Correctness trumps efficiency

--- Source code and build system co-evolve ---

Bram Adams
GH-SEL, Ghent University
http://users.ugent.be/~badams

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

1. Build System?

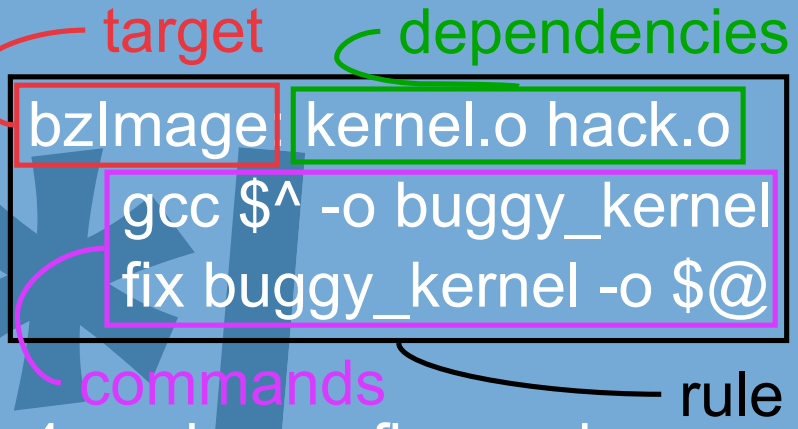
Linus T.:

1. gcc kernel.c -o bzImage
2. ./compile.sh

- tedious
+ automation
- incorrect builds
- long builds

BUILD SYSTEM

3. make (Feldman, 1977)



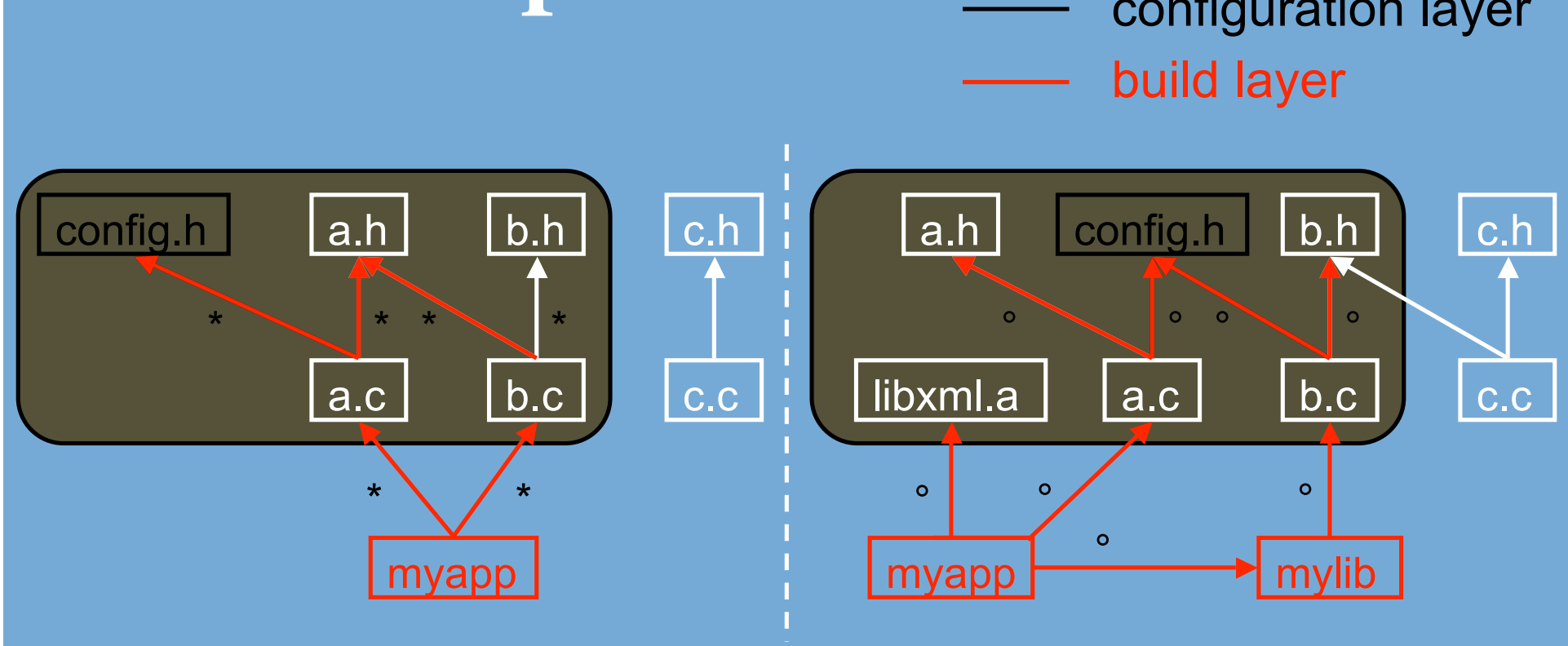
4. make config ; make

+ explicit dependencies
+ incremental
- source configuration
- platform dependencies
- co-evolution problem?

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

2. The full picture



Source code **changes**:

- refactoring
- code addition/removal
- reuse
- company reorganisation

co-evolution?

Build system **changes**:

- fix mistakes
- use other compilers, ...
- accelerate build
- finer-grained configuration

2. Claim: build system co-evolves with source code

Indications:

- out-of-sync ⇒ source code worthless or inconsistent build
- rigid build system ⇒ less freedom to restructure/refactor code

Implications:

- re-engineering issues
  - hidden cost
  - difficulty of integrating e.g. AOP
- build as mirror of source code
  - high-level structural knowledge of source code architecture
  - concern interaction detection

Case study:

- investigate Linux kernel build evolution ↔ how to "measure" build?

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

(S. I. Feldman. "Make-a program for maintaining computer programs". Software - Practice and Experience, 1979)

3. MAKAO (a)

Makefile

variable

```
make_OBJECTS = ar.o arscan.o \
        commands.o dir.o ... hash.o
```

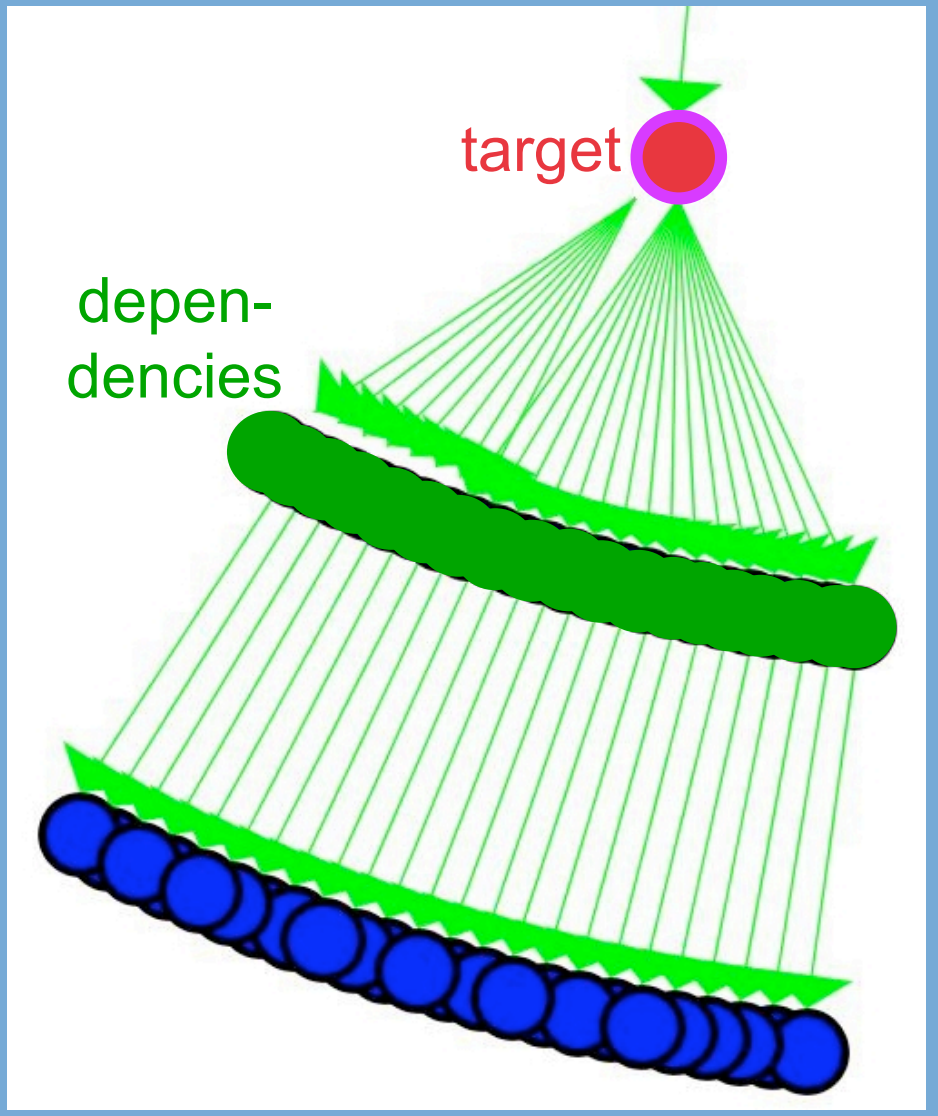
target dependencies

```
make$(EXEEXT): $(make_OBJECTS)
        @rm -f make$(EXEEXT)
        $(LINK) $(make_LDFLAGS) \
        $(make_OBJECTS) \
        $(make_LDADD) $(LIBS)
...
```

