the whole conditional.

Similar interpretations can be given to Dijkstra's guarded **do** loop, repeating its body until no guard is found to be true. Dijkstra eschewed fairness; see [3, §§4,8]. A limited degree of fairness can be achieved with an appropriate implementation [23].

11 Retrospective

We have built upon an enormous body of prior research and design. Control edges and data edges have been around since the advent of flowcharts ("process charts") circa 1921 by Lillian and Frank Gilbreth [21]. This led to the ASME standard, with operation and flow process charts, in 1947 [4]. Control and data edges are also prominent in the Petri net formalism for asynchronous computation, first conceived by Petri in 1939 [36, Foreword]. Data edges also made appearances in Karp and Miller's work in 1966 [30], in Sutherland's graphical data-flow programming system of 1966 [40], and subsequently with Dennis's Data Flow Graphs [14] and other enterprises. Combinations of control and value edges are to be found in various specification languages, including the Structured Analysis and Design Technique (SADT) in 1969 [38], the operational semantics of the Vienna Design Language (VDL) around the same time (see [32]), the Programmer's Apprentice project at MIT in the late 1970s [37], and Plotkin's Structural Operational Semantics from 1981 [35]. Vertices in these formalisms represent complex operations in general. The literature on distributed systems studies control graphs and the states represented by cuts in depth; for one discussion in the 1980s, see [34].

We gained much insight into the fundamentals of algorithmic computation from abstract state machines (ASMs), conceived by Yuri Gurevich in 1993 [24]. In their most basic manifestation, ASMs are sets of conditional parallel assignments, expressing a single step of an algorithm. Gurevich [25] showed that any sequential (deterministic, non-interactive) algorithm, be it effective or conceptual, can be described step-by-step and state-by-state by an ASM. The issue of axiomatizing and modeling, in that framework, contingent memory accesses and truly undefined instances of operations, such as division by zero, which—whenever accessed—never respond (corresponding perhaps to nonterminating function calls), was explored and formalized in [6, 5]. That work also suggested an intra-step partial ordering of memory accesses, which helped inspire the control quasi-ordering here. ASMs with bounded nondeterminism have been studied [26]; asynchronous ASMs were investigated in [11]. The fundamentals of interaction were also studied in [8, 9, 10] within that framework.

References

[1] Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3. Clarendon Press, Oxford, 1994. URL: `https://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf`.

[2] Oana Andrei and Hélène Kirchner. A rewriting calculus for multigraphs with ports. *Electronic Notes in Theoretical Computer Science*, 219:67–82, 2008. Proceedings of the Eighth International Workshop on Rule Based Programming (RULE 2007). URL: `https://www.sciencedirect.com/science/article/pii/S1571066108004295`, `doi:10.1016/j.entcs.2008.10.035`.

[3] Krzysztof R. Apt and Ernst-Rüdiger Olderog. Nondeterminism and guarded commands. In Krzysztof R. Apt and Tony Hoare, editors, *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, volume 45 of *ACM Books*. Association for Computing Machinery, New York, 2022. URL: `https://arxiv.org/pdf/2310.09004v1.pdf`.

[4] ASME. *Operation and Flow Process: Developed by the A.S.M.E. Special Committee on Standardization of Therbligs, Process Charts, and Their Symbols*. American Society of Mechanical Engineers, New York, 1947.

[5] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. Exact exploration. Technical Report MSR-TR-2009-99, Microsoft Research, Redmond, WA, 2009. URL: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Partial.pdf`.

[6] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. Exact exploration and hanging algorithms. In *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, volume 6247 of *Lecture Notes in Computer Science*, pages 140–154, Berlin, Germany, August 2010. Springer. URL: `https://www.microsoft.com/en-us/research/uploads/prod/2009/01/201-Exact-Exploration-and-Hanging-Algorithms.pdf`, `doi:10.1007/978-3-642-15205-4_14`.

[7] Andreas Blass and Yuri Gurevich. The logic of choice. *Journal of Symbolic Logic*, 65(3):1264–1310, September 2000. URL: `http://research.microsoft.com/en-us/um/people/gurevich/Opera/135.pdf`, `doi:10.2307/2586700`.

