R-customizers
Transitive Closure Simplification
Graph Customizers and Instances
Traversal Histories: Regularity and Similarity

Goal: define relation between graph and its customizers, study domains of adaptive programs, merging of interface class graphs

Definition

- R-customizers(I) = {T : (T R I)}
- Customizer Theorem: (I1 R I2) if and only if R-customizers(I1) $\subseteq$ R-customizers(I2) provided R is reflective and transitive
- I1 large graph, I2 small graph
- R: compatible, refinement, strong-refinement

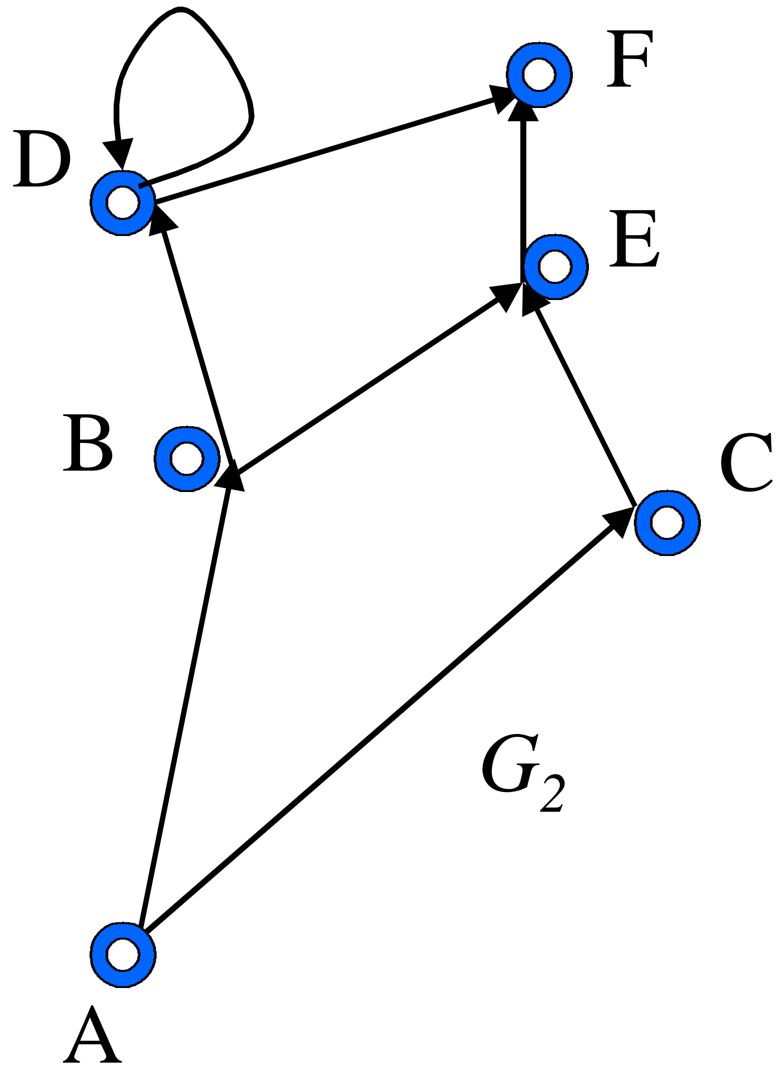
Example for Customizer Theorem

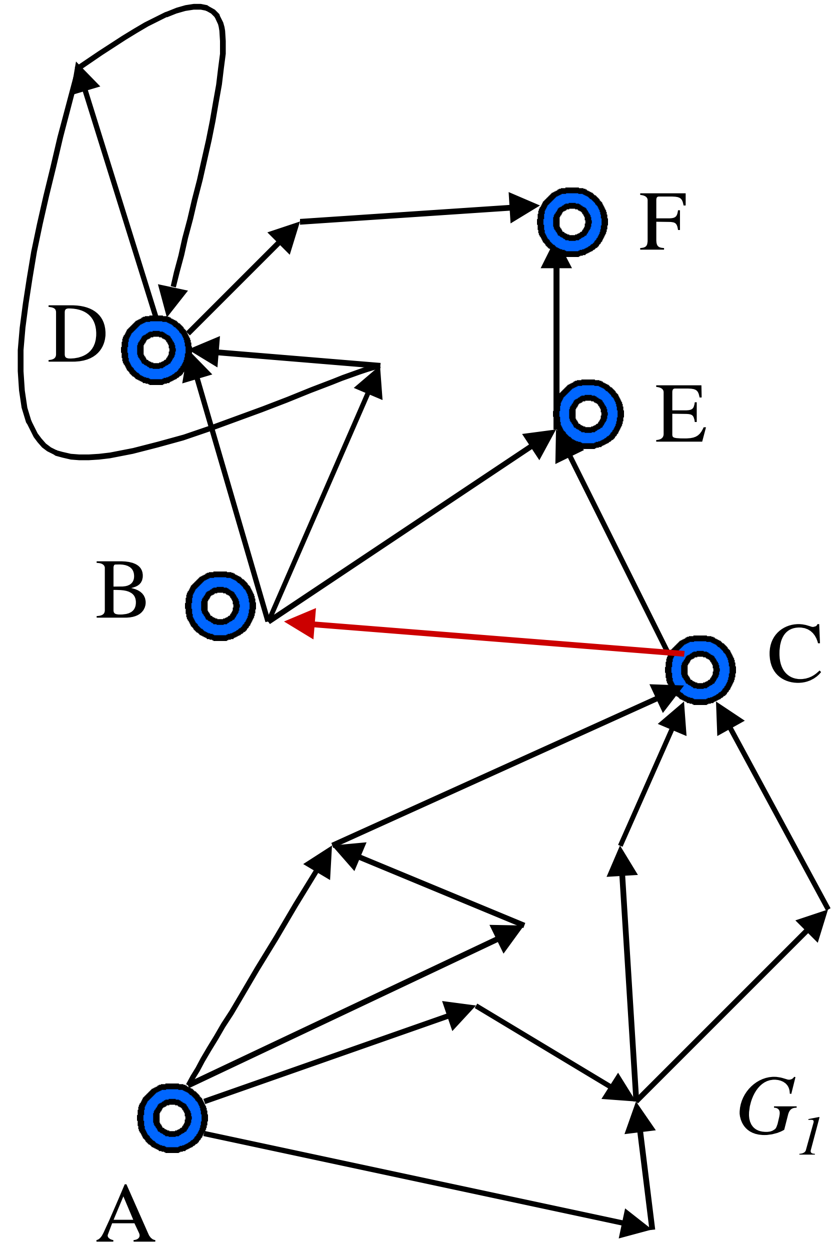
- I2 : A = B.
- I1 : A = B. B = C.
- R = compatible
- R-customizers(I2) = class graphs that have an A-node and a B-node with a path
- R-customizers(I1) $\subseteq$ R-customizers(I2)
- I1 is compatible with I2

Key concepts: compatability

- Let $G_1 = (V1,E1)$ and $G_2 = (V2,E2)$ be directed graphs with $V2$ a subset of $V1$. Graph G_1 is a ***compatible*** with G_2 if for all u,v in V_2 we have that (u,v) in E_2 implies that there exists a path in G_1 between u and v
- Polynomial.
- Motivation: All of G_2 is used in G_1 .