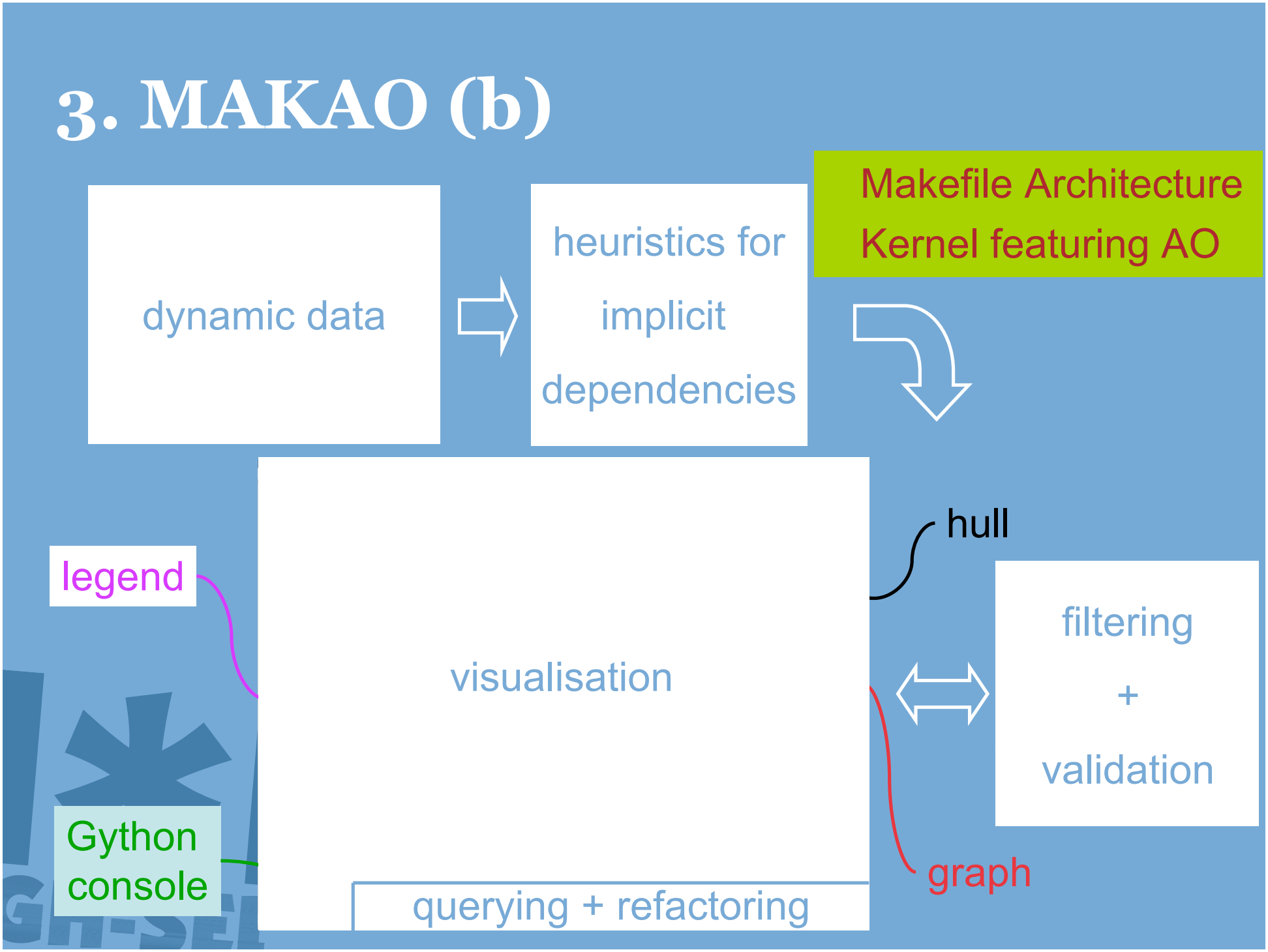
commands rule

⟹ de facto build tool/process model!

Directed Acyclic Graph (DAG)



3. MAKAO (b)


dynamic data
heuristics for implicit dependencies
Makefile Architecture
Kernel featuring AO
legend
visualisation
hull
filtering + validation
Gython console
querying + refactoring
graph

3) Implicit dependencies

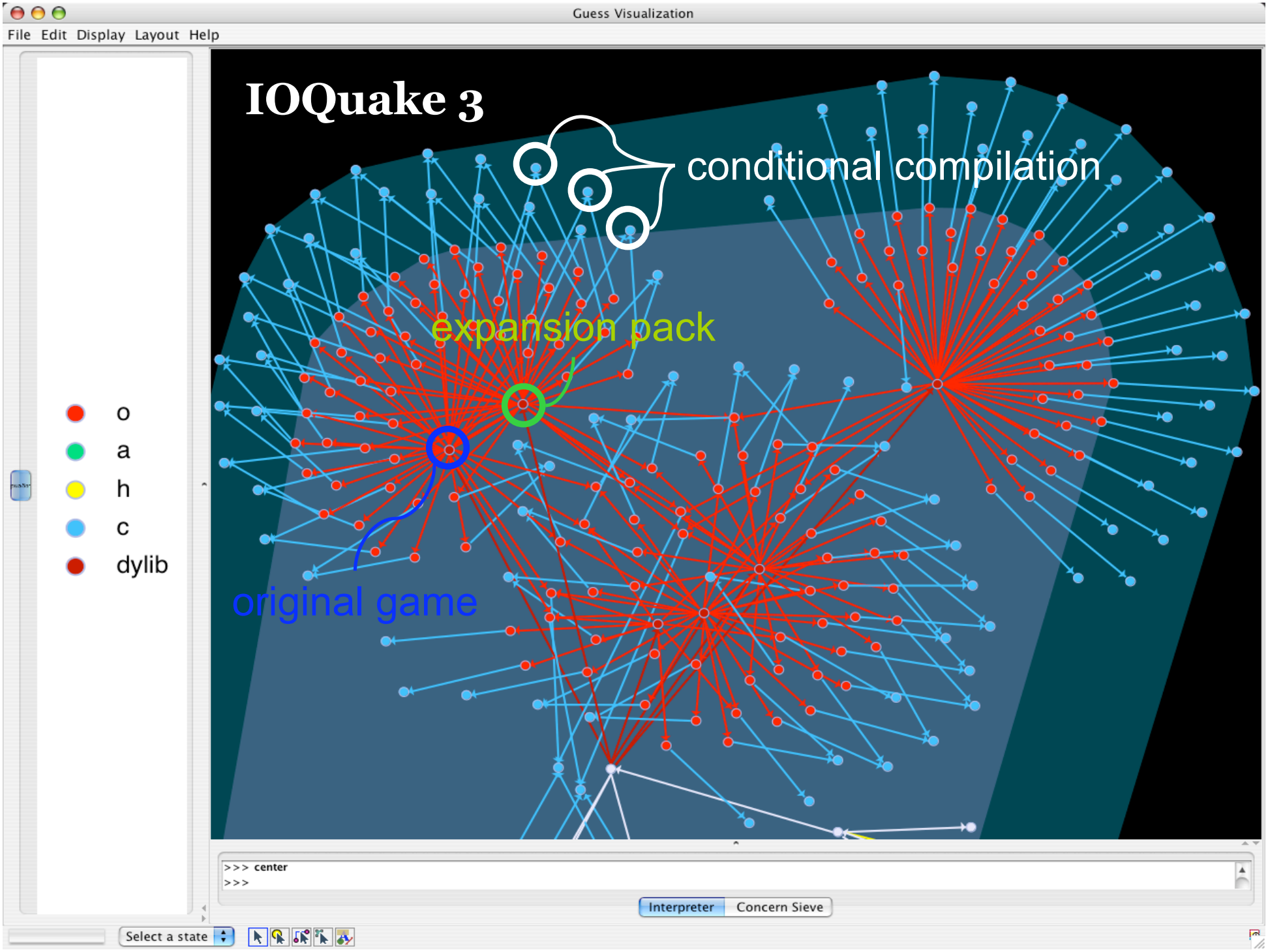
```
all: A.class app.jar

A.class: A.java
	javac $<

app.jar:
	jar cf $@ A.class `ls ../classes/*.class`
```

danger:

- build errors
- problems with incremental compilation
- idem for parallel builds


Guess Visualization
File Edit Display Layout Help
IOQuake 3
conditional compilation
expansion pack
original game
o
a
h
c
dylib
>>> center
>>>
Interpreter
Concern Sieve
Select a state

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

4. Case study

Linux kernel:

- source code grows superlinear [Godfrey et al.]
- what about build system?

