# The multiresolution analysis of flow graphs

Steve Huntsman

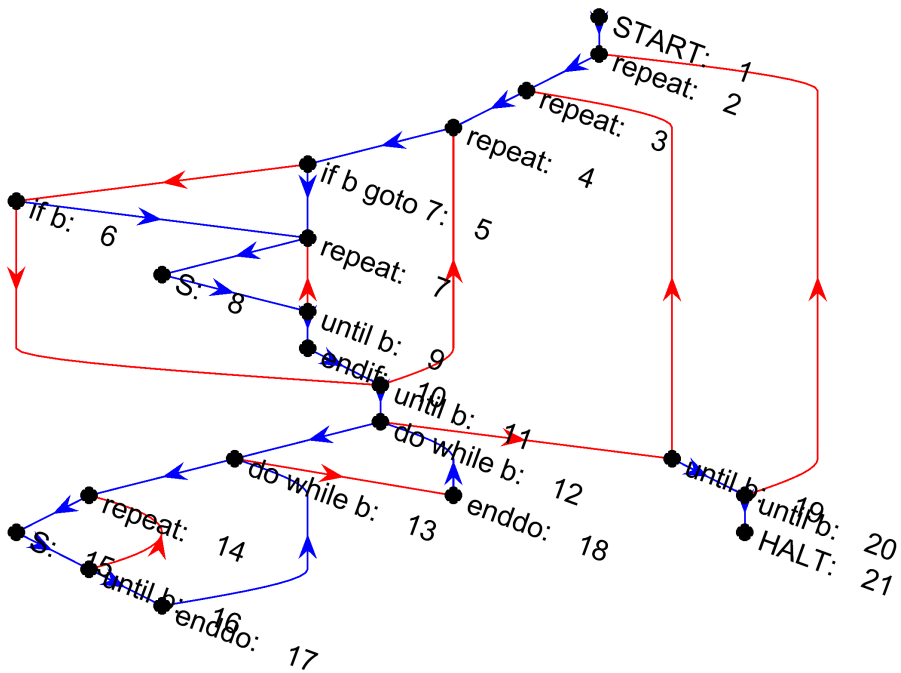
Cyber and Communications Technologies (goo.gl/DRnf3e)

5 November 2017

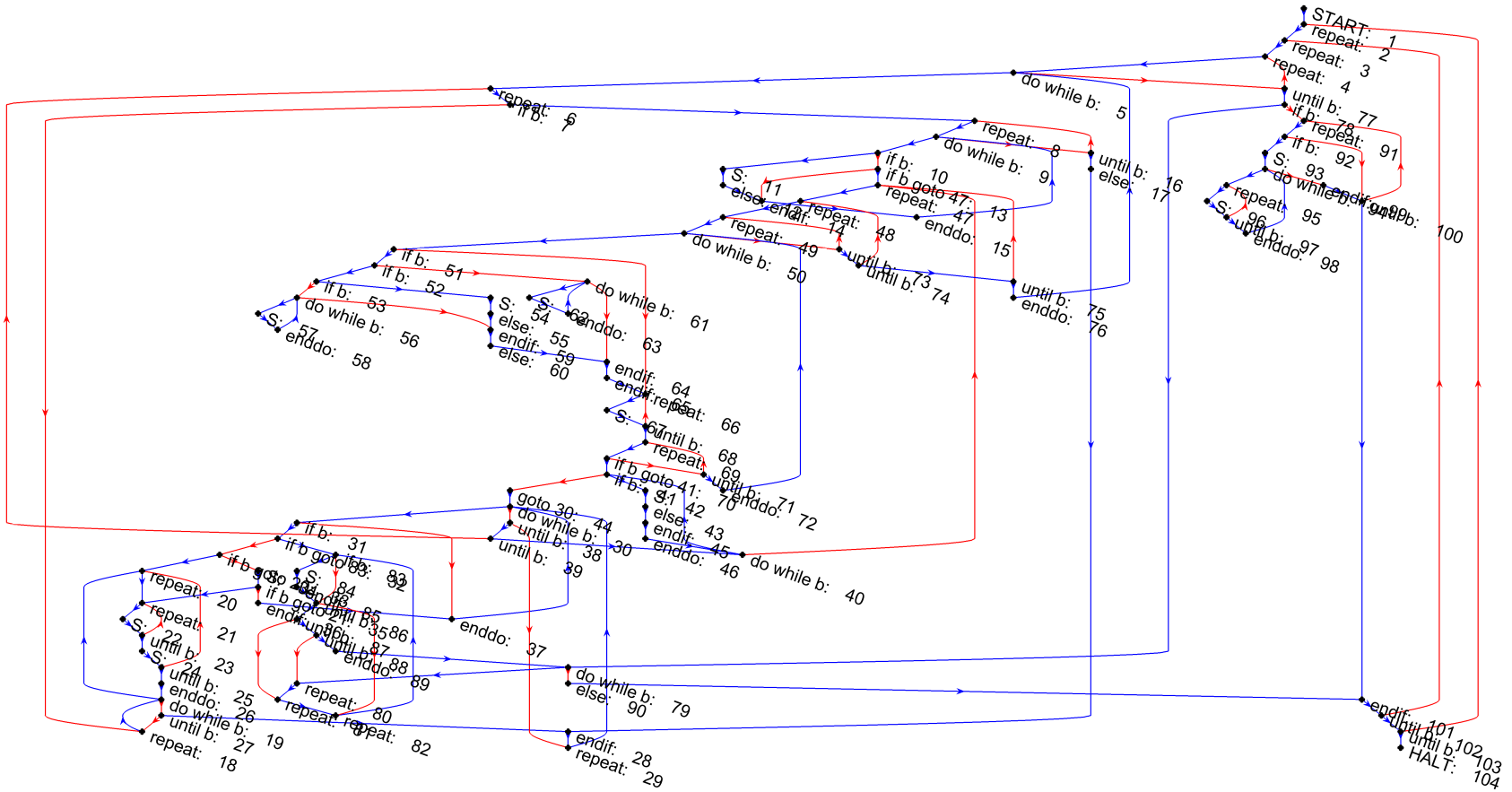
# A control flow graph (CFG) for a tiny program

```
1   START
2   repeat
3       repeat
4           repeat
5               if b goto 7
6               if b
7                   repeat
8                       S
9                   until b
10              endif
11          until b
12          do while b
13              do while b
14                  repeat
15                      S
16                  until b
17              enddo
18          enddo
19      until b
20  until b
21  HALT
```