[8] Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, Part I. *ACM Transactions on Computational Logic*, 7(2):363–419, April 2006. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.5100&rep=rep1&type=pdf`, `doi:10.1145/1131313.1131320`.

[9] Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, Part II. *ACM Transactions on Computational Logic*, 8(3), July 2007. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.5100&rep=rep1&type=pdf`, `doi:10.1145/1243996.1243998`.

[10] Andreas Blass and Yuri Gurevich. Ordinary interactive small-step algorithms, Part III. *ACM Transactions on Computational Logic*, 8(3), July 2007. Article 16. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.3181&rep=rep1&type=pdf`, `doi:10.1145/1243996.1243999`.

[11] Egon Börger and Klaus-Dieter Schewe. Communication in abstract state machines. *Journal of Universal Computer Science*, 23(2):129–145, February 2017. URL: `http://www.jucs.org/jucs_23_2/communication_in_abstract_state/jucs_23_02_0129_0145_boerger.pdf`.

[12] Jacques Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, January 1988. `doi:10.1145/35043.35045`.

[13] Jaco W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall International, Englewood Cliffs, NJ, 1980.

[14] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, Berlin, 1974. Springer. `doi:10.1007/3-540-06859-7_145`.

[15] Nachum Dershowitz. The generic model of computation. In *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zürich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science, pages 59–71, 2012. URL: `https://arxiv.org/pdf/1208.2585.pdf`, `doi:10.4204/EPTCS.88.5`.

[16] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Hand-*

book of Theoretical Computer Science, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990. Draft at URL: https://www.cs.tau.ac.il/~nachum/papers/survey-draft.pdf.

[17] Nachum Dershowitz and Jean-Pierre Jouannaud. Drags: A compositional algebraic framework for graph rewriting. *Theor. Comput. Sci.*, 777:204–231, 2019. doi:10.1016/j.tcs.2019.01.029.

[18] Nachum Dershowitz, Jean-Pierre Jouannaud, and Fernando Orejas. Drag rewriting. HAL archive, March 2023. URL: https://inria.hal.science/hal-04029105.

[19] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965. doi:10.1145/365559.365617.

[20] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. doi:10.1145/360933.360975.

[21] Frank B. Gilbreth and Lillian M. Gilbreth. Process charts: First steps in finding the one best way to do work. *The American Society of Mechanical Engineers*, December 1921. URL: http://www.archive.org/details/processcharts00gilb.

[22] Ralph E. Griswold and Madge T. Griswold. History of the Icon programming language. In *Preprints of the History of Programming Languages Conference (HOPL-II)*, volume 28 of *SIGPLAN Notices*, pages 53–68, New York, NY, March 1993. Association for Computing Machinery. doi:10.1145/155360.155363.

[23] Orna Grumberg, Nissim Francez, Johann A. Makowsky, and Willem P. de Roever. A proof rule for fair termination of guarded commands. *Information and Control*, 66(1):83–102, 1985. URL: https://www.sciencedirect.com/science/article/pii/S0019995885800140, doi:10.1016/S0019-9958(85)80014-0.

[24] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, 1995. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/103.pdf.

[25] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.3017&rep=rep1&type=pdf, doi:10.1145/343369.343384.

[26] Yuri Gurevich and Tatiana Yavorskaya. On bounded exploration and bounded nondeterminism. Technical Report MSR-TR-2006-07, Microsoft Research, January 2006. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2006-07.pdf.

[27] David Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980. doi:10.1145/358886.358892.

[28] David Harel. On the total correctness of nondeterministic programs. *Theoretical Computer Science*, 13(2):175–192, 1981. URL: https://www.sciencedirect.com/science/article/pii/0304397581900384, doi:10.1016/0304-3975(81)90038-4.

[29] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. International Series in Computer Science. Prentice Hall, 1990.

[30] Richard M. Karp and Raymond E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing related databases. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, 1966. URL: https://www.jstor.org/stable/2946247.

[31] Stephen C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936. doi:10.1007/BF01565439.