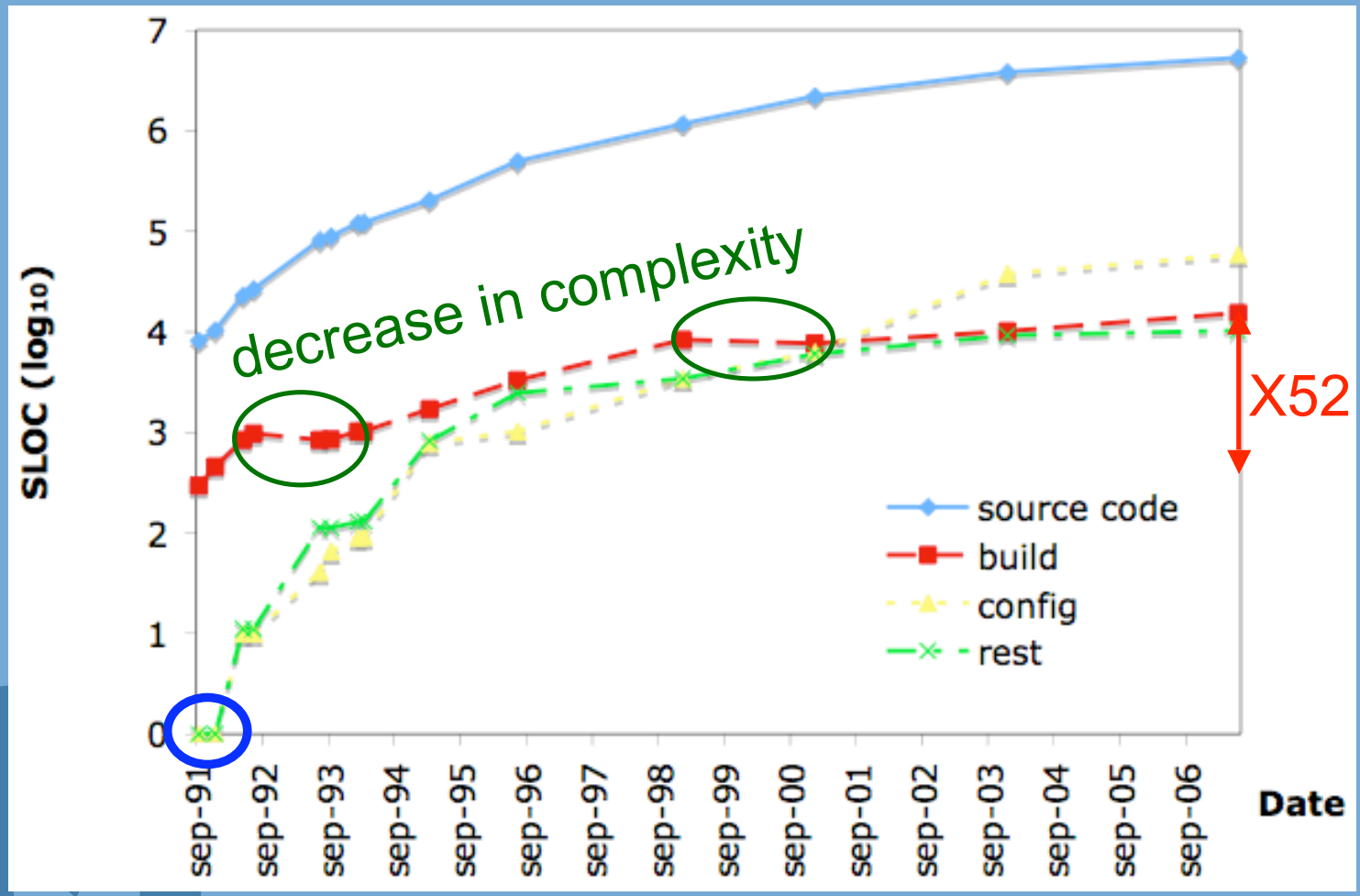
Case study:

- SLOC
- (full) build dependency graph characteristics + correlation with development history
- zoom in on evolution step (2.6 kernel)

How?

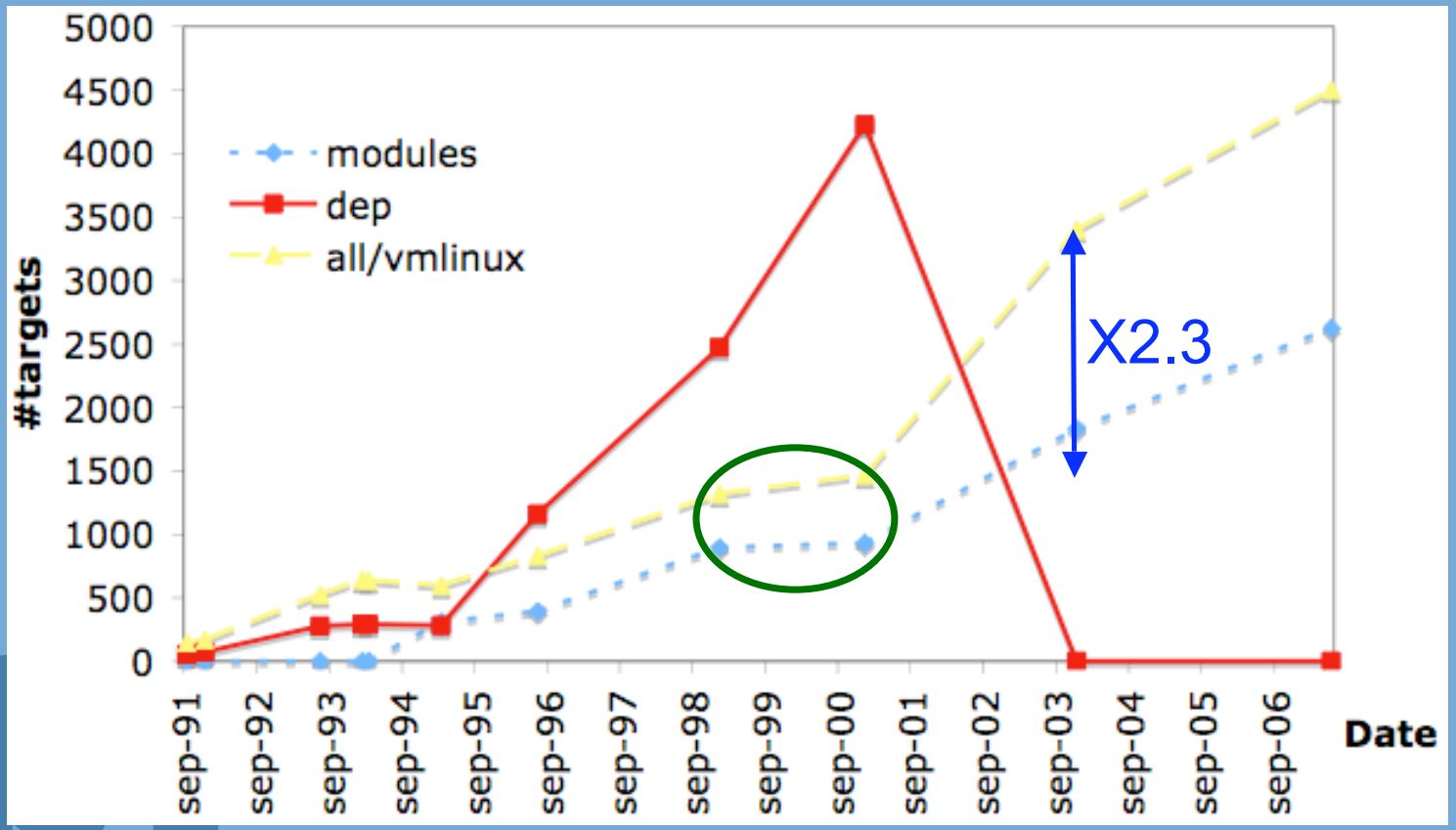
- SLOCCount (http://www.dwheeler.com/sloccount/)
- MAKAO (Makefile Architecture Kernel featuring AO)

4. SLOC



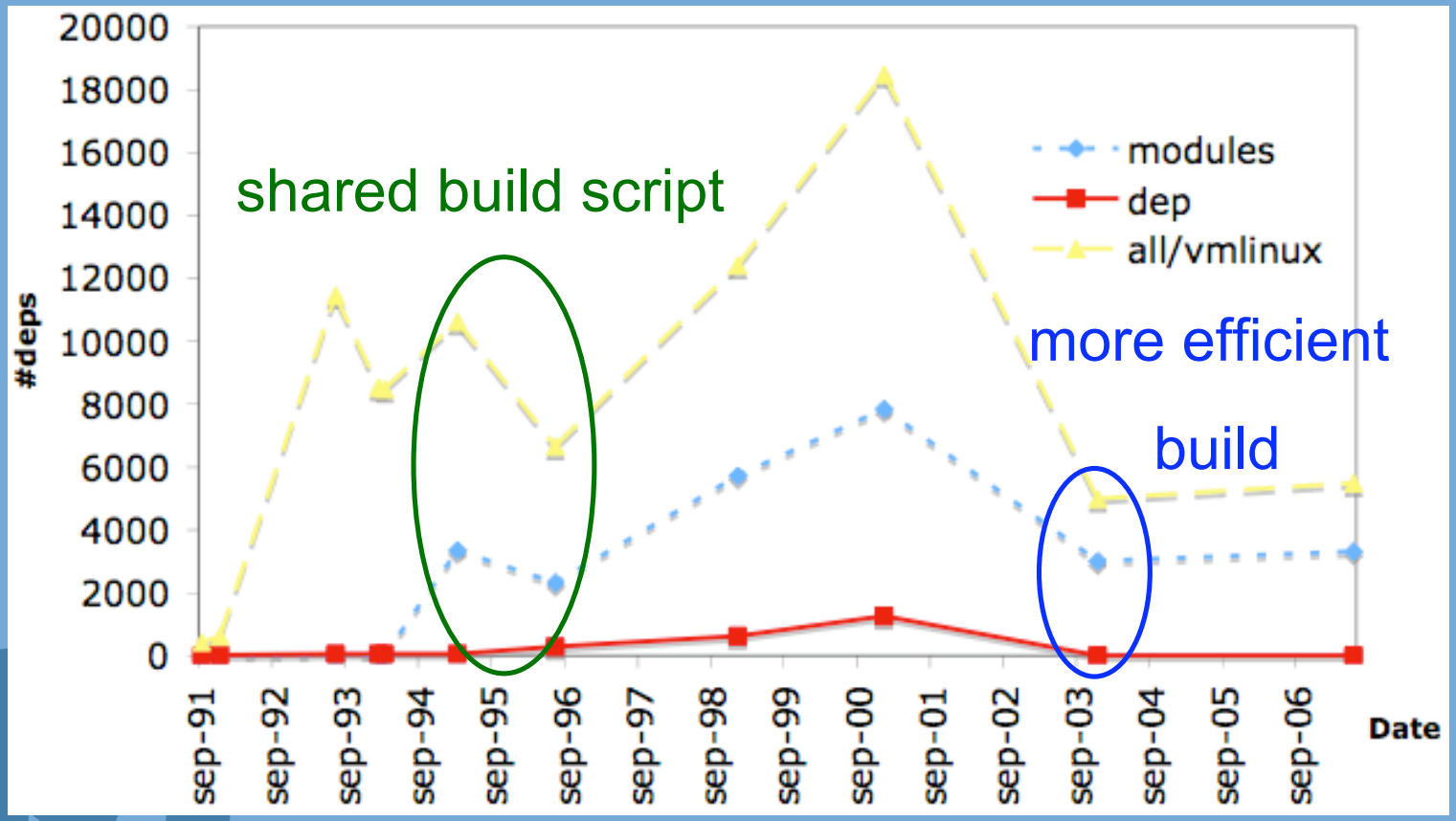
build system evolves with the source code

4. Build target count



build system responsibilities evolve with the source code

4. Explicit dependency count



build system fluctuates (reflects source code refactoring?)