G_1 compatible G_2

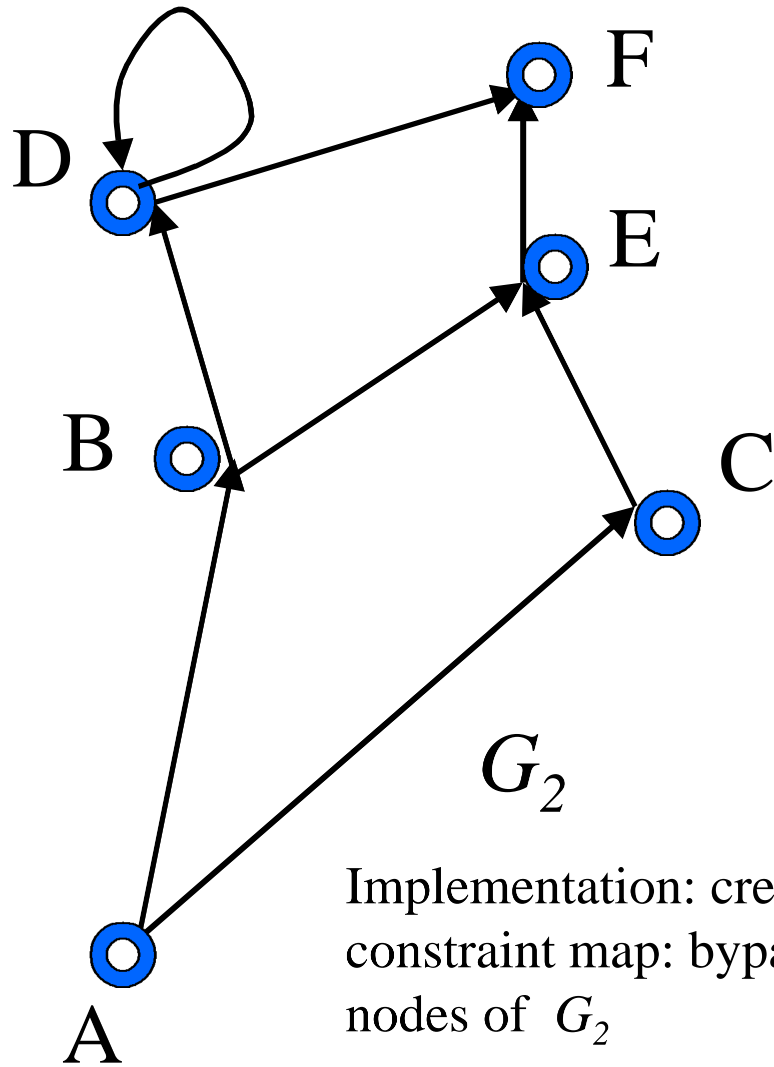


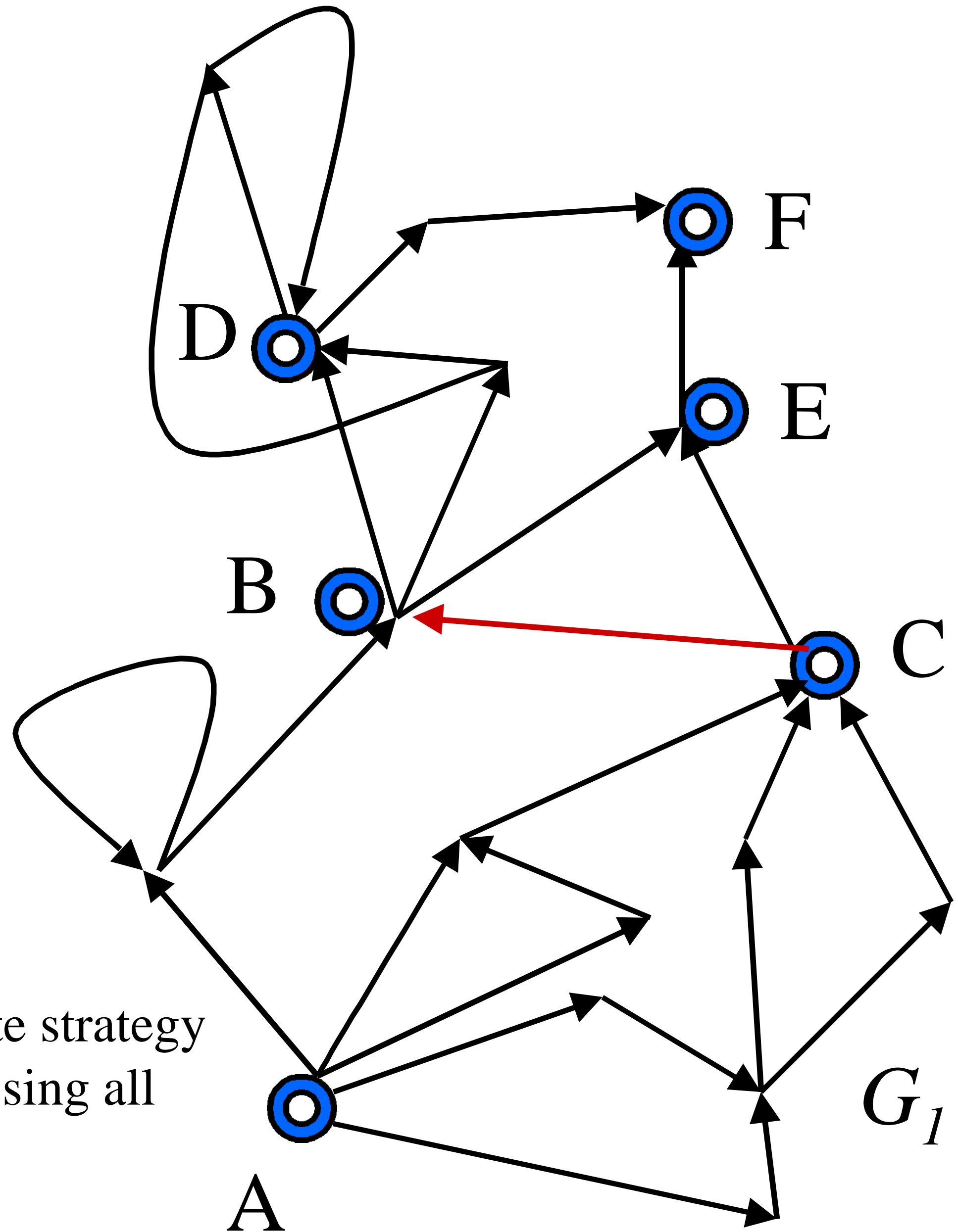


Key concepts: refinement

- Let $G1=(V1,E1)$ and $G2=(V2,E2)$ be directed graphs with $V2$ a subset of $V1$. Graph G_1 is a ***refinement*** of G_2 if for all u,v in V_2 we have that (u,v) in E_2 implies that there exists a path in G_1 between u and v which does not use in its interior a node in V_2.
- Polynomial.
- Motivation: No surprises.

G_1 **refinement** G_2





Implementation: create strategy constraint map: bypassing all nodes of G_2

Motivation for Refinement

- Refinement has nice implications on instances of G_1 and G_2 (consider the graphs to be class graphs). By contracting edges of instances of G_1 without eliminating G_2 nodes and by deleting parts from instances of G_1 we can transform any G_1 instance to a G_2 instance.
- G_1 objects are *similar* to G_2 objects.

Motivation for Refinement

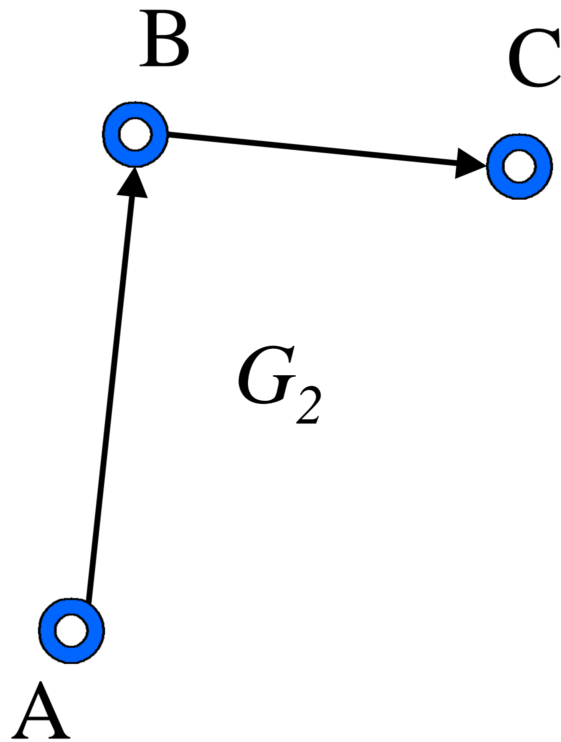
- Extra paths in G_1 can be eliminated during traversal.
- Similarity between G_1 and G_2 objects helps to guarantee that program for G_2 works correctly when applied to G_1.

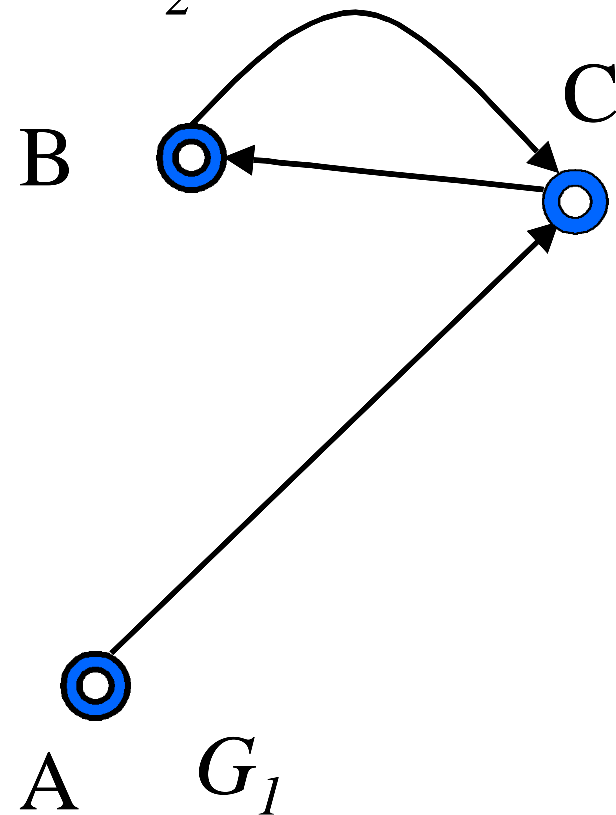
Refinement means: no surprises

not G_1 strong refinement G_2

not G_1 refinement G_2

G_1 compatible G_2

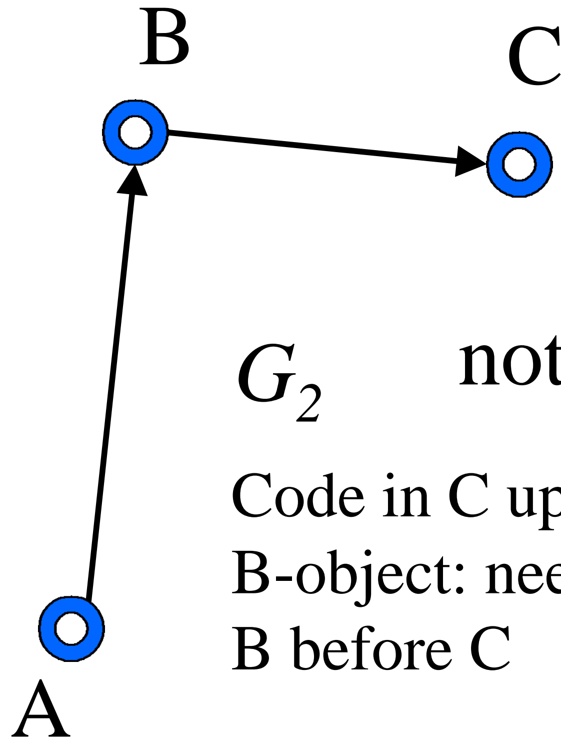




No Refinement: objects are not similar

aA(
 aB(
  aC()))

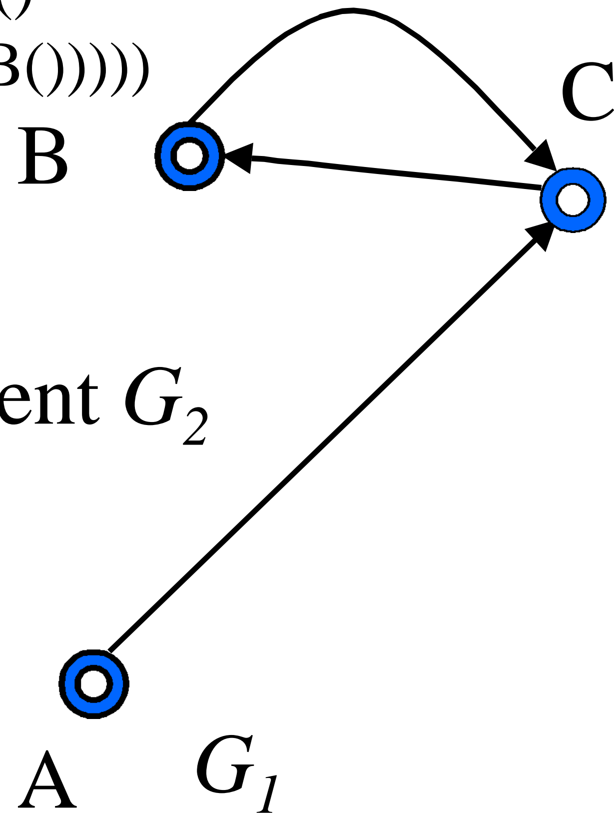
aA(
 aC(
  aB()
   aC()
    aB())))))