[32] Peter Lucas. Formal Semantics of Programming Languages: VDL. *IBM Journal of Devt. and Res.*, 25(5):549–561, 1981. doi:10.1147/rd.255.0549.

[33] Zohar Manna. Logics of programs. In Simon H. Lavington, editor, *Proceedings of the 8th IFIP Congress on Information Processing*, pages 41–51. North-Holland/IFIP, October 1980.

[34] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Parallel and Distributed Algorithms Conference*, pages 215–226, 1988. URL: https://homes.cs.washington.edu/~arvind/cs425/doc/mattern89virtual.pdf.

[35] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, 1981. URL: http://web.eecs.umich.edu/~weimerw/590/reading/plotkin81structural.pdf.

[36] Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985. Foreword by Carl Adam Petri. doi:10.1007/978-3-642-69968-9.

[37] Charles Rich, Howard E. Shrobe, and Richard C. Waters. Overview of the Programmer's Apprentice. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 827–828, Cambridge, MA, August 1977.

[38] Douglas T. Ross. Structured Analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–34, January 1977. doi:10.1109/TSE.1977.229900.

[39] Neil J. A. Sloane. *A Handbook of Integer Sequences*. Academic Press, 1973.

[40] William Robert Sutherland. *The On-Line Graphical Specification of Computer Procedures*. Ph.D. dissertation, Massachusetts Institute of Technology, 1966. URL: https://dspace.mit.edu/bitstream/handle/1721.1/13474/25697177-MIT.pdf.

[41] Alan M. Turing. Intelligent machinery. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7. Edinburgh University Press, 1969. Reprinted unpublished 1948 report for the National Physical Laboratory. URL: https://www.npl.co.uk/getattachment/about-us/History/Famous-faces/Alan-Turing/80916595-Intelligent-Machinery.pdf.

A Example: Greatest Common Divisor

Imagine a state that stores two integer-valued variables, x and y, and also incorporates the wherewithal to perform standard operations on the integers. Fig. 8 is the graphical description of a program that repeats the following instructions:

- If $x \geq y > 0$, then subtract y from x.
- If $x \leq y$, then swap the values of x and y.
- If both conditions hold (i.e. $x = y > 0$), then do either action (but not both).
- If neither condition holds ($x > y = 0$), then halt.

This program sometimes terminates with the greatest common divisor (g.c.d.) of the initial values of x and y as the current value of x (when $y = 0$). Or it may (but need not) loop forever with $x = y$, in which case their common value is the g.c.d.

In the graph version shown in the figure, the red arrows indicate flow of control and the ones, flow of data. Yellow circular vertices are memory reads; blue square vertices are writes; pink diamonds are conditional branches; black circles are conjunctions, all incoming are needed before all outgoing pass control; and white circles are disjunctions, with only one outgoing edge getting control when one or more incoming edges are live. All this will be explained in detail later on.

The graphical program minimizes memory access: The memory location holding 0 is accessed only once during the whole computation; x and y, once per iteration. When both guards hold, and $x = y > 0$, an arbitrary choice is made (at the circular vertex in the middle) whether to subtract y from x or to swap the two.

Suppose, for example, that $x = 3$ and $y = 6$. Starting at the top, the values of 0, x, and y are retrieved from the store. The comparisons $6 > 0$, $3 \geq 6$, and $3 \leq 6$, reading from left to right, yield T, F, and T, respectively. Taking the appropriate exits from the tests below the comparisons, gives the situation displayed in Fig. 9(right). There are no directed paths between those comparisons or between those tests, so the relative order in which they are made is immaterial. Control reaches the two assignment boxes