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

5. The 2.6 kernel

Why important?

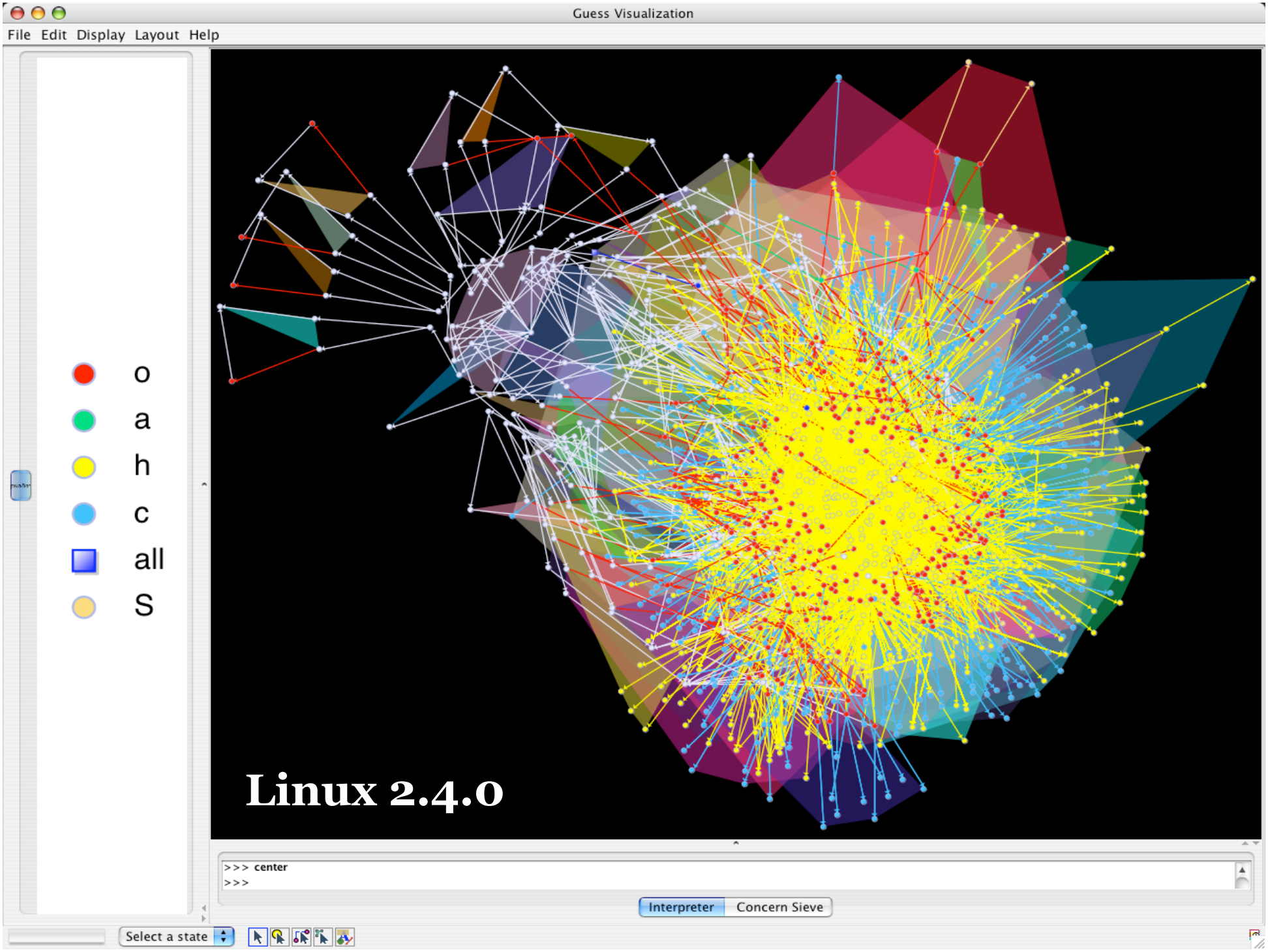
- big changes in dependency graph measurements
  - end of "dep" phase
  - huge increase in number of targets
  - huge decrease in number of explicit dependencies
- lots of mailing list activity
  - CML 2 (Eric S. Raymond)
  - Kbuild 2.5 (Keith Owens)
- political decisions governing build system evolution

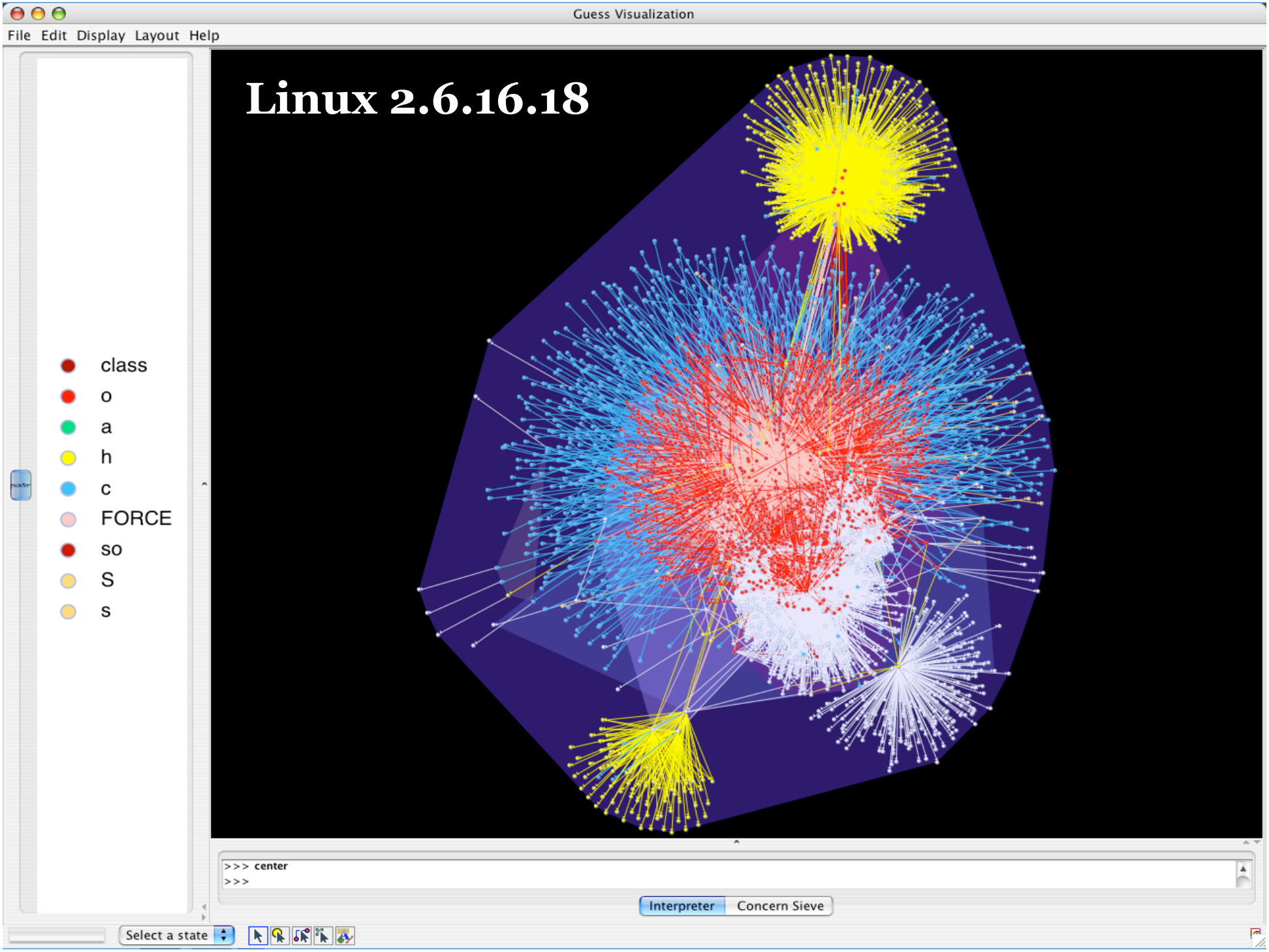
What are we going to do?