G_2

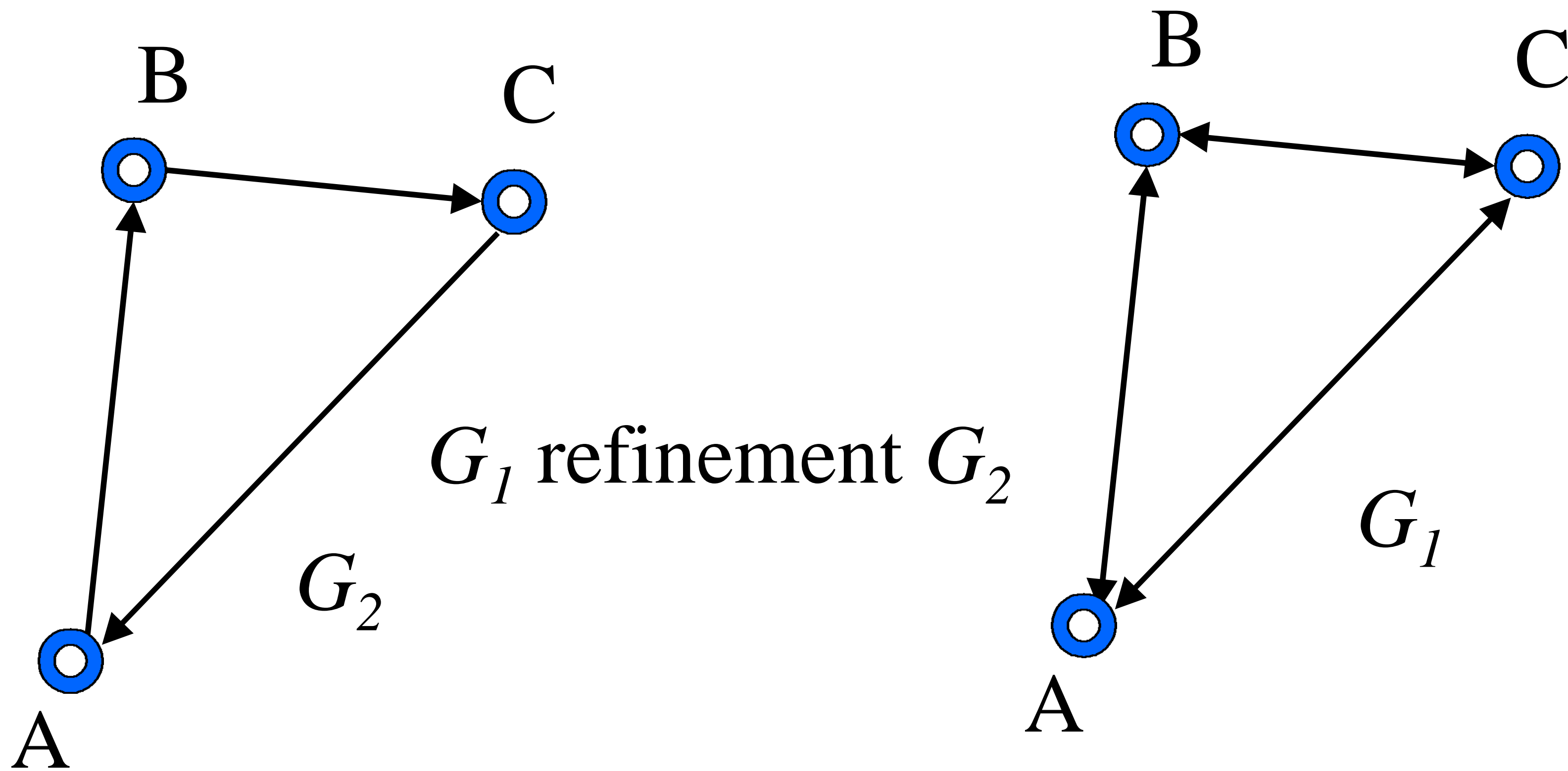
not G_1 refinement G_2

Code in C updates the B-object: need to visit B before C



G_1

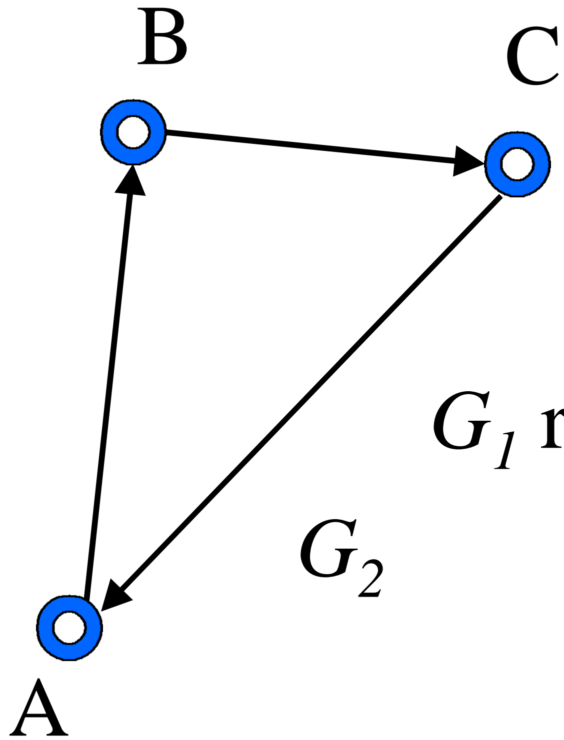
Refinement means: no surprises



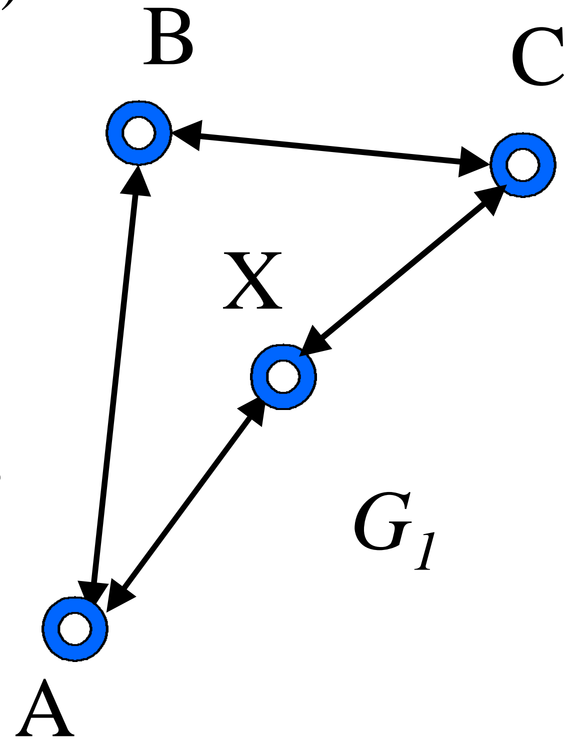
aA(
aB(
aC(
aA()))))

aA(
aB(
aC(
aX(
aA())
aB())
aA())
aX()))