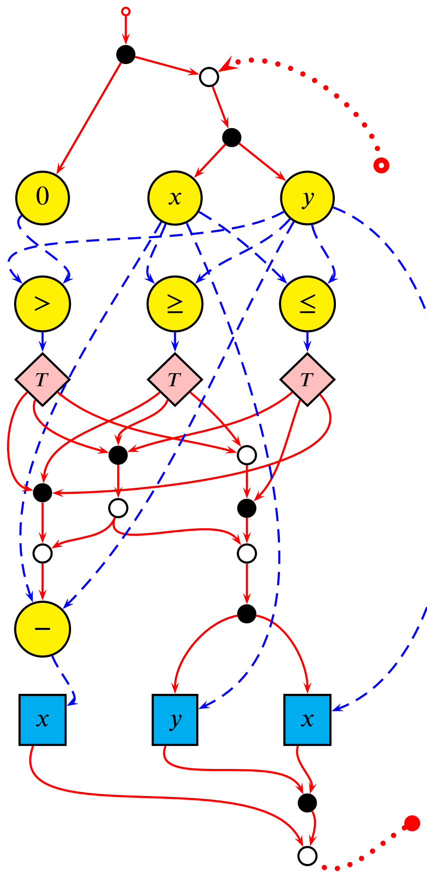


Figure 8: A graphical scheme representing a version of Euclid's greatest common divisor algorithm by repeated subtraction that need not halt. Control lines are solid red; data channels with control are dashed blue; black dots are conjunctions; white dots are disjunctions; blue squares are assignments; yellow circles are memory accesses; and pink diamonds are conditional branches (testing for truth, T). The thick dotted root and sprout indicate the repeated loop.

near the bottom, and $y := 3$ and $x := 6$, again with no order imposed. Vertices in the program that are not reached in the computation have been erased.

In the above version, no attempt is made to avoid testing a guard even if another branch has been committed to. Adding tests that probe control wires to determine whether the need for a test has been obviated, results in a graph algorithm, as shown in Fig. 10, with lots of extra wiring in the middle. The solid red rhombi near the top are conditionals that transfer control depending on whether the incoming (green) thick line, called a "probe", is live (hot) or not (cold). When a branch is found to be unnecessary, either because another branch has succeeded or because one conjunct has already been found false, a choice is made whether to go ahead regardless. In the sample iteration in Fig. 11, for $x = 6$ and $y = 3$, the fact that the tests $y > 0$ and $x \geq y$ are tested first and hold true allows the program to skip testing $x \leq y$ altogether. The store gets updated with $x := 6 - 3$.