- Unravel 2.6 build dependency graph
- Try to detect what has changed
- Use logic rules to abstract away idioms

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion


Guess Visualization
File Edit Display Layout Help
o
a
h
c
all
S
Linux 2.4.0
>>> center
>>>
Interpreter
Concern Sieve
Select a state


Guess Visualization
File Edit Display Layout Help
Linux 2.6.16.18
class
o
a
h
c
FORCE
so
S
s
>>> center
>>>
Interpreter
Concern Sieve
Select a state

5a. Dependency management

```
depend dep: dep-files

dep-files: scripts/mkdep archdep include/linux/version.h
    scripts/mkdep init/*.c > .depend
    scripts/mkdep `find $(FINDHPATH) ... -name \*.h ... -print` > .hdepend
    $(MAKE) $(patsubst %,_sfdep_%,$(SUBDIRS)) _FASTDEP_ALL_SUB_DIRS="$(SUBDIRS)"
```

"dep" phase

```
ifneq ($(wildcard .depend),)
  include .depend
endif
```

"all"/"modules"

2.4.0

```
ifneq ($(cmd_files),)
  $(cmd_files): ; #Do not try to update
  include $(cmd_files)
endif
```

build iteration i-1

"vmlinux"/"modules"

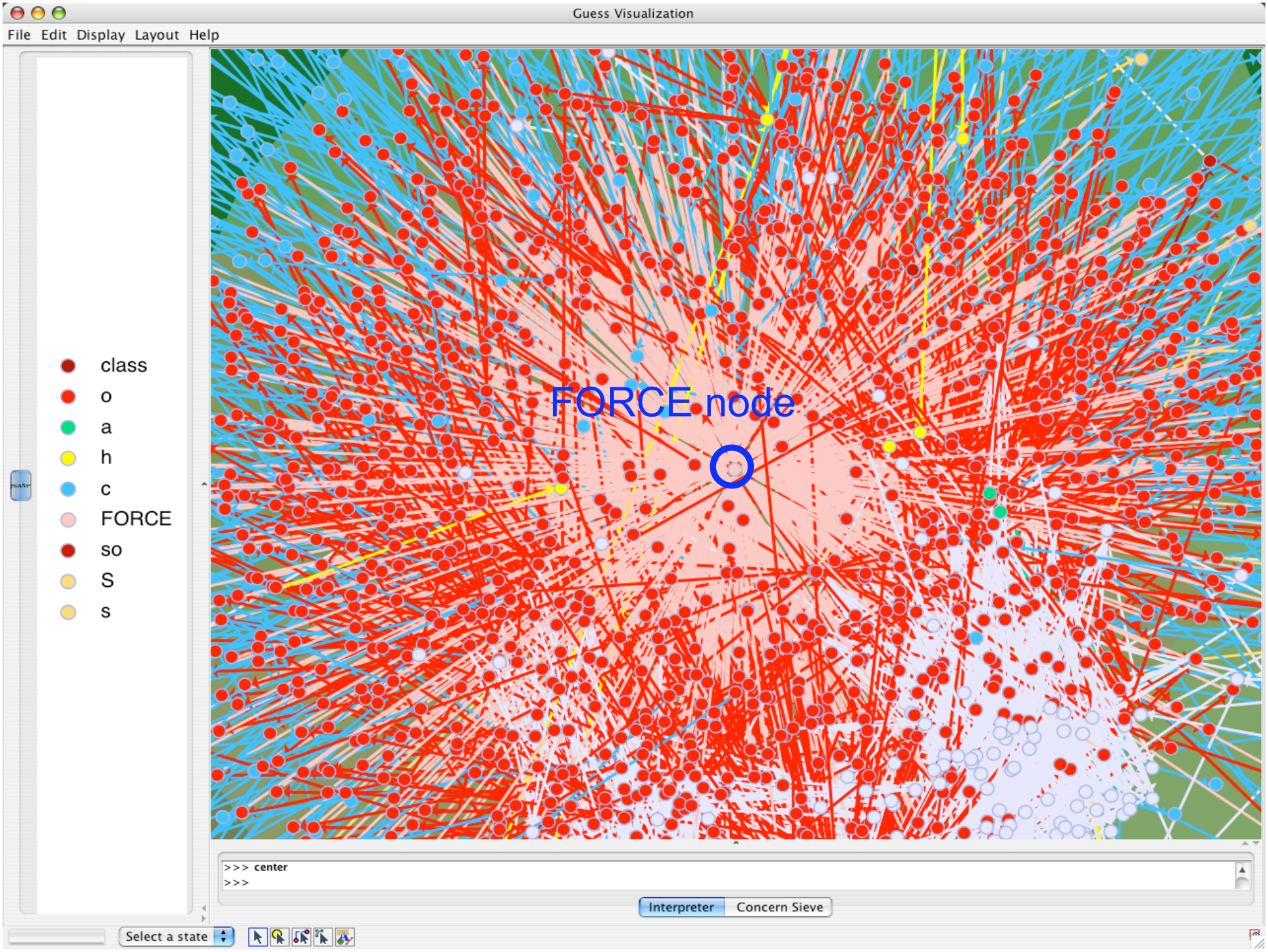
2.6.0

```
gcc -Wp,-MD,net/ethernet/eth.o.d ... -c -o file.o file.c
```

build iteration i

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion


Guess Visualization
File Edit Display Layout Help
class
o
a
h
c
FORCE
so
S
s
FORCE node
>>> center
>>>
Interpreter
Concern Sieve
Select a state

5b. Filtering with logic rules

MAKAO's logic rule component:

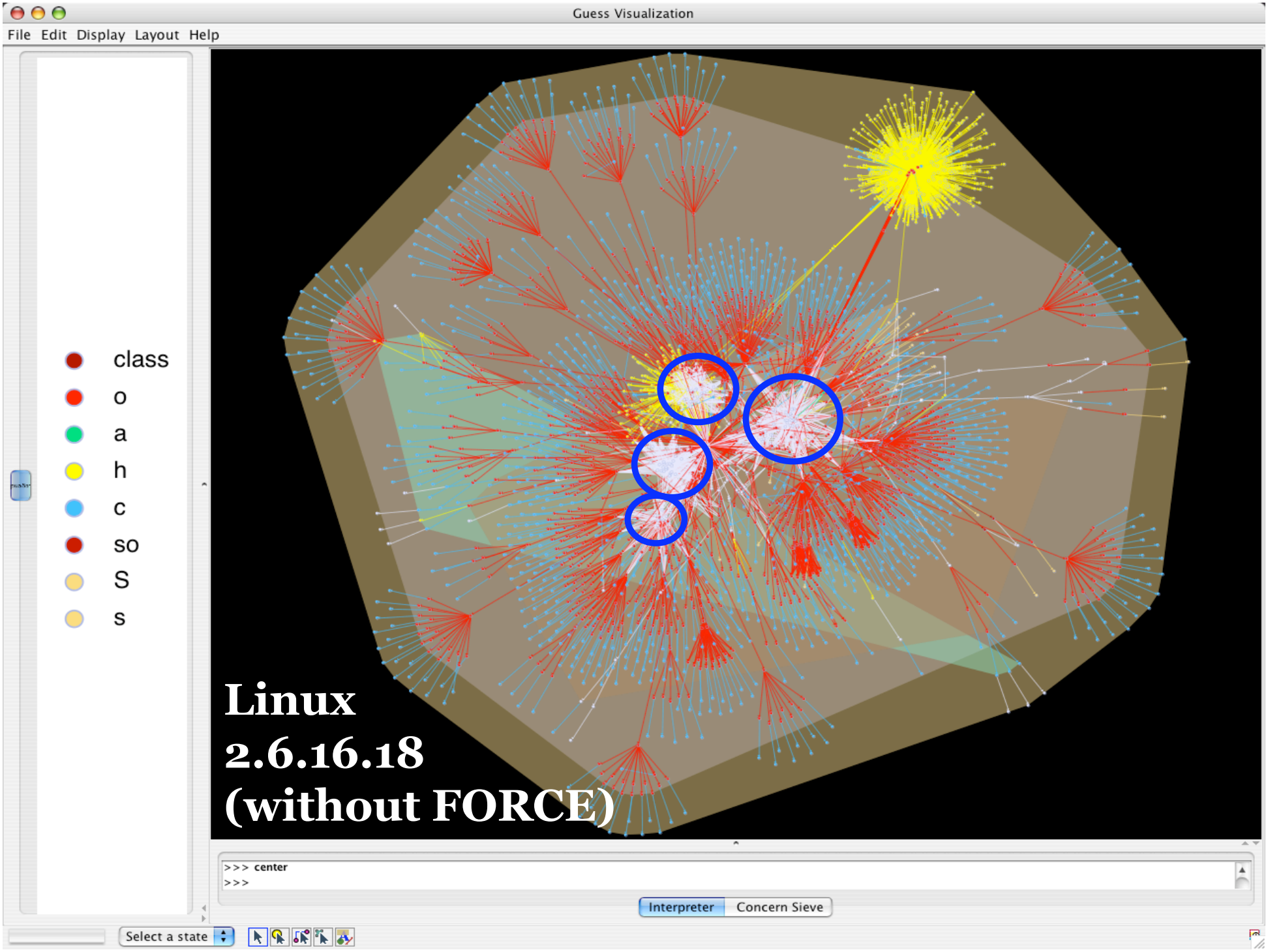
- flexibility to express application-specific build idioms
- abstract away from idioms
- declare high-level views

Structure of rules (old-style):

```
rule(name,RemoveList,AddList,OrphanList):-
   %> rule logic.
```

Example: Lose the FORCE, Luke!

```
rule(force,[edge(_,FORCE,_)],[],[FORCE]):-
   node(FORCE,'FORCE',_).
```