contracting



G_1 refinement G_2



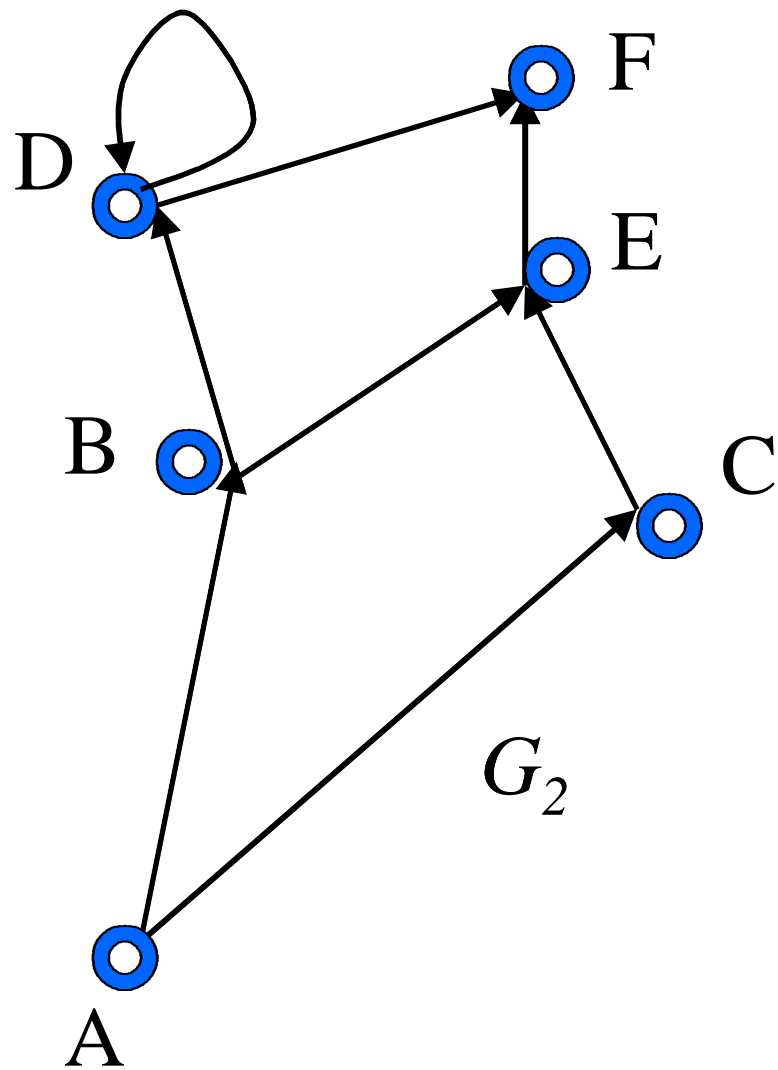
Instantiation view

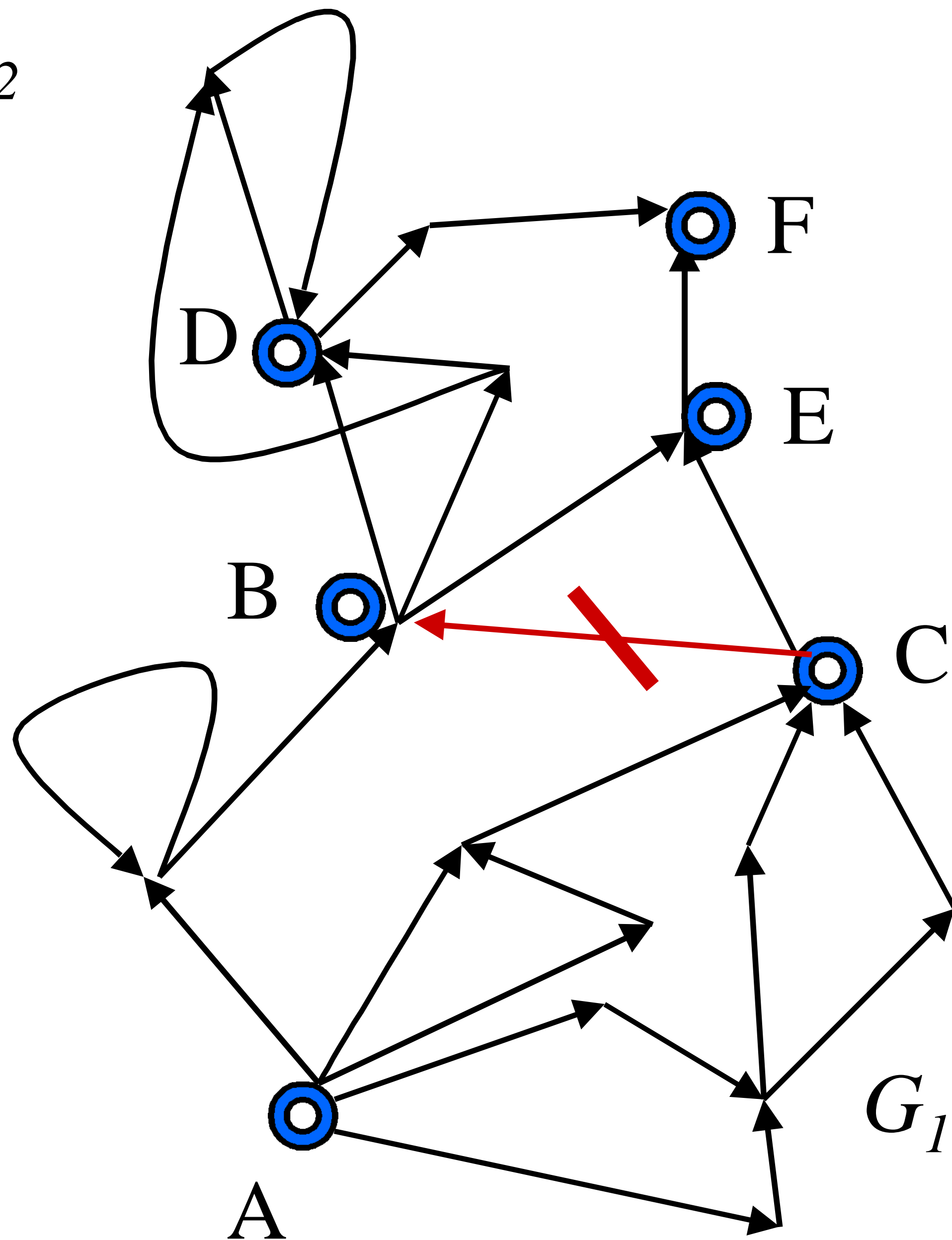
- Interface class graph: instances are class graphs
- Class graph: instances are objects
- Want objects to “conform” to interface class graph: objects of class graph and immediate objects of interface class graph must be similar

Key concepts: strong refinement

- Let $G1=(V1,E1)$ and $G2=(V2,E2)$ be directed graphs with $V2$ a subset of $V1$. Graph G_1 is a ***strong refinement*** of G_2 if for all u,v in V_2 we have that (u,v) in E_2 *if and only if* there exists a path in G_1 between u and v which does not use in its interior a node in V_2.
- Polynomial.
- Motivation: no surprises, no extra paths

G_1 strong refinement G_2





Connections

- Strong refinement implies Refinement implies Compatible
- When do we use which relationship?

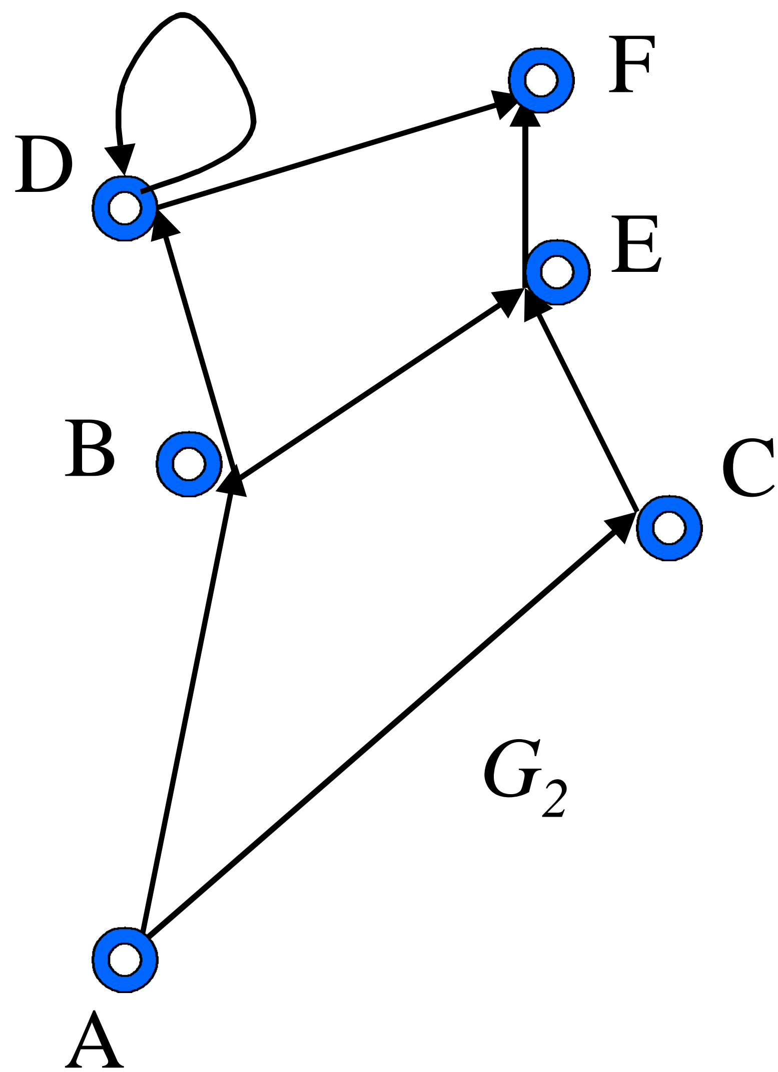
Roles graphs play in OOD

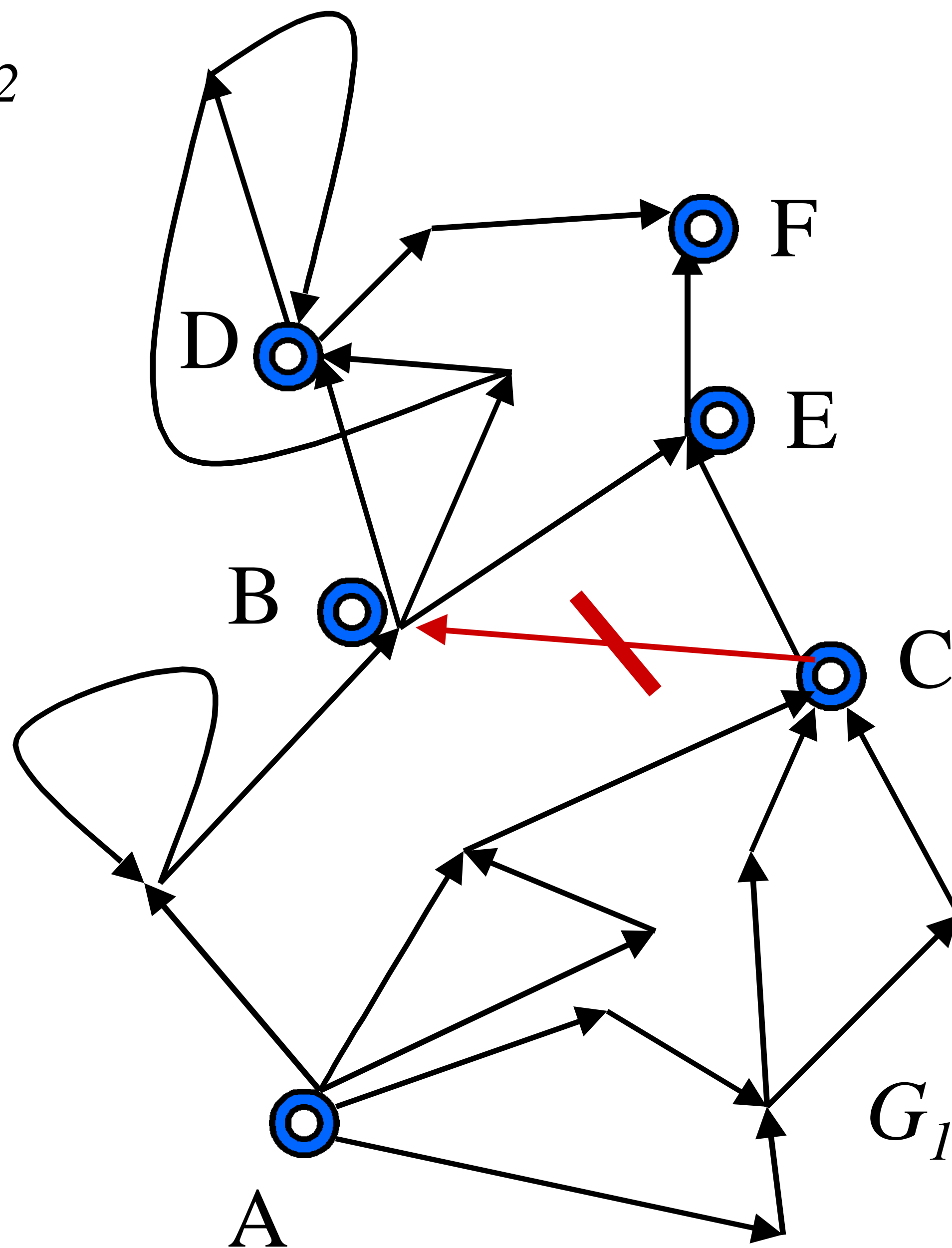
SMALL	BIG	INTENT
CG	CG	Evolution C
ICG	CG	view (abstr.) R
CG	ICG	view (roles) R
ICG	ICG	Layering R
PSG	CG	AP C or R
PSG	ICG	Improv. AP R