Each S is its own statement or subroutine; each b is its own Boolean predicate; branches are colored according to whether or not their corresponding b evals to ⊤ or ⊥



# Another (still small) CFG



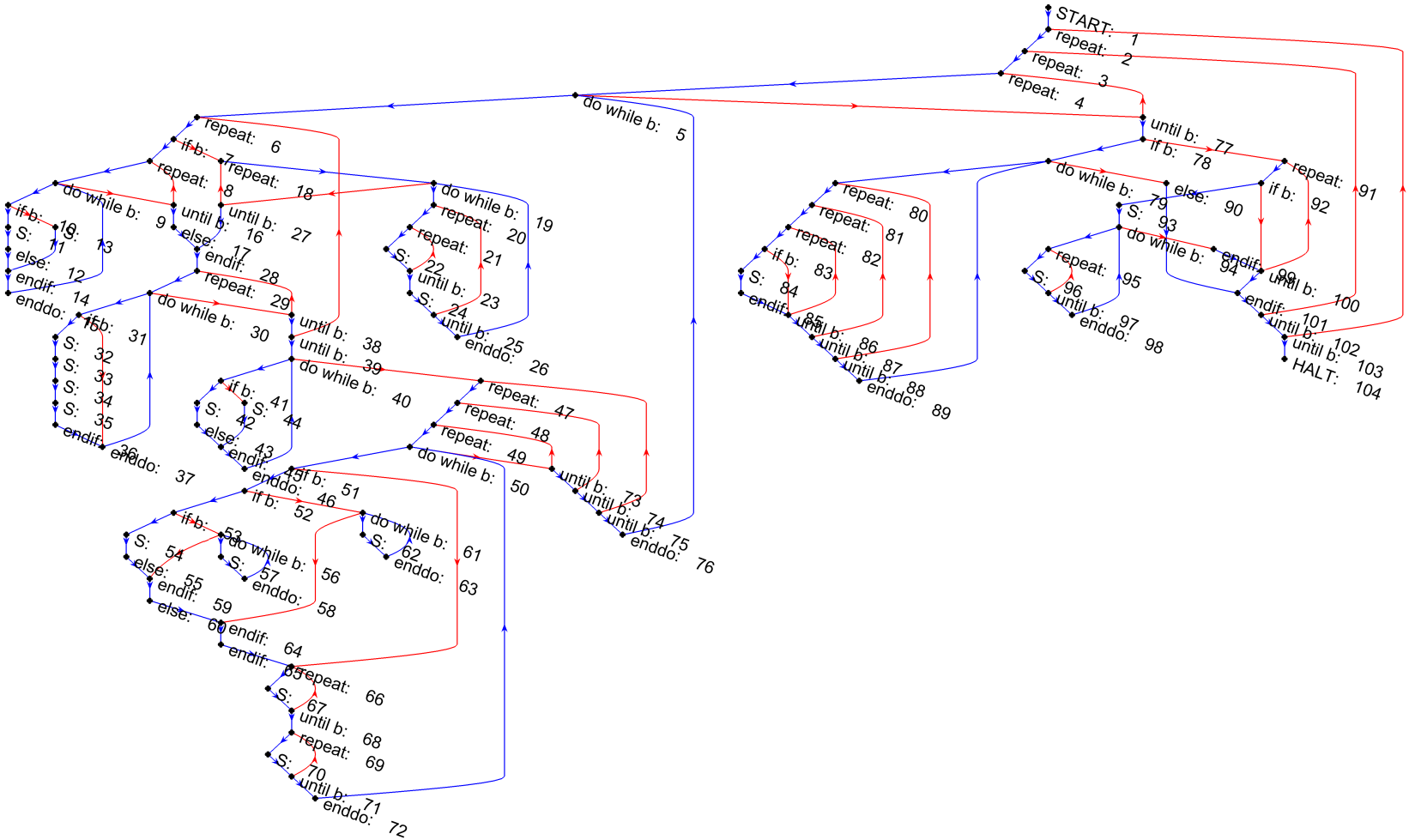



BAE SYSTEMS
INSPIRED WORK

# Background and motivation

- Machine code of any program of practical interest is much larger than examples shown above
- A ray of sunshine: code can be restructured to eliminate `gotos` [Zhang and D'Hollander, 2004]
  - Effective version of Böhm-Jacopini structured program theorem
  - Dovetails with the constructions we discuss here
- Subroutines are programs in their own right
  - Recursively (de)compose programs: multiresolution analysis
  - Much more interesting when trying to parallelize source or reverse engineer binary code than when merely parsing Python
- Similar considerations inform myriad other domains where flow graphs are good process models