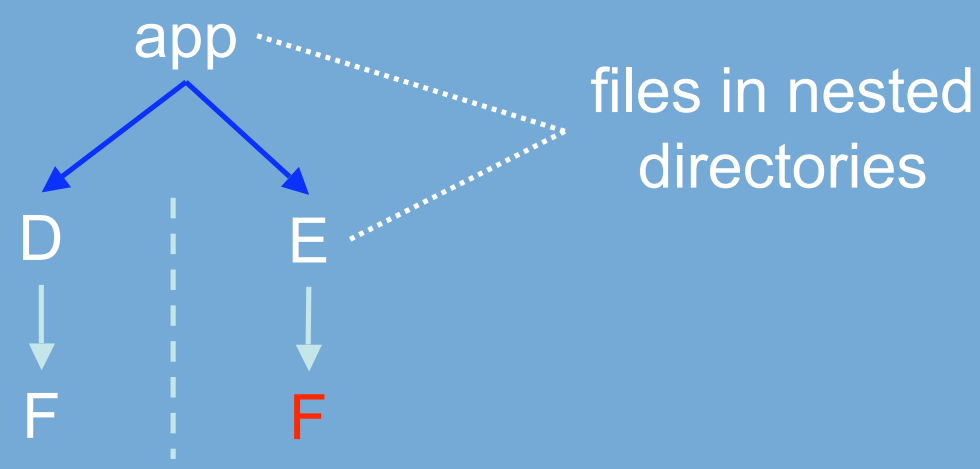

Guess Visualization
File Edit Display Layout Help
class
o
a
h
c
so
S
s
Linux
2.6.16.18
(without FORCE)
>>> center
>>>
Interpreter
Concern Sieve
Select a state

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

5c. Recursive make

Recursive Make:



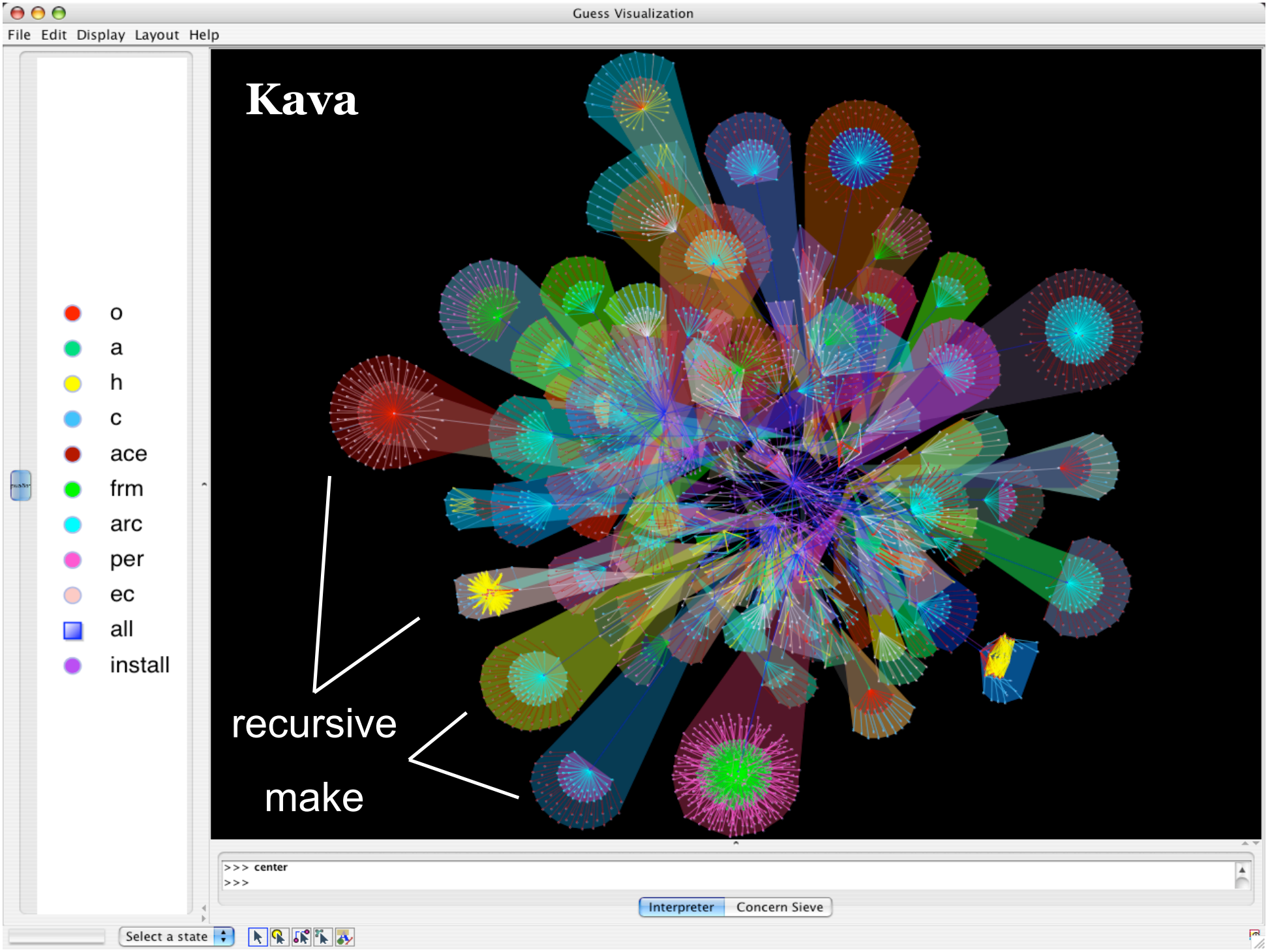
PRO:

- straightforward
- modular

CONTRA:

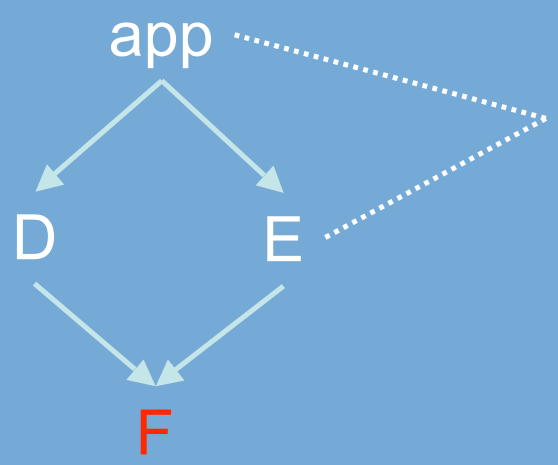
- missing dependencies
- costly subprocess creation
- parallel build capabilities reduced
- duplication of build/recursion logic

- fixed build order
- iterate
- ...


Guess Visualization
File Edit Display Layout Help
Kava
o
a
h
c
ace
frm
arc
per
ec
all
install
recursive
make
>>> center
>>>
Interpreter
Concern Sieve
Select a state

5c. Non-Recursive make

Non-Recursive Make:



files in nested directories

PRO:

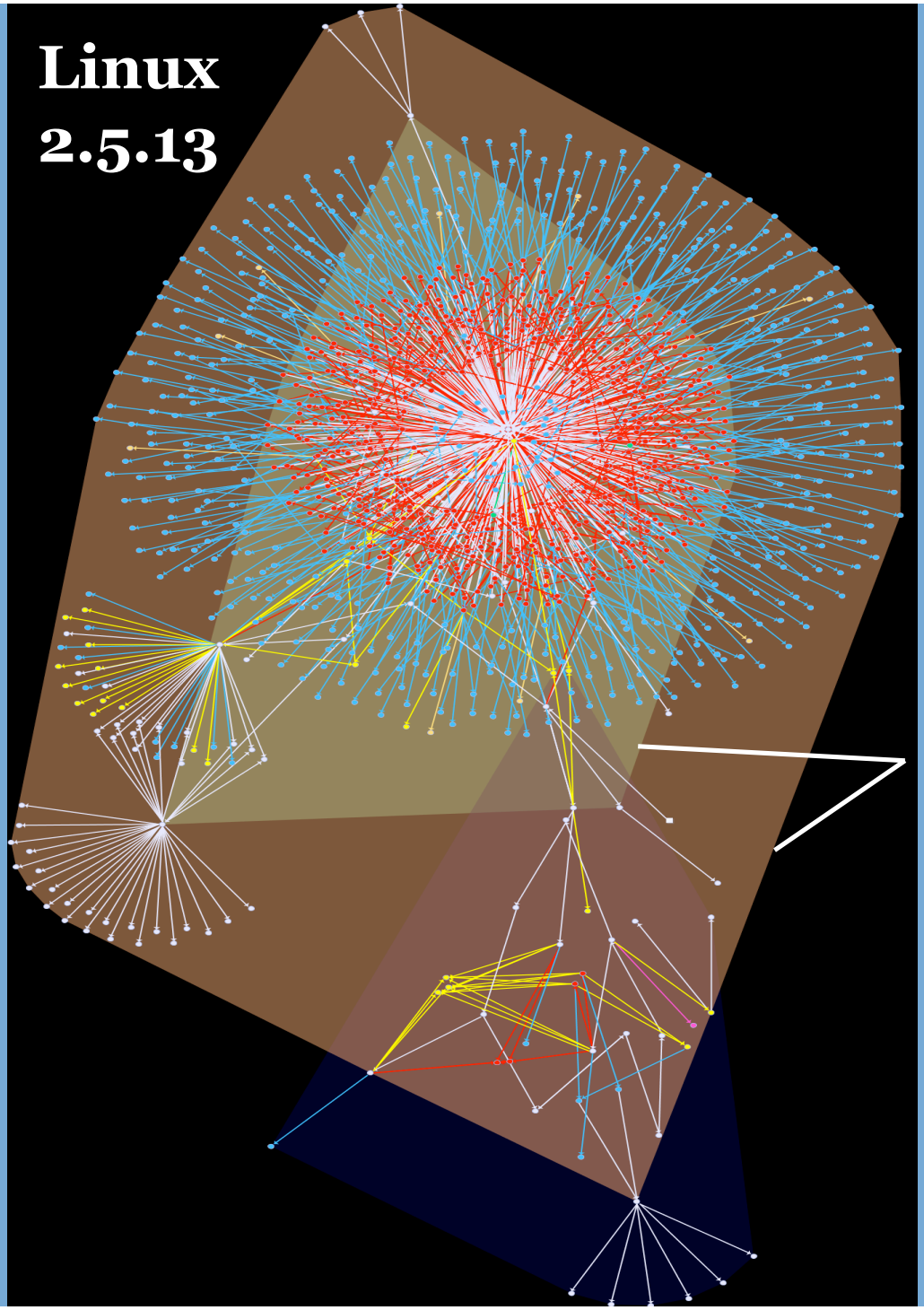
- whole-system view
- optimally parallelisable
- no artificial recursion logic

- include
- one big makefile
- ...