Refinement

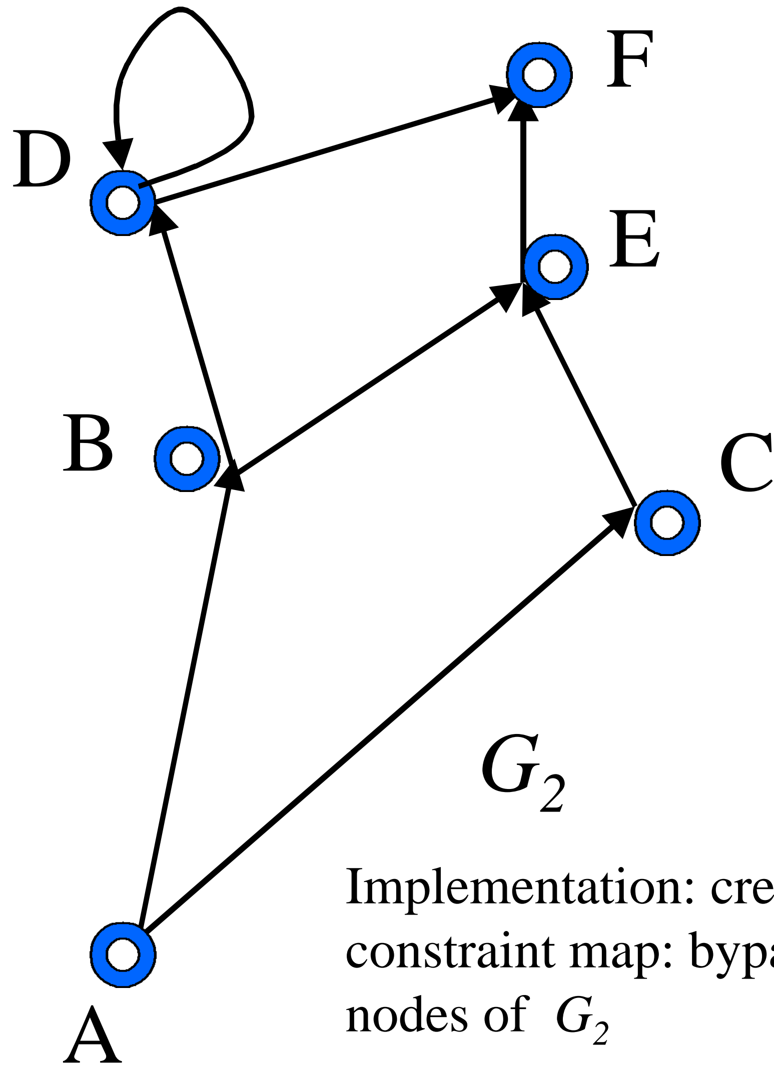
- For each edge in G2 there must be a corresponding pure path in G1.
- Pure path = in interior no nodes of G2.
- Refinement = strong refinement with “if and only if” replaced by “implies”.