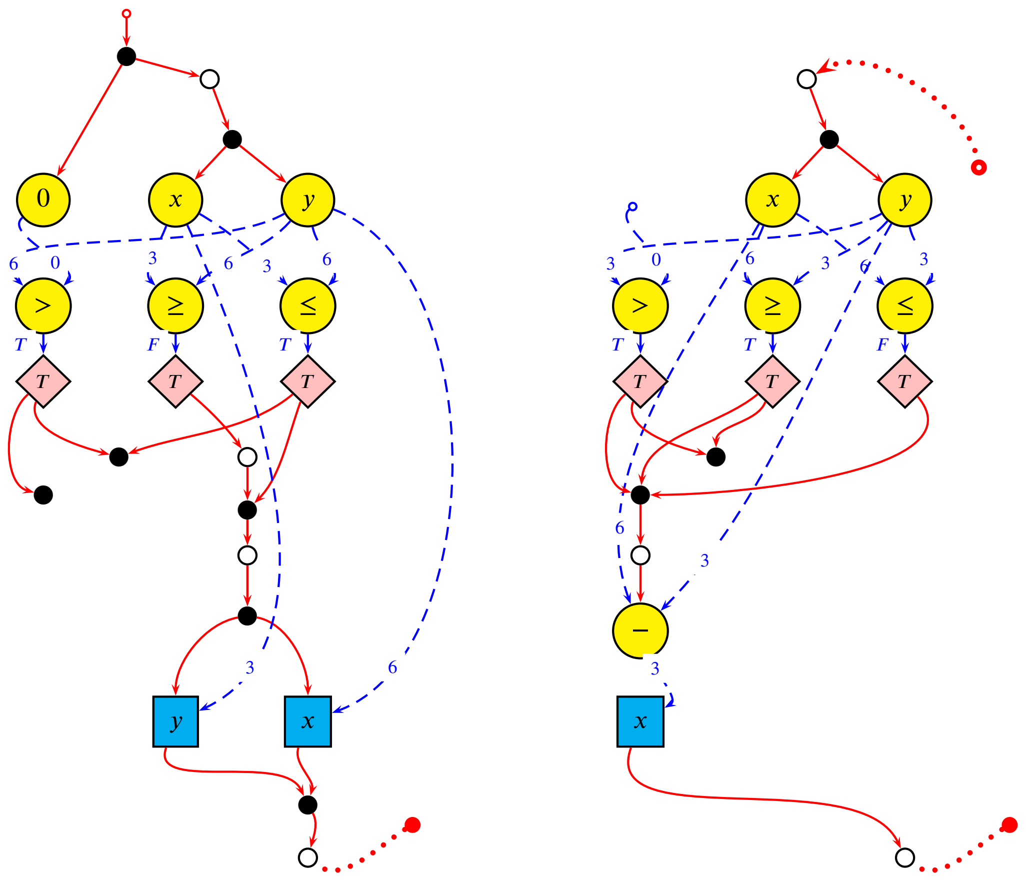


Figure 9: The first (left) and second (right) iterations of the algorithm in Fig. 8, with $x = 3$ and $y = 6$ at the outset. The first swaps the values so that $x = 6$ and $y = 3$; the second subtracts y from x, leaving $x = y = 3$.