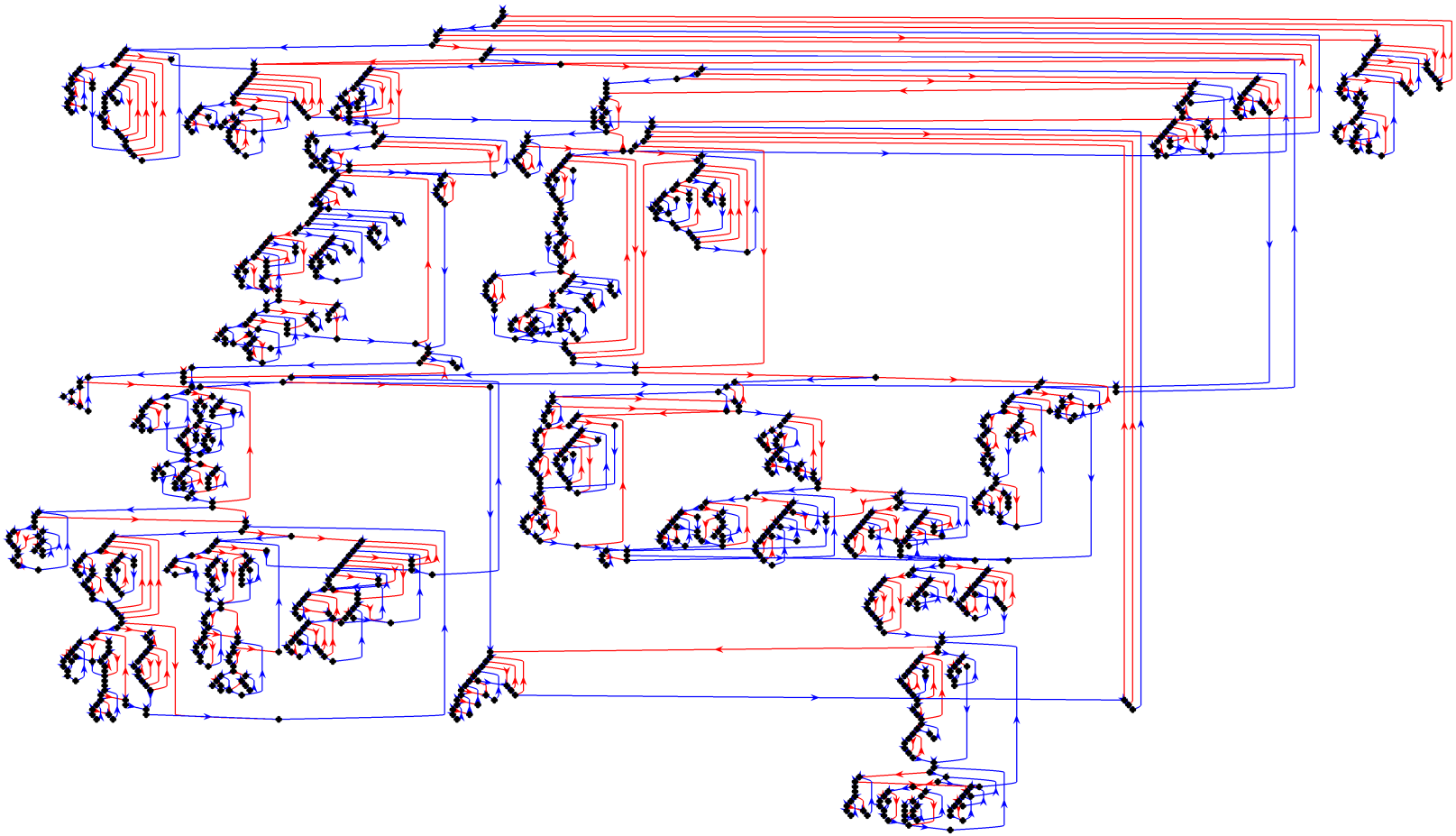
# A CFG with no gotos





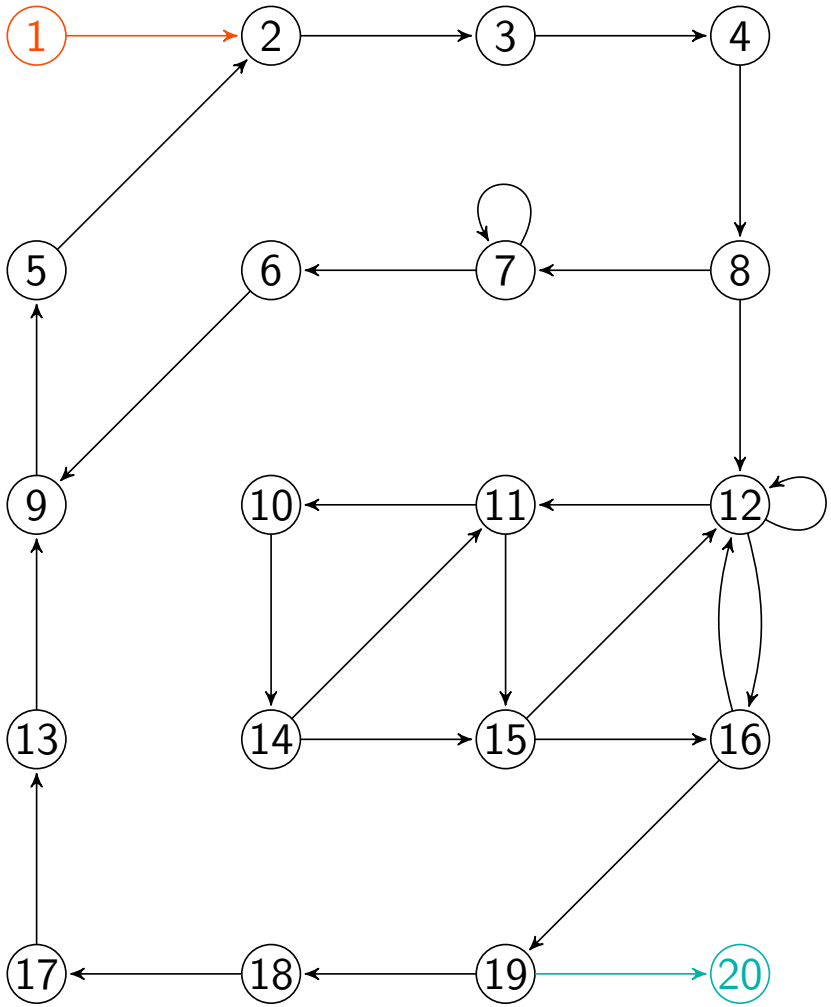

BAE SYSTEMS
INSPIRED WORK

# A CFG with no gotos



# Definition. A *flow graph* is a digraph such that:

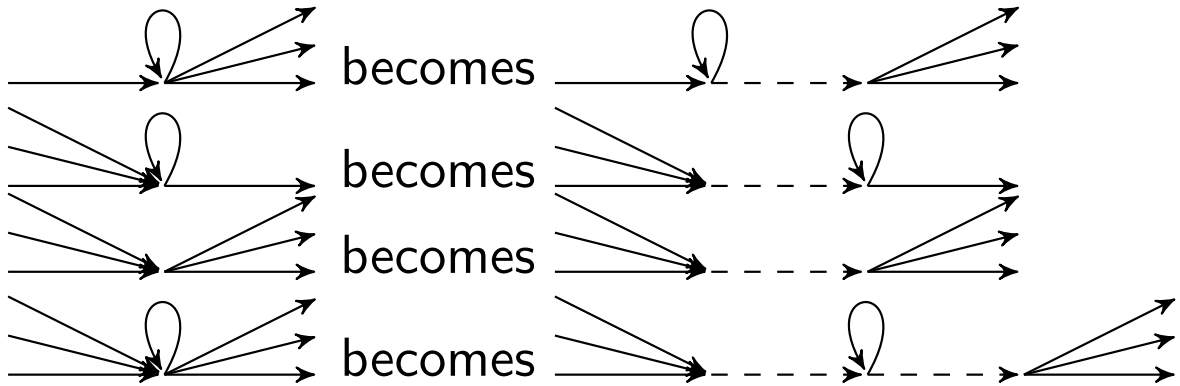
- There is a unique source (indegree = 0) vertex and a unique target (outdegree = 0) vertex
- There is a unique edge (entry) from the source vertex and a unique edge (exit) to the target vertex
- Identifying the source and target vertices yields a strongly connected digraph
  - Trivial case: entry = exit