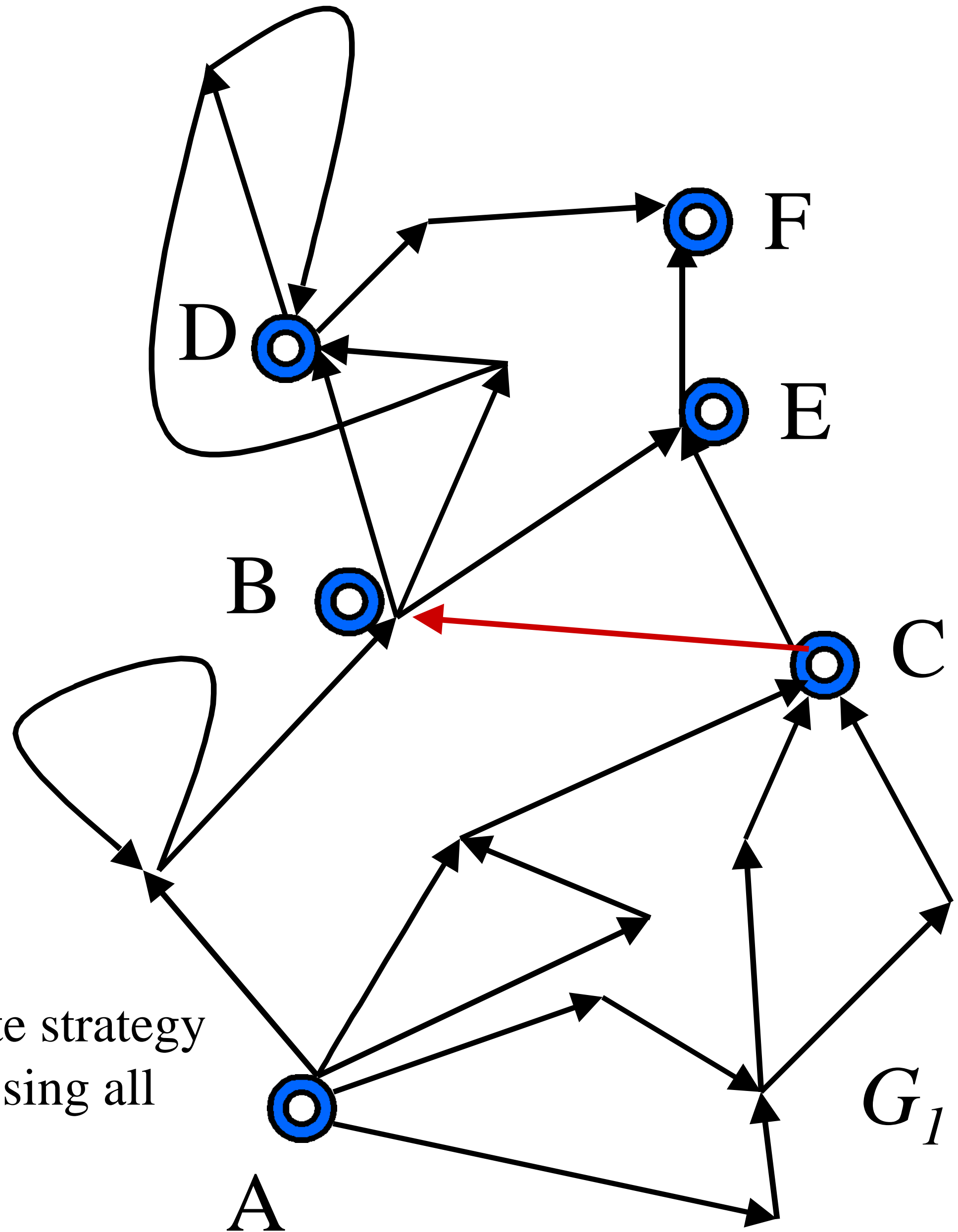
G_1 strong refinement G_2





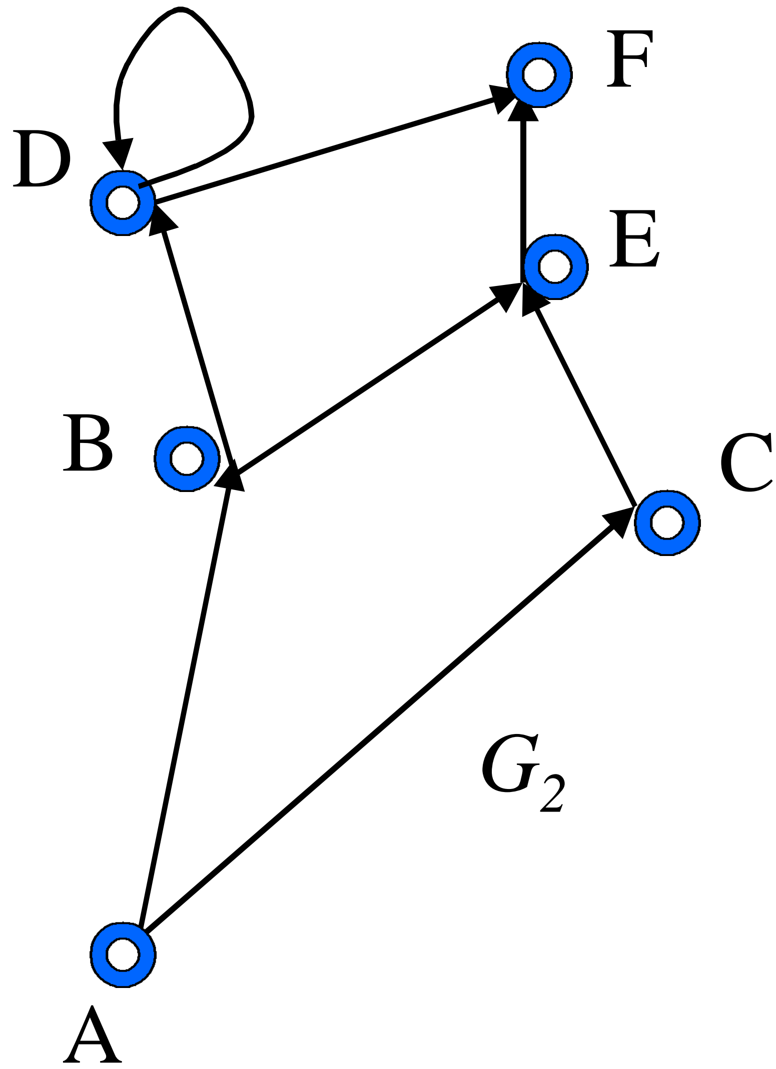
G_1 refinement G_2

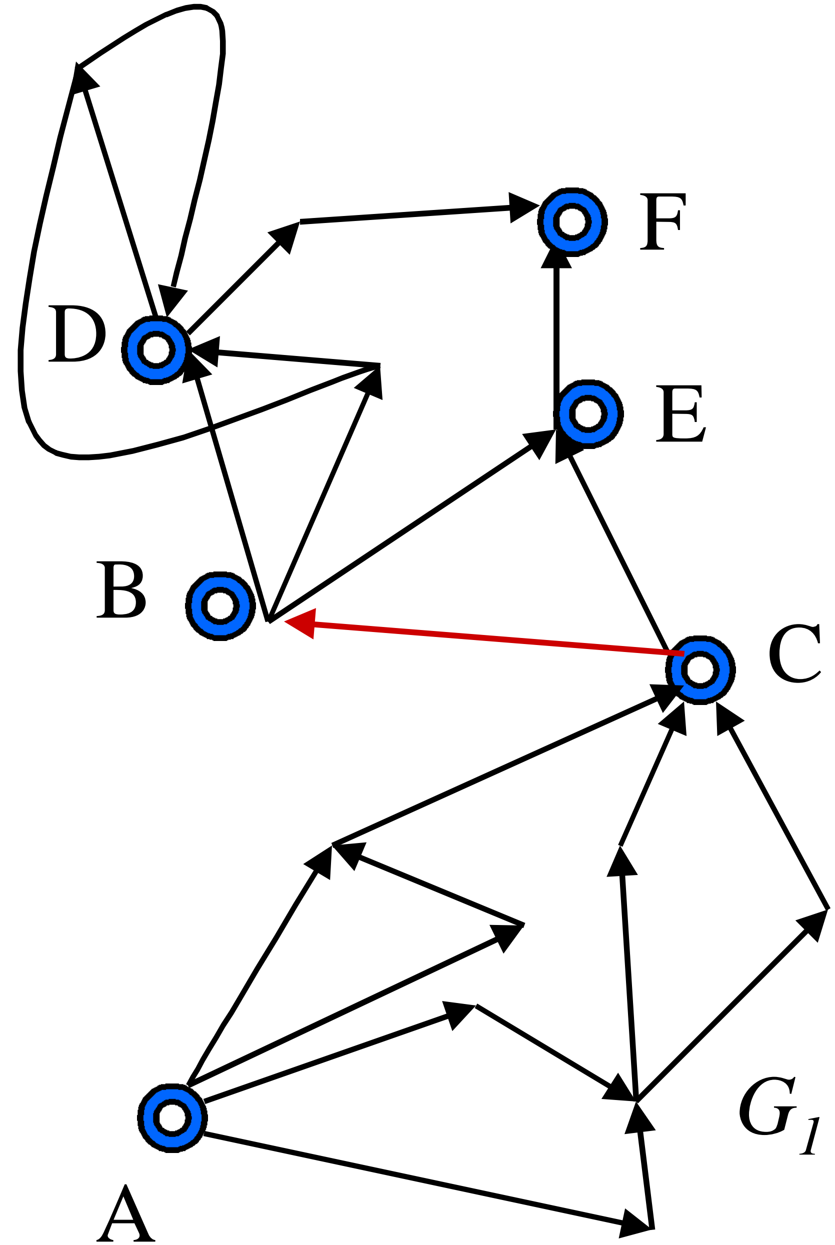




Implementation: create strategy constraint map: bypassing all nodes of G_2

G_1 compatible G_2



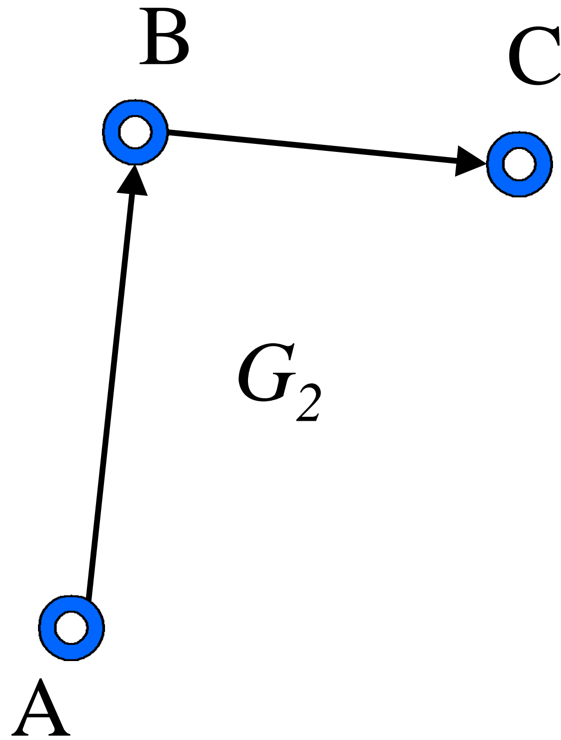


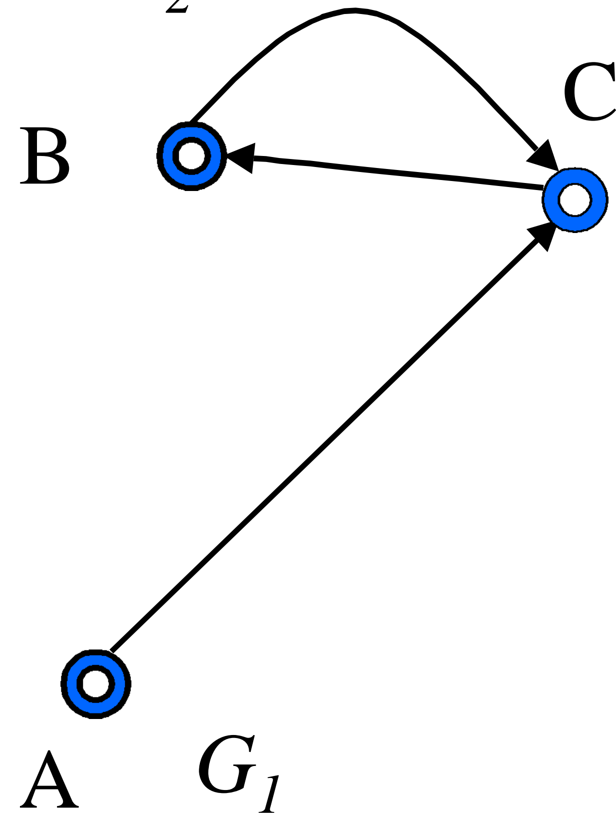
Refinement means: no surprises

not G_1 strong refinement G_2

not G_1 refinement G_2

G_1 compatible G_2

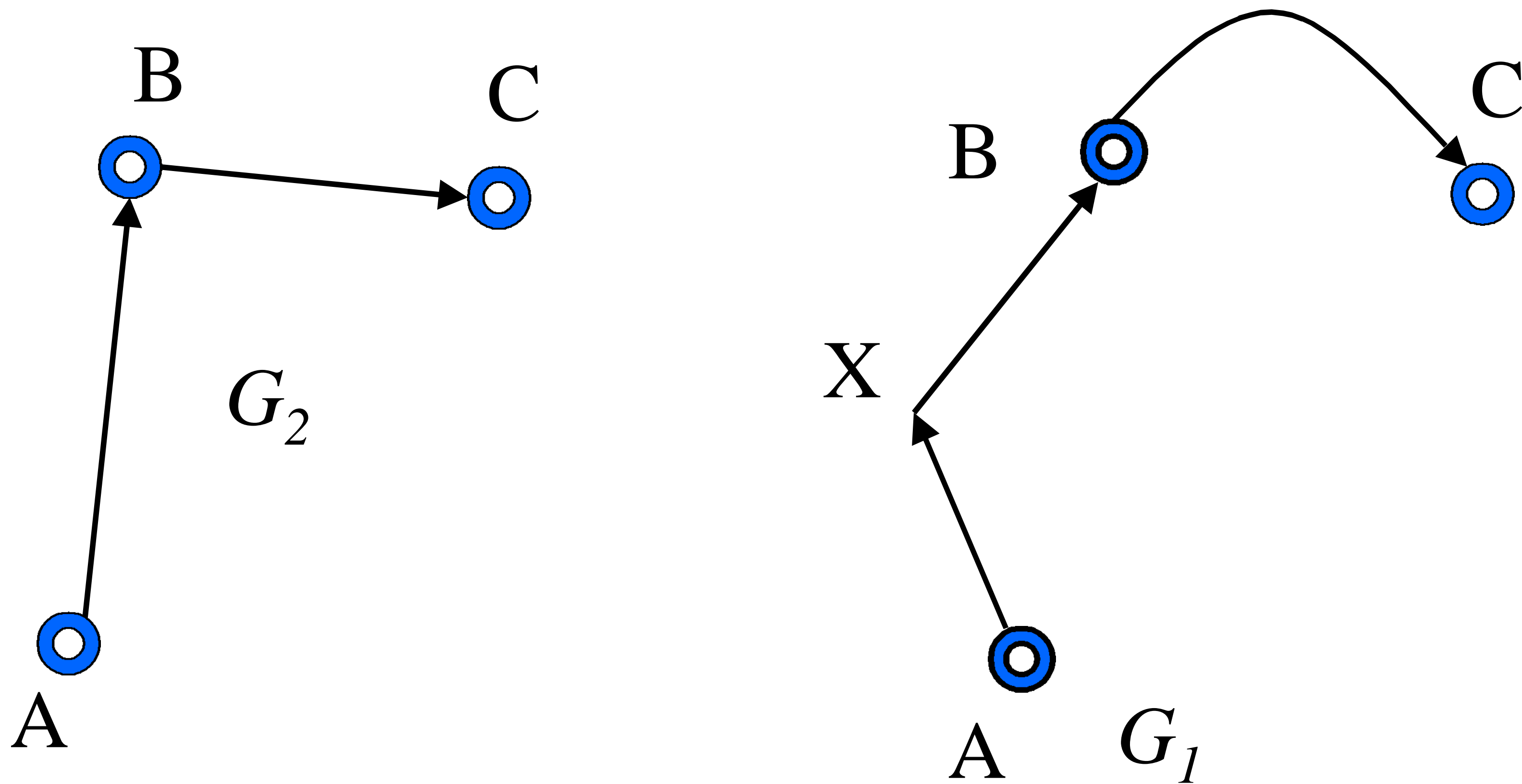




Refinement means: no surprises

G_1 strong refinement G_2

G_1 refinement G_2

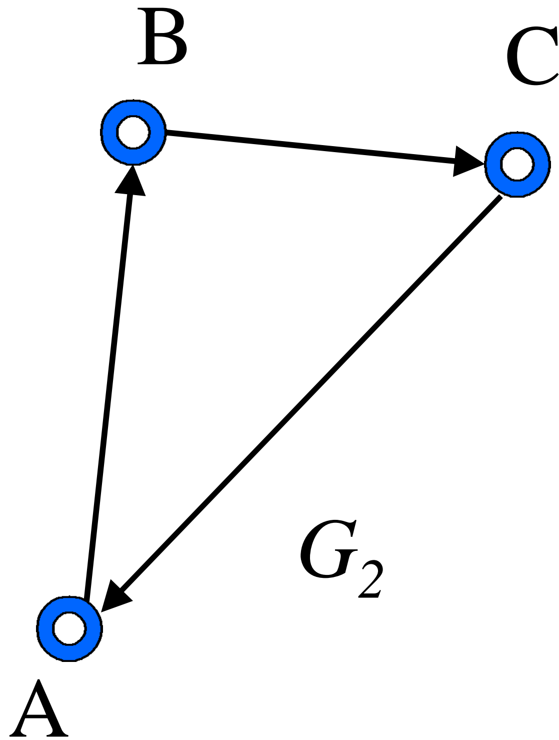


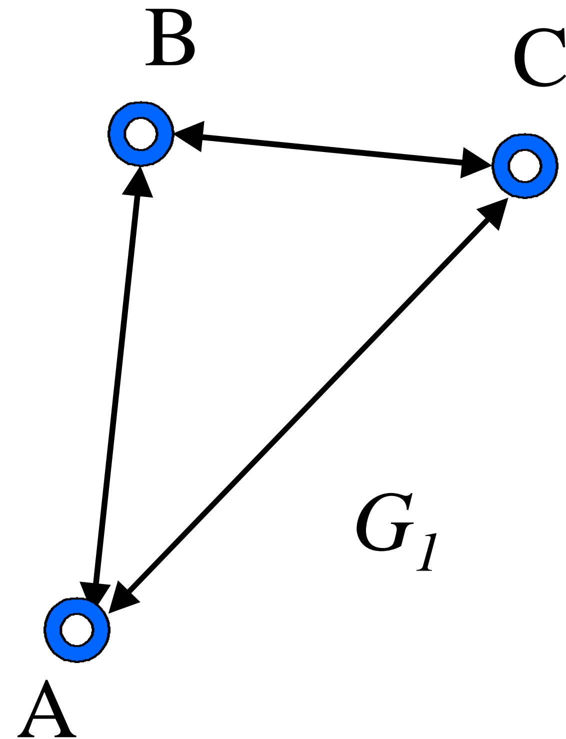
Refinement means: no surprises

G_1 compatible G_2

not G_1 strong refinement G_2

G_1 refinement G_2





Implementation of refinement: reduce to compatability

- Translate G_2 into a strategy graph S that has "bypassing all nodes" as constraint on each edge.
- Check whether S is compatible with G_1 , i.e. there is a path in G_1 satisfying the constraint for each edge in S.
- Reuses Traversal Graph Algorithm.

Traversing pure paths only

- Use same strategy graph construction
- Compute traversal graph for that strategy graph
- Run-time traversals will only follow pure paths

Intersection

R-customizers(I1) intersect R-customizers(I2) is always non-empty if R = refinement. Choose union of two graphs.

Improved strategy definition: embedded, positive strategies