CONTRA:

- name clashes, current directory, subdirectory builds, ...
- construction of dependency graph

Kbuild 2.5



non-recursive

make

GH-SEL

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

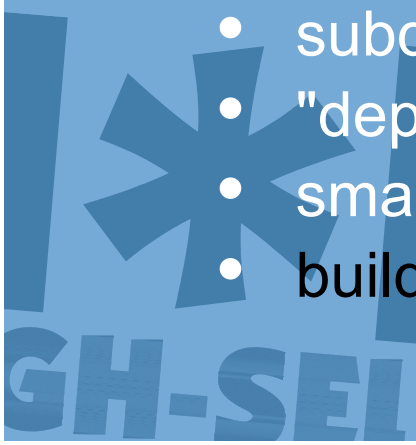
5d. Compromise

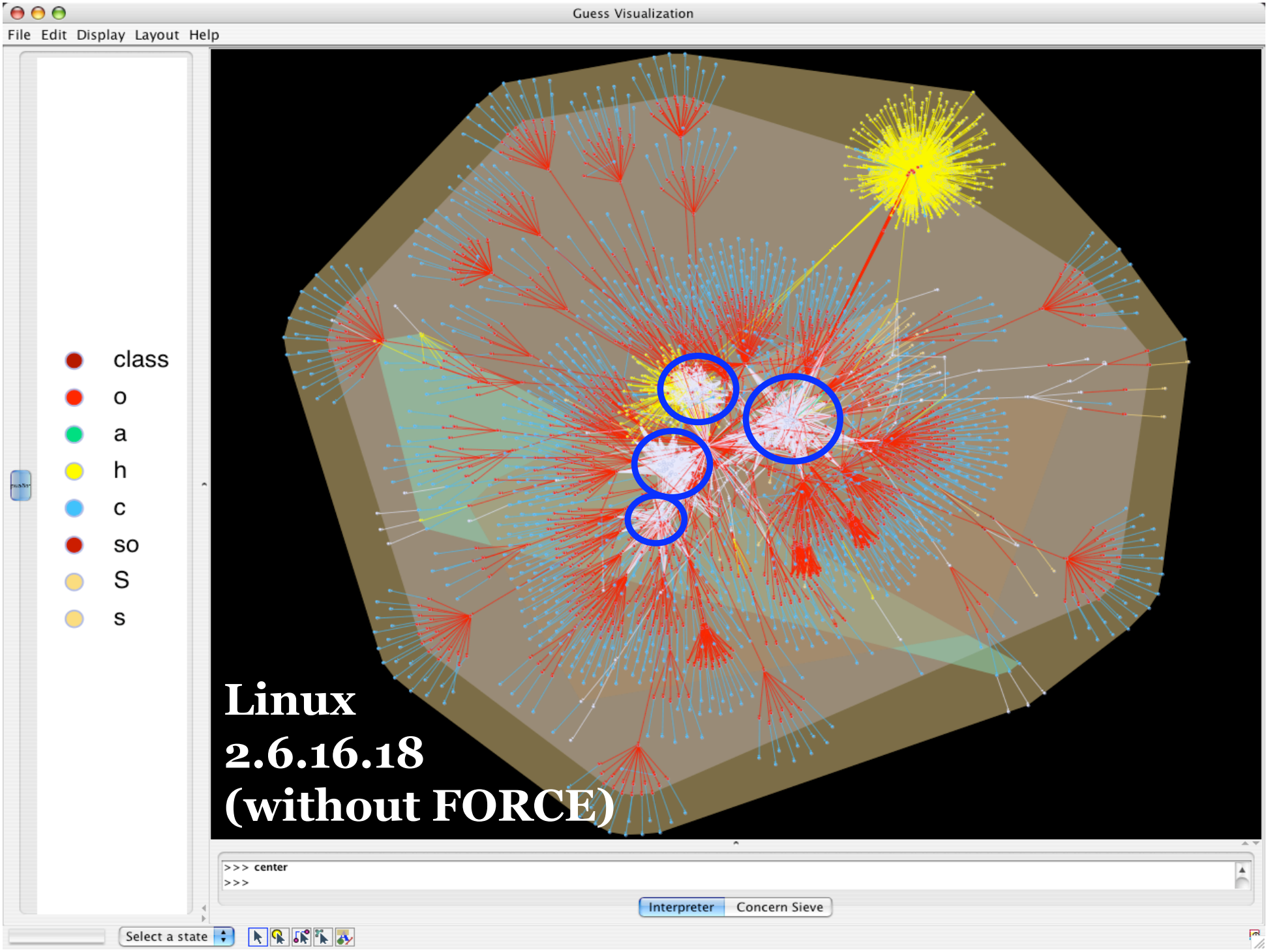
Kbuild 2.5 (Keith Owens):

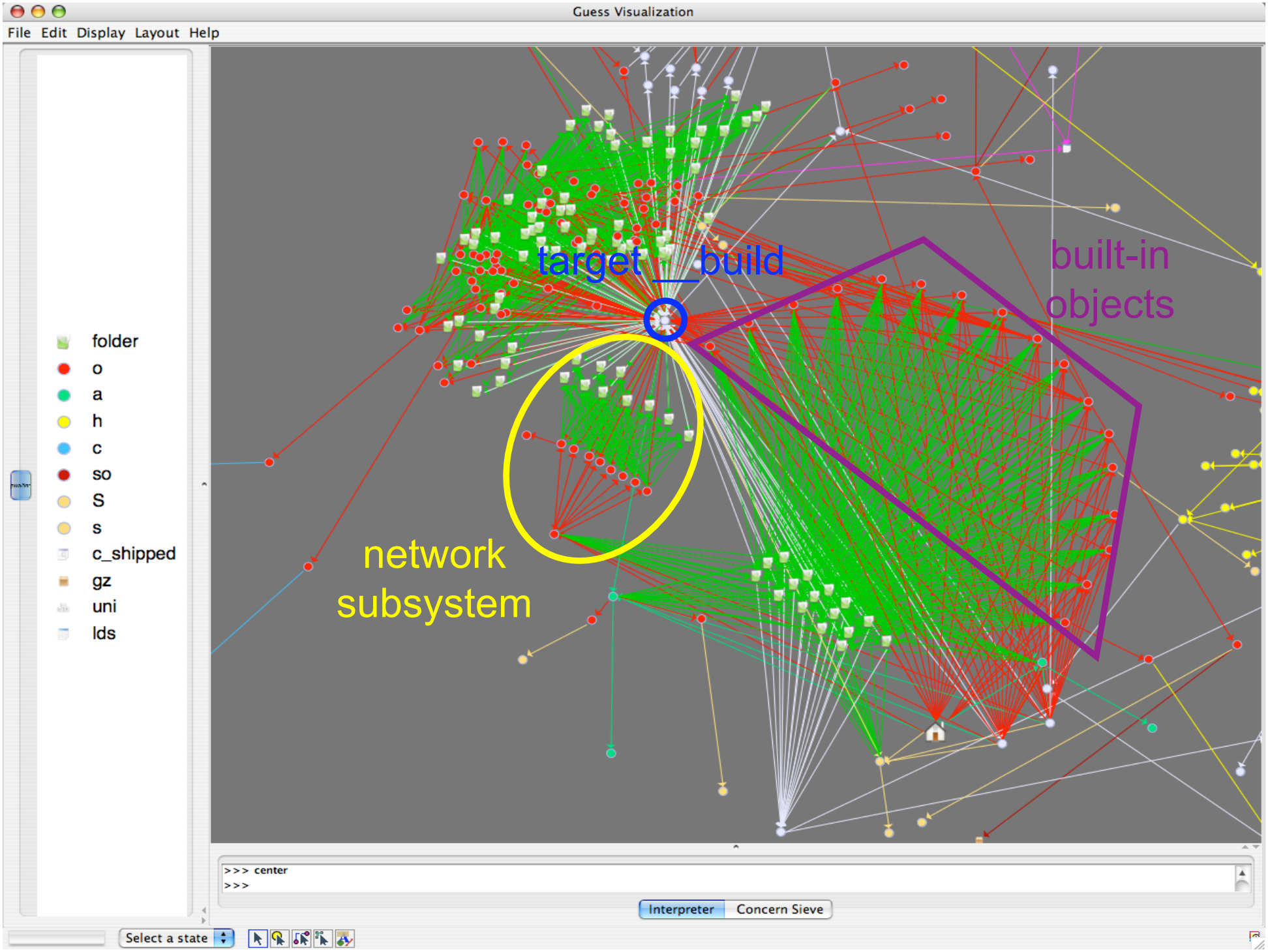
- non-recursive make
- wrappers and dependency database
- fixed bugs
- no stepwise migration plan