  - Precludes, e.g. `do while b; goto <num>; enddo`



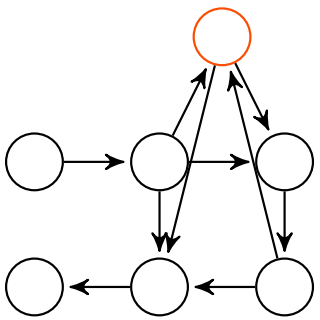
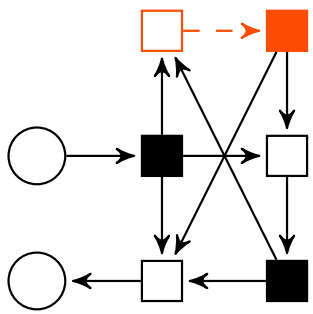
# Single-entry/single-exit regions

- Definition. Let $G$ be a digraph and $j, k \in V(G)$: $j$ dom $k$ iff every path from a source $s$ in $G$ to $k$ passes through $j$
  - Relation extends to edges; dual relation denoted pdom
- Definition. A *single entry/single exit region (SESER)* in a digraph $G$ is an ordered pair of edges $(e_1, e_2)$ s.t.
  - $e_1$ dom $e_2$
  - $e_2$ pdom $e_1$
  - a cycle in $G$ contains $e_1$ iff it contains $e_2$
- Notes