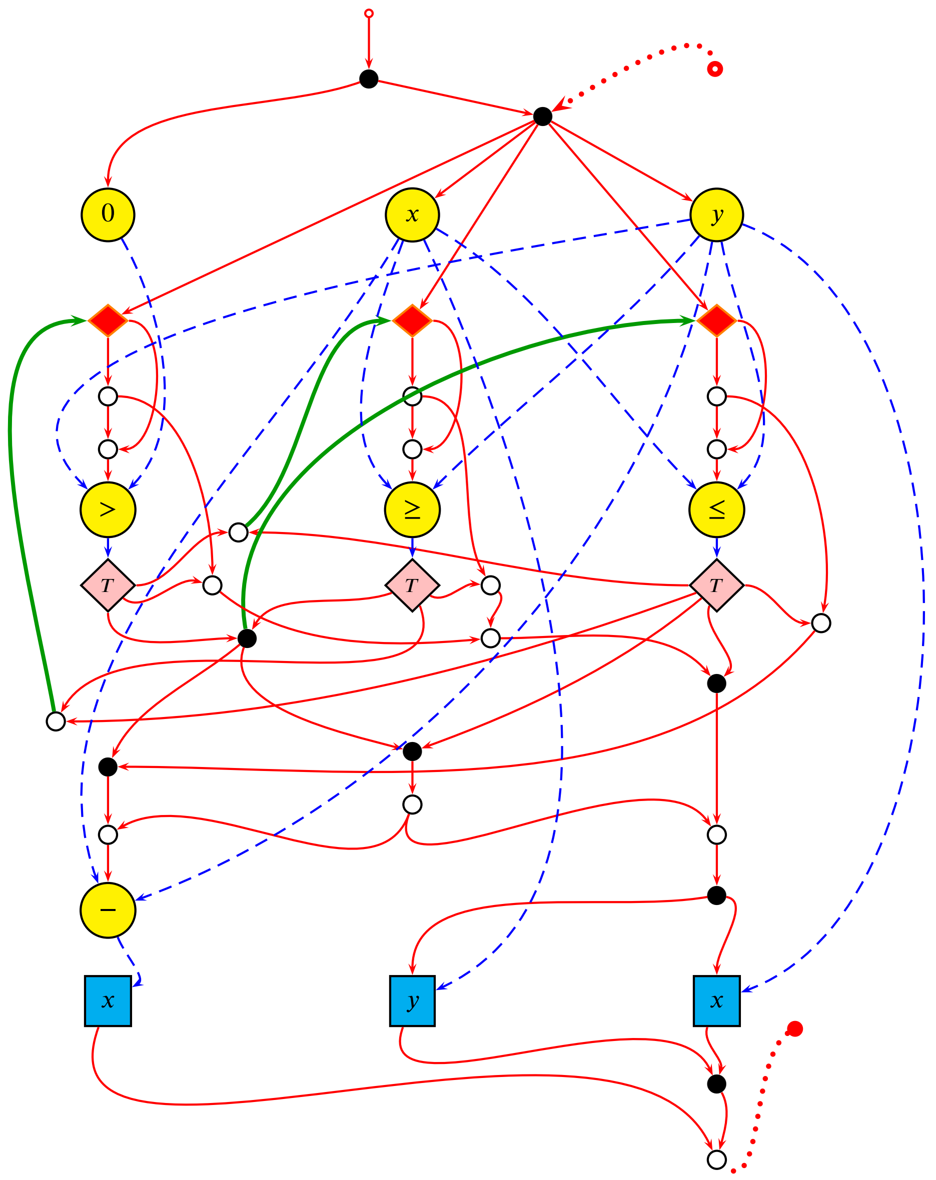


Figure 10: A spaghetti-like version of greatest common divisor that has the option of not performing all comparisons each iteration. If the left (center) comparison is performed first and fails, then the middle (left) path has the choice whether to test anyway. If the rightmost is performed and succeeds, then the other two have the option of skipping. Probes of control lines are thick green.

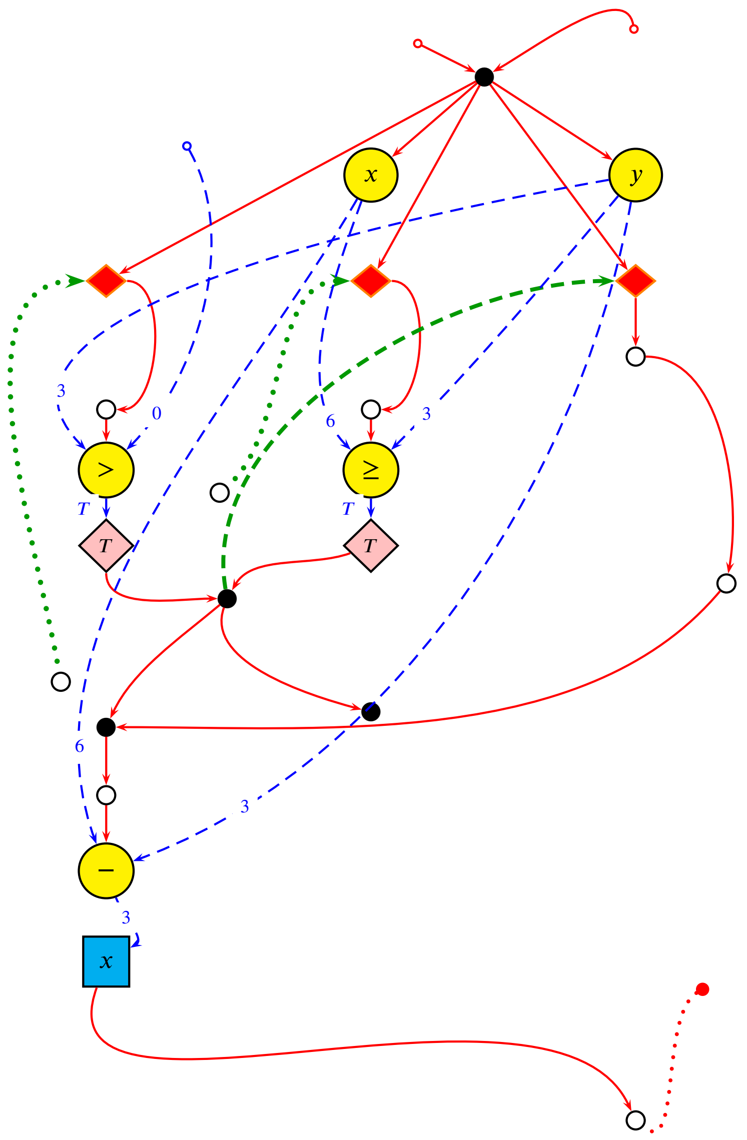


Figure 11: The second iteration, with $x = 6$ and $y = 3$ at the beginning and $x = y = 3$ at the end. Hot probes are dashed; cold ones are dotted.