- Given a graph G, a strategy graph S of G is any connected subgraph of the transitive closure of G.
- The transitive closure of G=(V,E) is the graph G*=(V,E*), where E*={(v,w): there is a path from vertex v to vertex w in G}.

TC

S is a strategy for G

Discussion

- Seems strange: define a strategy for a graph but strategy is independent of graph.
- Many very different graphs can have the same strategy.
- Better: A graph G is a customizer of a graph S, if S is a connected subgraph of the transitive closure of G. (call G: concrete graph, S: abstract graph).

Discussion: important is concept of instance/abstraction

- A graph G is a customizer of a graph S, if S is a connected subgraph of the transitive closure of G. (call G: concrete graph, S: abstract graph).
- A graph S is an abstraction of graph G iff G is a customizer of S.

Improved definition

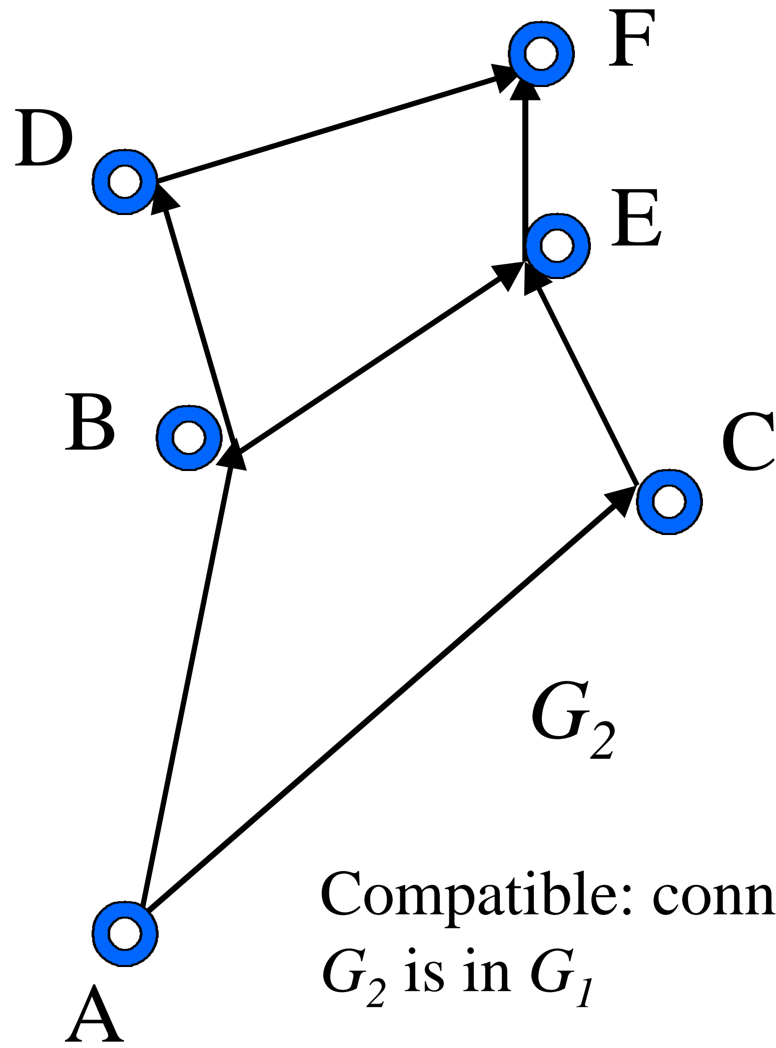
- Graph G is compatible with graph S by definition. What if we want G to be a refinement of S?
- A graph G is a refinement-customizer of a graph S, if S is a connected subgraph of the pure transitive closure of G with respect to the node set of S.

Pure transitive closure

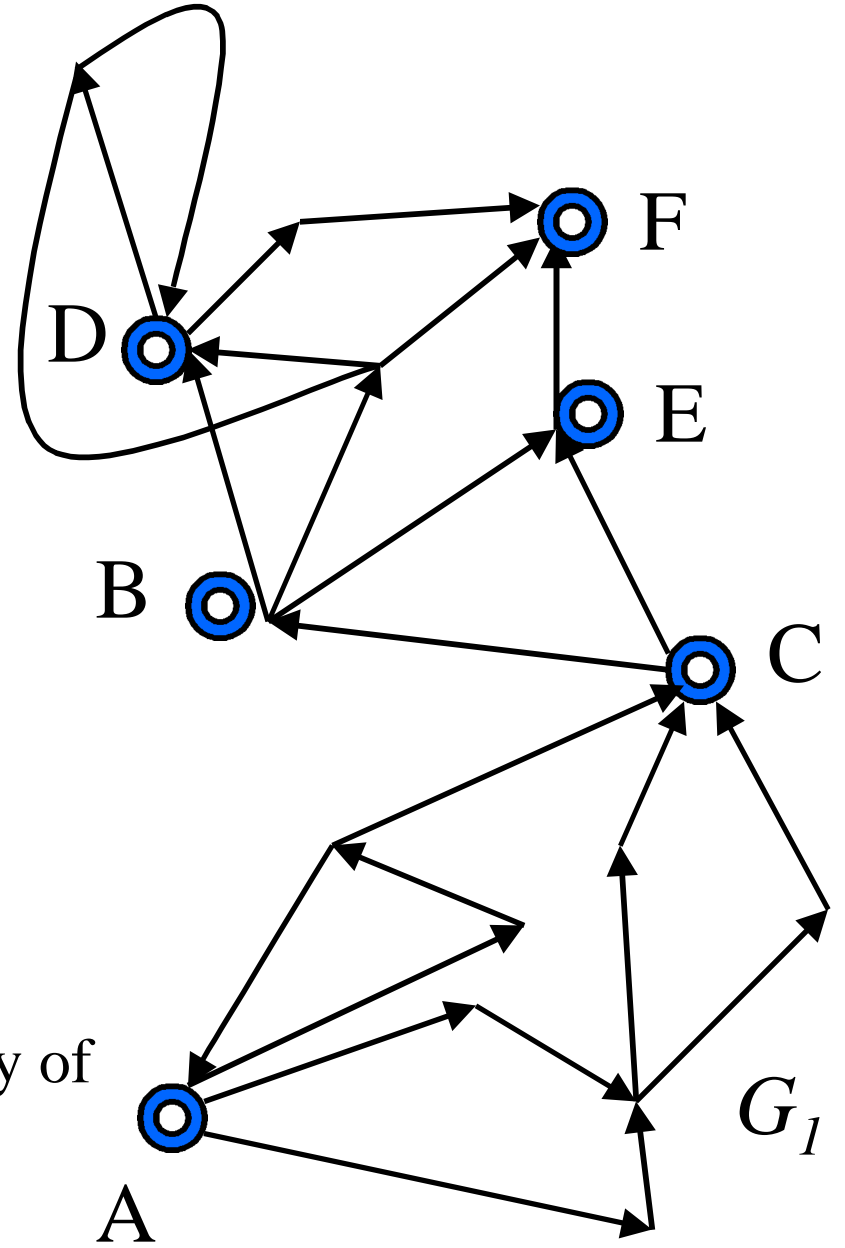
- The pure transitive closure of G=(V,E) with respect to a subset W of V is the graph G*=(V,E*), where E*={(i,j): there is a W-pure path from vertex i to vertex j in G}.
- A W-pure path from i to j is a path where i and j are in W and none of the inner nodes of the path are in W.