  - SESERs correspond to sub-flow graphs
  - $(e_1, e_1)$ is a degenerate SESER
  - A nondegenerate SESER $(e_1, e_2)$ can also be specified by the vertex pair $(t(e_1), s(e_2))$, where $s(\cdot)$ and $t(\cdot)$ respectively denote the source and target of an edge
  - Very easy to find SESERs in DAGs, but requires some sophistication in general

# Stretching flow graphs is helpful

- Insert edges into a flow graph as follows:
  - 

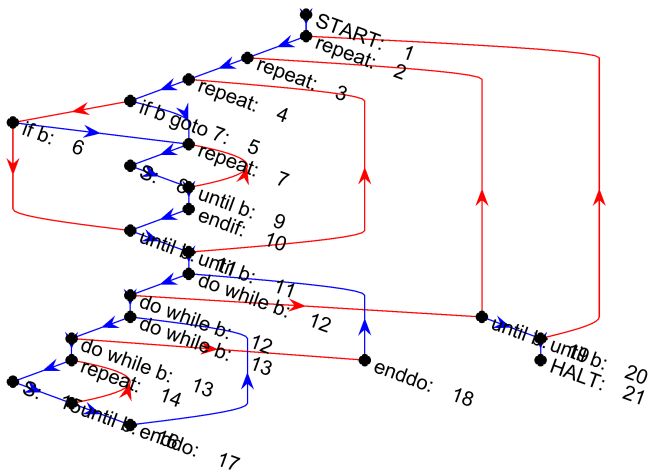
- Lemma. The resulting *stretching* is well defined
- There is a planar flow graph whose stretching is nonplanar:

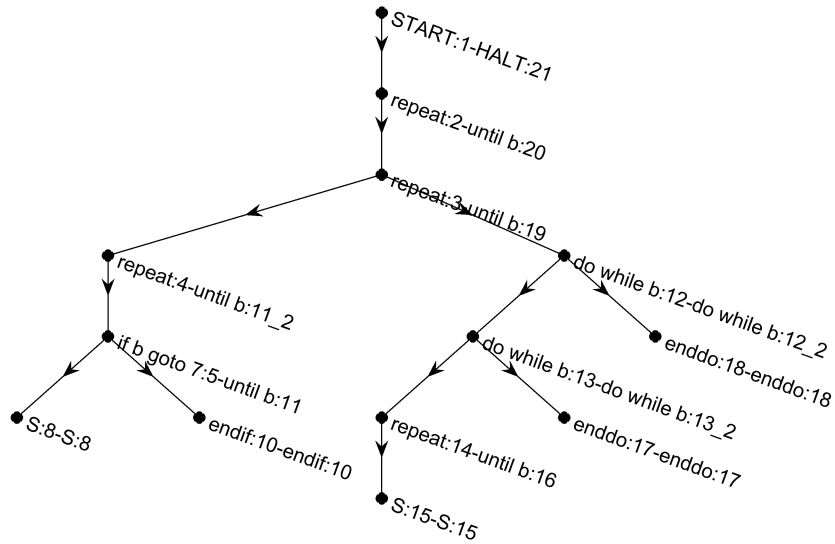

stretches to

which contains $K_{3,3}$

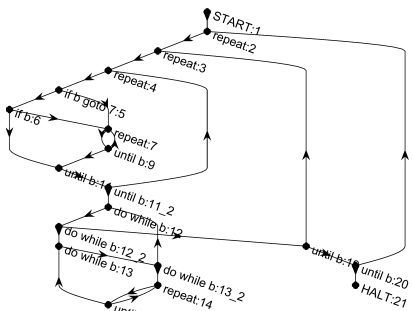
# The program structure tree

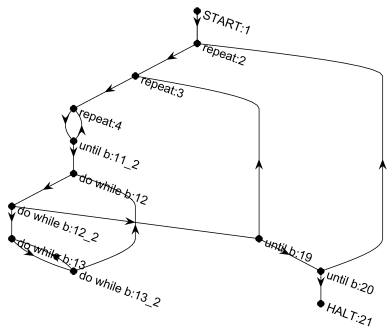
- Definition. The *interior* of a SESER $(e_s, e_t)$ is the set of vertices on paths from $t(e_s)$ that do not encounter $t(e_t)$
  - Different from flawed definition 6 of [Johnson, Pearson and Pingali, 1994]: §5 of [Boissinot *et al.*, 2012] illustrates this and why it matters
- Definition. A nondegenerate SESER $(e_1, e_2)$ is *canonical* if
  - For any SESER $(e_1, e_2')$ we have $e_2$ dom $e_2'$
  - For any SESER $(e_1', e_2)$ we have $e_1$ pdom $e_1'$
- Theorem [easily salvaged from JPP'94]. Interiors of distinct canonical SESERs are either disjoint or nested
  - "Canonical = minimal"
  - Inclusion relation induces the *program structure tree* (PST)

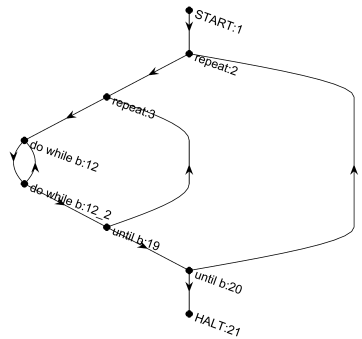
# Stretching, PST, and “coarsening” 1, 2, 3, & 6x



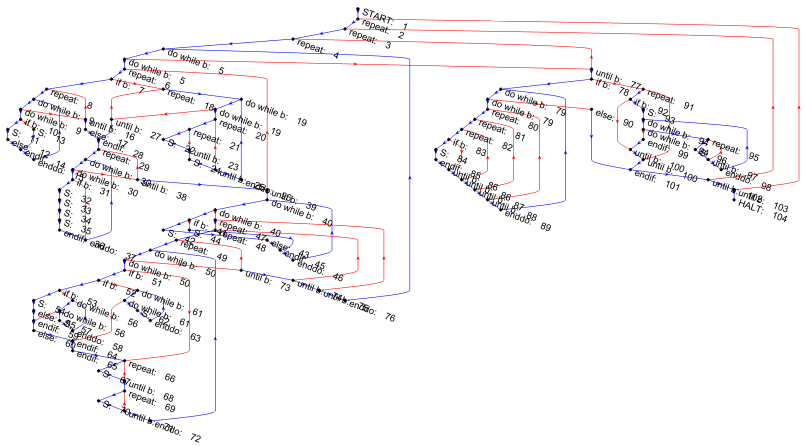



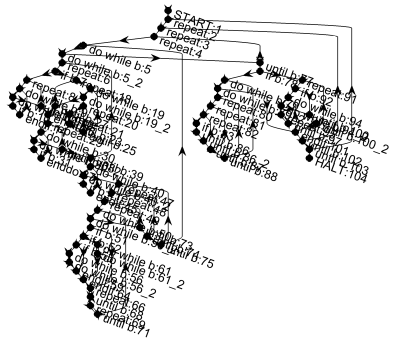


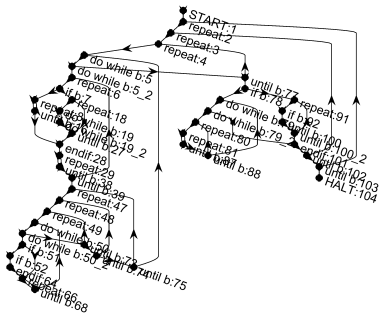


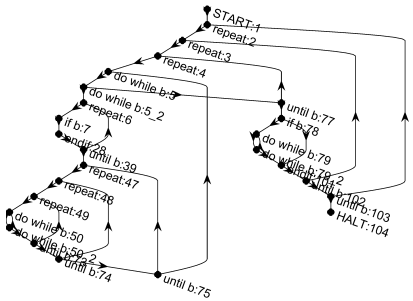
BAE SYSTEMS
INSPIRED WORK

# Stretching, PST, and “coarsening” 1, 3, 5, & 13x




# Introducing categories

- Definition. A *category* is a collection of *objects* endowed with *morphisms* between them that can be composed like functions
  - There is an identity morphism for every object
  - Composition is associative
  - Category theory = abstract math of systems engineering
- We will encounter some of the most fundamental structures:
  - Operadic: objects "plug into each other" like $f_{(m)} : X^m \to X$
    - $f_{(m)} \circ_\ell g_{(n)} := f(\cdot_1, \ldots, \cdot_{\ell-1}, g(\cdot_\ell, \ldots, \cdot_{\ell+n-1}), \cdot_{\ell+n}, \ldots, \cdot_{m+n})$
  - Monoidal (a/k/a tensor): objects and morphisms can be combined simultaneously
    - We'll actually see two varieties of this: sequential and parallel
  - Enriched: objects from another category are like morphisms
    - Sub-flow graphs

# The category of flow graphs

- There ought to be one that behaves like $\mathbf{Cob}_n$ or $\mathbf{Tan}_k$
  - Could have software with a mathematically nice API
  - Program analysis transformations should be morphisms
- Unfortunately, categories of digraphs are complicated
  - Problem: how to deal with loops [Brown *et al.* 2008]
  - Identifying vertices "should" induce a graph morphism, but edges must also be preserved, so any edges between identified vertices induce a loop
  - Insofar as loops in a coarse CFG ought to correspond to actual loops in the program, this behavior is bad
- Solution: treat loops and non-loop edges differently
- Resulting category **Dgph** is awkward to define but works
- **Flow** is the full subcategory whose objects are flow graphs

# Coarsening flow graphs

- Definition. For $j, k \in V$, the *absorption* of $k$ into $j$ is the morphism induced by identifying $j$ and $k$ and (if $k \neq j$) annihilating any loop at $j$ (by mapping it to the vertex $j$)
  - Definition chosen to dovetail with ideas of program abstraction
  - Absorbing $k, m$ into $j$ is equivalent to absorbing $m, k$ into $j$
  - For $G, H \in \mathrm{Ob}(\mathbf{Flow})$ with $H \subset G$, define the absorption of $H$ to be the image of absorbing the interior of $H$ into its source (considered as a vertex in $G$)
    - Amounts to replacing $H$ w/ single edge from source to target
- Definition. The *coarsening* $\circledcirc G$ is obtained by absorbing all of the sub-flow graphs corresponding to leaves of $\mathrm{PST}(G)$
- Observation: the pullback of $a \xrightarrow{g \circ f} c \xleftarrow{g} b$ is $a \xleftarrow{id} a \xrightarrow{f} b$
  - In particular, $f$ is the pullback of $g \circ f$ by $g$
  - We may therefore think of $\circledcirc G$ somewhat literally as a kind of pullback of $G$ by the leaves of $\mathrm{PST}(G)$