TC

G_1 compatible G_2



Compatible: connectivity of G_2 is in G_1



Definition of graph instance

- A vertex-labeled graph I is an instance of a graph G if
  - the vertex labels of I are nodes of G
  - the labels of the successors of a node i in I with label g is a subset of the successors of node g in G

Similar instances

- S a graph, G a refinement-customizer of S
- IG: instance of G
- IG can be turned into an instance of IS by
  - deleting parts
  - contracting edges of instances of *G* without eliminating *S* nodes

Construct similar instance

- S a graph, G a refinement-customizer of S
- IG: instance of G. Construct IS(IG)
  - delete parts not covered by S
  - contract edges if one end-node is not in S
- Idea: The APPC/Java Beans operates on IS(IG) that is constructed in a lazy way.

aA(
 aB(
  aC(
   aA())))

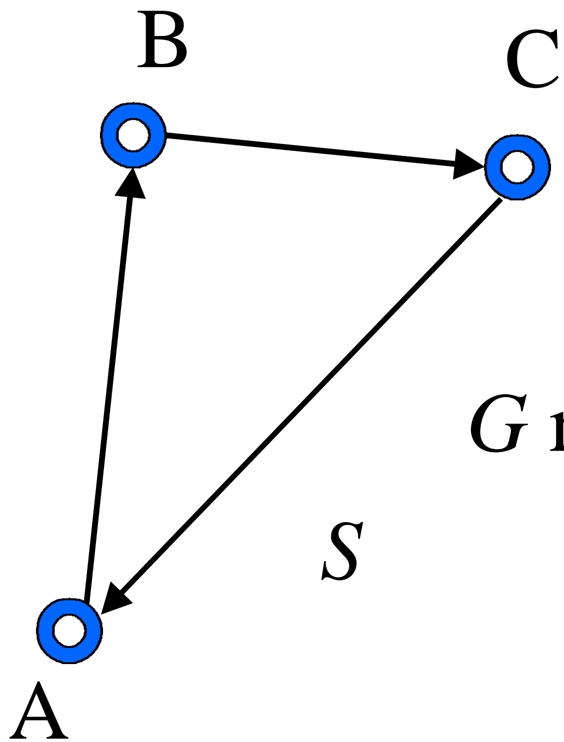
IS

aA(
 aB(
  aC(
   aX(
    aA())
   aB())
  aA())
 aX()))

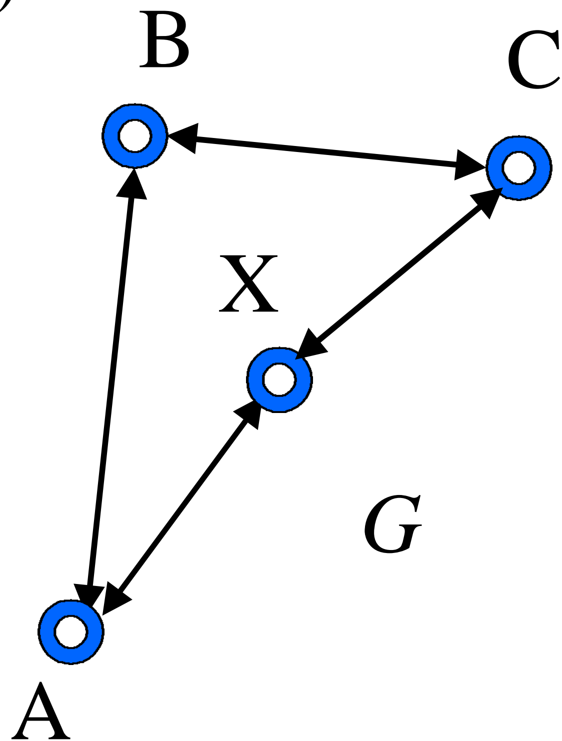
IG

APPC

contracting



G refinement *S*



Traversal Following a Graph

- S G IG
- S a graph, G a customizer of S, IG an instance of G.
- Traverse IG following S.
  - want it simpler than in TOPLAS paper
  - construct IS(IG)?
  - traverse IS following S: all of it.
  - map traversal to IG

Classifications of Traversal Histories

- S G I
- S a graph, G a customizer of S, I an instance of G. Traverse I following S.
- Edge traversal history:
  - edge-down (i1,g1) ->(i2,g2)
  - edge-up (i1,g1)->(i2,g2).
- Consider as language over edge-down and edge up-alphabet: Context-free? Regular?

Classifications of Traversal Histories

- S G I
- S a graph, G a customizer of S, I an instance of G. Traverse I following S.
- Node traversal history:
  - visit (i1,g1)
- Consider as language over node visit alphabet: Regular.

Comparison of Traversal Histories

- S G IG
- S a graph, G a refinement-customizer of S, IG an instance of G. IS(IG) instance of S. Traverse IG following S.
- Node traversal history of IS(IG) is a pure subtraversal of traversal of IG.

Note

- In practice: S ICG CCG I
- ICG a refinement-customizer of S
- CCG a refinement-customizer of ICG
- I an instance of CCG
- So far we simplified: S=ICG
- But same idea applies: traversal of ICG instance is a pure subtraversal of traversal of CCG instance

Note

- Terminology is not ideal
- R-customizers sounds like refinement customizer.
- R can be: refinement-customizer
- refinement-customizer-customizers: not elegant

What get's simpler?

- If we have refinement-customizer instead of customizer, what gets simpler besides the use of adaptive programming?
- Any of the algorithms?

Strategy Graph Minimization (SGM)

- Given G and a subgraph G' with vertex basis of size 1 for G' (node s) and reversed G' (node t), find the smallest subgraph S of the transitive closure of G so that Graph($PathSet_{st}$(G,S))=G'
- What if we replace "transitive closure" by "pure transitive closure with respect to node set of S".

Strategy Graph Minimization (SGM)

- SGM may have no solution.
- What is the complexity of deciding whether there is a solution?

Relaxed Strategy Graph Minimization (RSGM)

- Given G and a subgraph G' with vertex basis of size 1 for G' (node s) and reversed G' (node t), find a subgraph S of the transitive closure of G so that G' is a subgraph of Graph($PathSet_{st}$(G,S))=Q and size of S plus Q is minimal.
- What if we replace "transitive closure" by "pure transitive closure with respect to node set of S".

Definitions

- Given G and a subgraph S of the transitive closure of G, $PathSet_{i,j}(G,S)$ is the set of all paths in G from i to j that are expansions of paths in S from i to j.
- Graph(PathSet(G,S))=smallest subgraph of G that contains all paths in path set.

Class graph synthesis

- Given two graphs S1 and S2, find a smallest graph G such that both S1 and S2 are subgraphs of the transitive closure of G.
- Can take G to be a cycle.

Class graph synthesis

- Given two graphs S1 and S2, find a smallest graph G such that S1(S2) is a subgraph of the pure transitive closure of G with respect to S1(S2), respectively.

Hierarchical Graphs

- In the following use the definition of hierarchical graphs in the Yannakakis FSE '98 paper.
- That paper shows how to solve reachability problems efficiently for hierarchical graphs.

Hierarchical Graphs

- Traverse according to hierarchical graph
- Is G (hierarchical) a customizer of S?
- Is G (hierarchical) a refinement-customizer of S?
- SGM for hierarchical graphs
- Pure SGM for hierarchical graphs
- RSGM for hierarchical graphs

Counting argument?

- Given a graph G, how likely is it that a connected subgraph G' of G can be described by a subgraph S of the transitive closure of G so that S is smaller than G'?
- Depends on naming of nodes. Allow regular expressions on nodes in S.
- Study classes of graphs: trees: high likelihood.

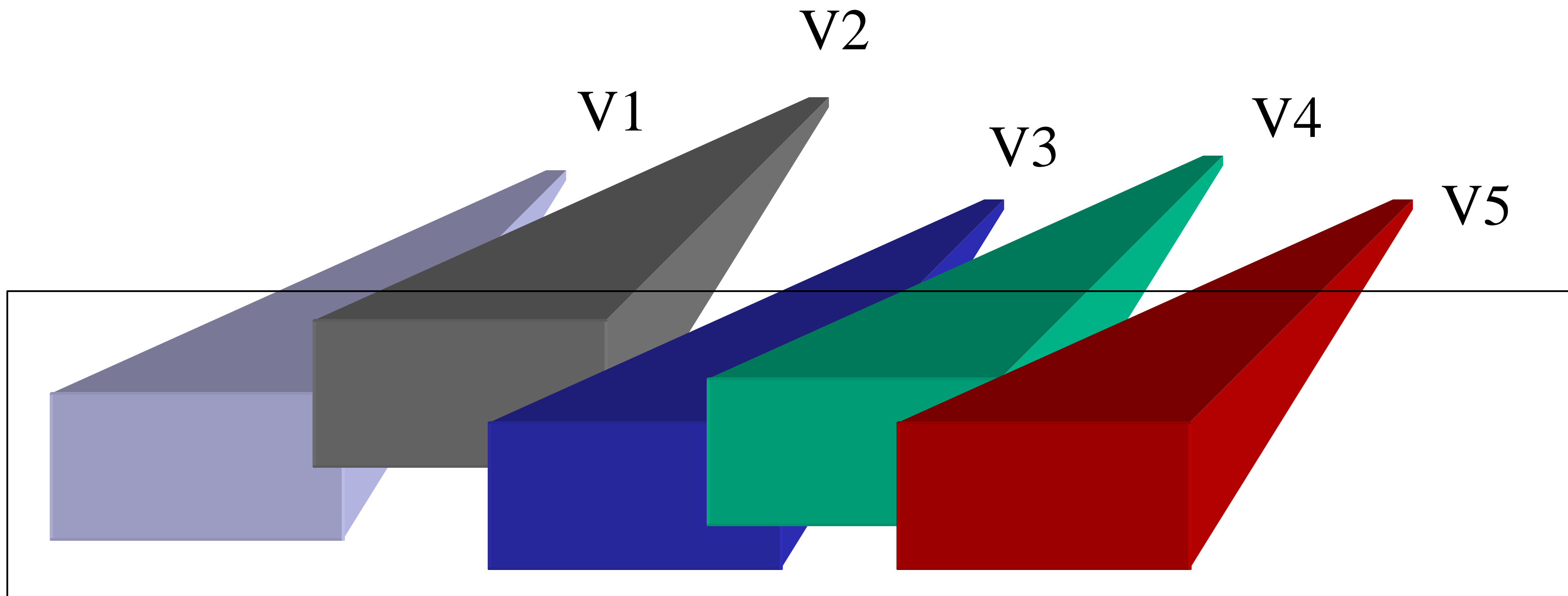
Co-existence of multiple organizations in software systems

- OOP: class graph
- Many issues cross-cut the class organization: subgraph of class graph plus additional decorations on class graph
- problem: subgraph definitions are brittle with respect to class graph: change of class graphs requires changes to many subgraphs

- Lift subgraphs: define them abstractly and compose them.
- Subgraph definitions are graphs themselves: call them interface class graphs
- Set of interface class graphs: When can they be used together on a class graph. Want to automate the mapping.

View View of AP

Multiple views of the same class graph



Application class graph

In Demeter/Java: view = strategy graph

Connection to Frameworks

- A framework is a set of cooperating classes that make up a reusable behavior
- Typical use of a framework: by subclassing
- Leads to inversion of control: “We will call you; don’t call us”

Adaptive Programming and Frameworks

- Frameworks
  - a few big ones
  - hard to combine
  - hard to map
  - conventional technology
- APPCs
  - many small ones
  - easy to combine
  - easy to map
  - new