# The operad of flow graphs

- Let $P(n) := \{\text{flow graphs with } n \text{ ordered edges}\}$ and define

$$\circ : P(n) \times P(k_1) \times \cdots \times P(k_n) \to P(k_1 + \cdots + k_n)$$
$$(G, G_1, \ldots, G_n) \mapsto G \circ (G_1, \ldots, G_n)$$

  by replacing the $j$th edge in $G$ with $G_j$ in the obvious way
  - Edge ordering on $G \circ (G_1, \ldots, G_n)$ inherited from constituents
- Theorem. The triple $\{e, \{P(n)\}_{n=1}^{\infty}, \circ\}$, where $e$ denotes the trivial flow graph, forms a operad (in **Set**)
- $\circ$ and $\circledcirc$ are complementary:
  - Lemma. If $G \in P(n)$ and $\circledcirc G_j = e \neq G_j$ for $j \in [n]$, then $\circledcirc(G \circ (G_1, \ldots G_n)) = G$

# Tensoring flow graphs (series)

- Proposition. $(\mathbf{Flow}, \boxtimes, e)$ is a monoidal category, where $\boxtimes$ is defined below and the unit object $e$ is the trivial flow graph
  - $G \boxtimes G'$: identify exit edge of $G$ with entry edge of $G'$
  - For $f \in \mathbf{Flow}(G, G_f)$ and $f' \in \mathbf{Flow}(G', G'_{f'})$, we obtain $f \boxtimes f' \in \mathbf{Flow}(G \boxtimes G', G_f \boxtimes G'_{f'})$ by identifying the output of $f$ on the exit edge of $G$ with that of $f'$ on the entry edge of $G'$
- Proposition. For a flow graph $G$, we can form a category $\mathbf{SubFlow}_G$ enriched over $\mathbf{Flow}$ as follows:
  - $\mathrm{Ob}(\mathbf{SubFlow}_G) := E(G)$ (recall this excludes loops);
  - For $e_s, e_t \in \mathrm{Ob}(\mathbf{SubFlow}_G)$, $\mathbf{SubFlow}_G(e_s, e_t) \in \mathrm{Ob}(\mathbf{Flow})$ is the (possibly empty) flow graph with entry $e_s$ and exit $e_t$;
  - The composition morphism is induced by $\boxtimes$;
  - The identity element is determined by the trivial flow graph
- Unlike $\mathbf{Free}(G)$, $\mathbf{SubFlow}_G$ is always finite and we can build it

# Tensoring flow graphs (parallel)

- Theorem. $(\mathbf{Flow}, \otimes, e)$ is a monoidal category, where $\otimes$ is defined below and the unit object $e$ is the trivial flow graph
  - $G \otimes G' := G \sqcup G' / \sim$, where $\sim$ is mostly obvious but has messy technical details to account for the cases where entry and exit edges are either identical or adjacent
    - $\sim$ always identifies entry edges
    - $\sim$ identifies exit edges if interiors are unaffected...
    - ...otherwise $\sim$ collapses a "small" factor to make things work
    - Constraints on how to fill in these technical details are perhaps the main benefit of invoking category theory *ab initio*
  - Let $[\cdot]$ denote an equivalence class under $\sim$ and set

$$(f \otimes f')(k) := \begin{cases} [(f(j), 0)] & \text{if } k = [(j, 0)] \\ [(f'(j'), 1)] & \text{if } k = [(j', 1)] \end{cases}$$

  along with an implied extension to edges

# Denouement

- $\boxtimes$ corresponds to sequential execution
- $\otimes$ corresponds to an `if` (or parallel execution)
-  $\circ\,(e, \bullet, e, e)$ corresponds to a `do while` or `repeat`
- By the structured program theorem and an effective version thereof, we have a category-theoretical framework for (de)composing structured programs up to statement/predicate vertex labels and $\top/\bot$ edge labels
  - Exercise: eliminate the "up to" disclaimer
- Requiring that flow graphs exhibit various category-theoretical desiderata places strong but satisfiable restrictions on them that can usefully inform the architecture of program analysis platforms, program synthesizers, compilers, etc.

# Thanks

steve.huntsman@baesystems.com