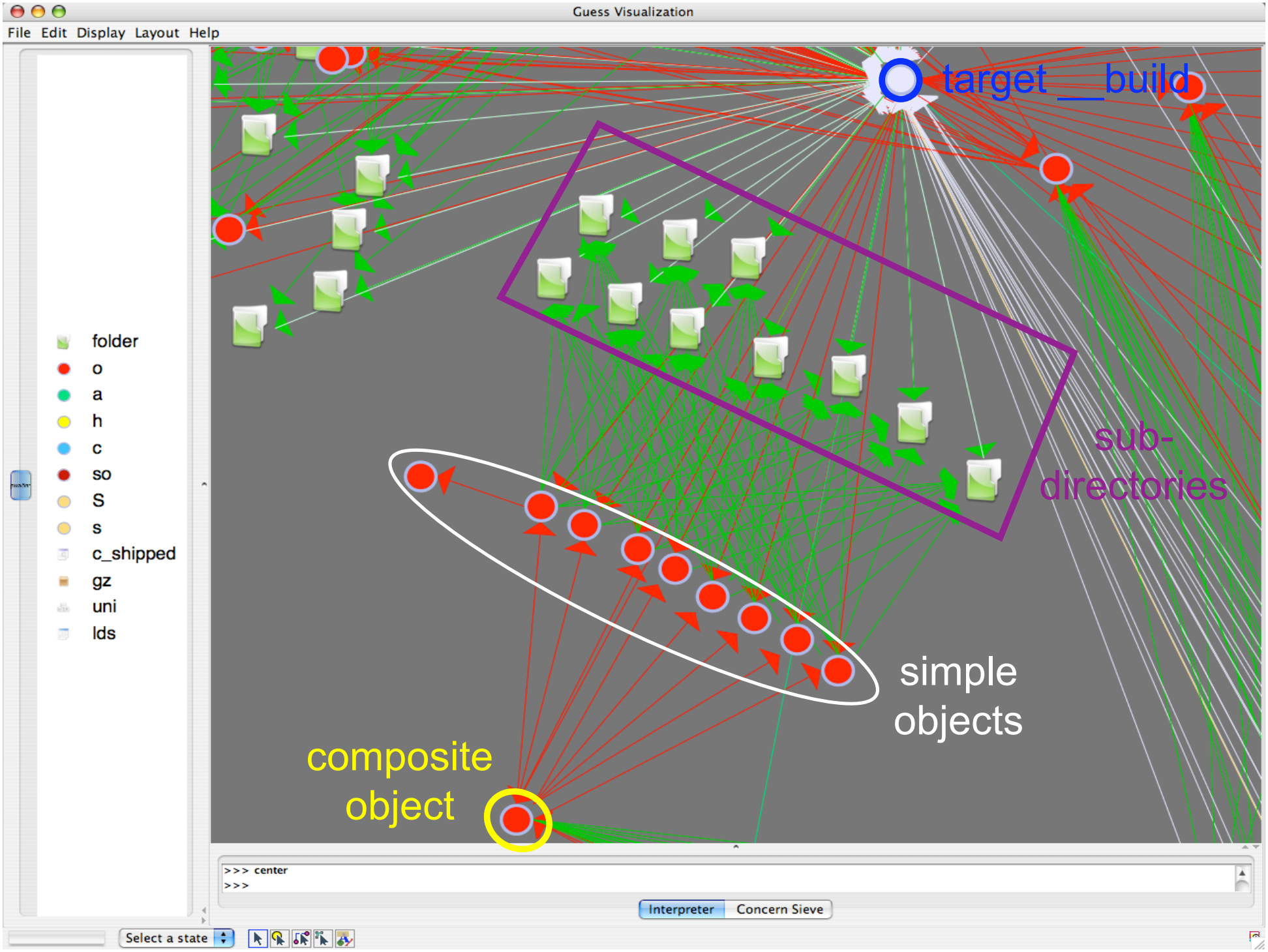
Kai Germaschewski:

- gradually migrates 2.4 build system to 2.6 kernel
- subdirectory builds don't need full dependency graph
- "dep" phase removed
- smart use of list-style makefiles
- build idioms, e.g. "circular dependency chain"

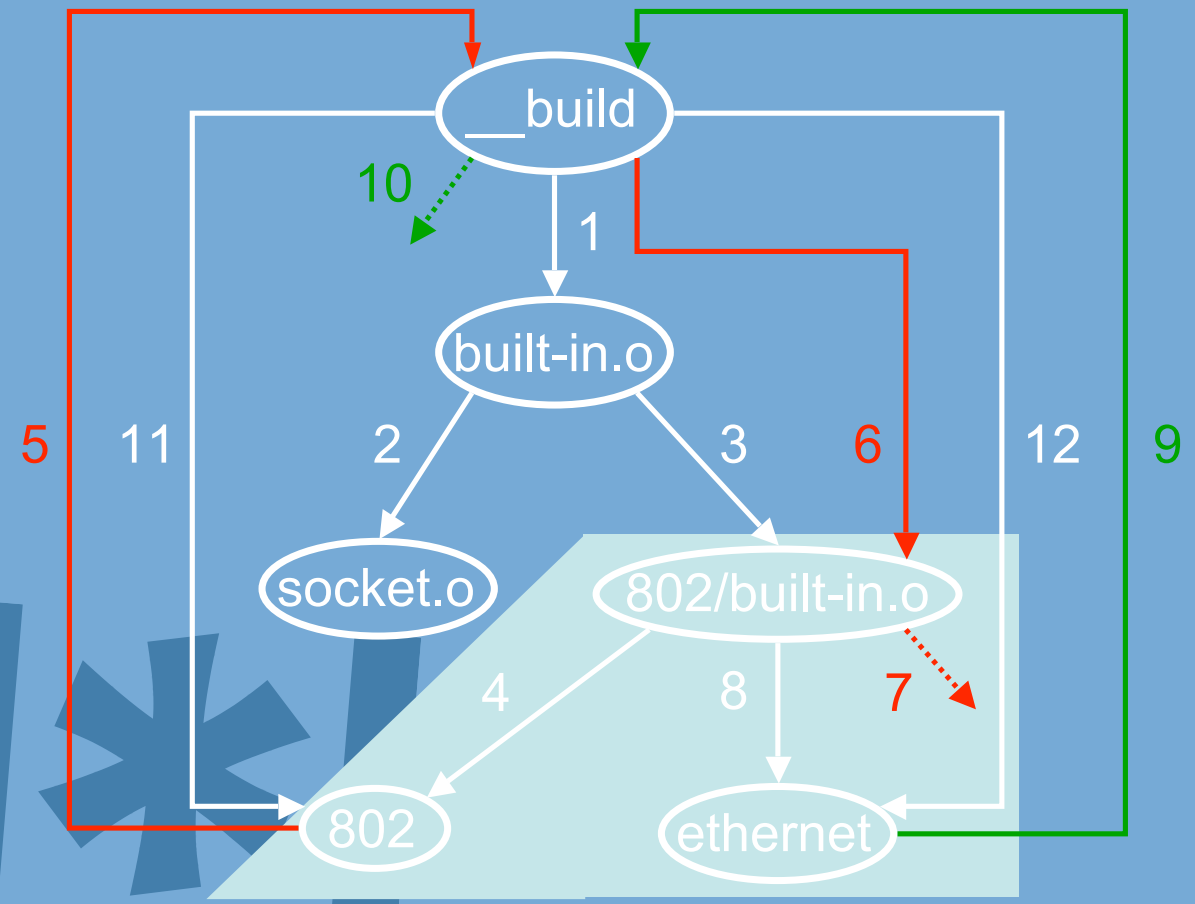


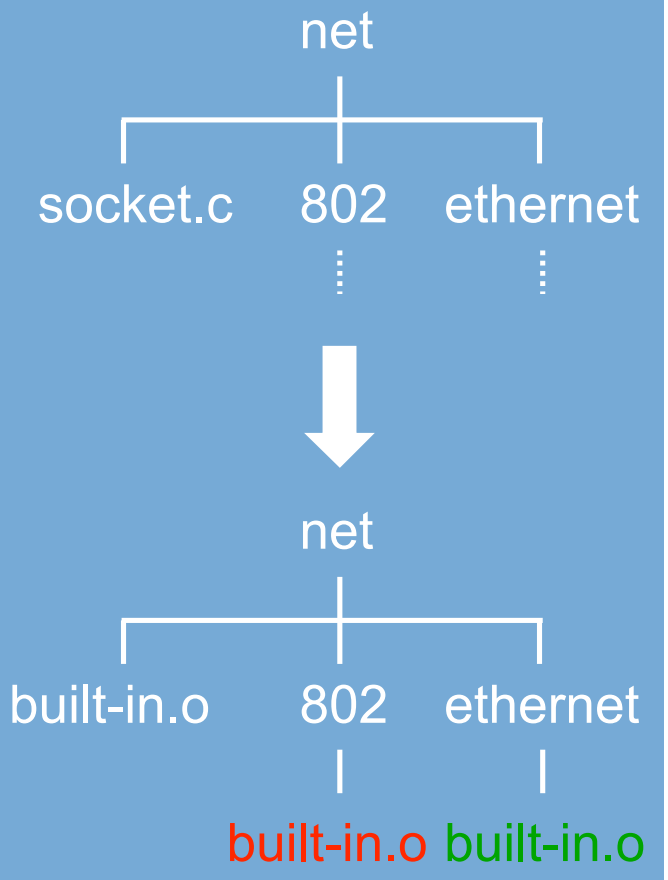

Guess Visualization
File Edit Display Layout Help
class
o
a
h
c
so
S
s
Linux
2.6.16.18
(without FORCE)
>>> center
>>>
Interpreter
Concern Sieve
Select a state


Guess Visualization
File Edit Display Layout Help
folder
o
a
h
c
so
S
s
c_shipped
gz
uni
lds
target __build
built-in
objects
network
subsystem
>>> center
>>>
Interpreter
Concern Sieve
Select a state


Guess Visualization
File Edit Display Layout Help
target __build
sub-
directories
simple
objects
composite
object
folder
o
a
h
c
so
S
s
c_shipped
gz
uni
lds
>>> center
>>>
Interpreter
Concern Sieve
Select a state

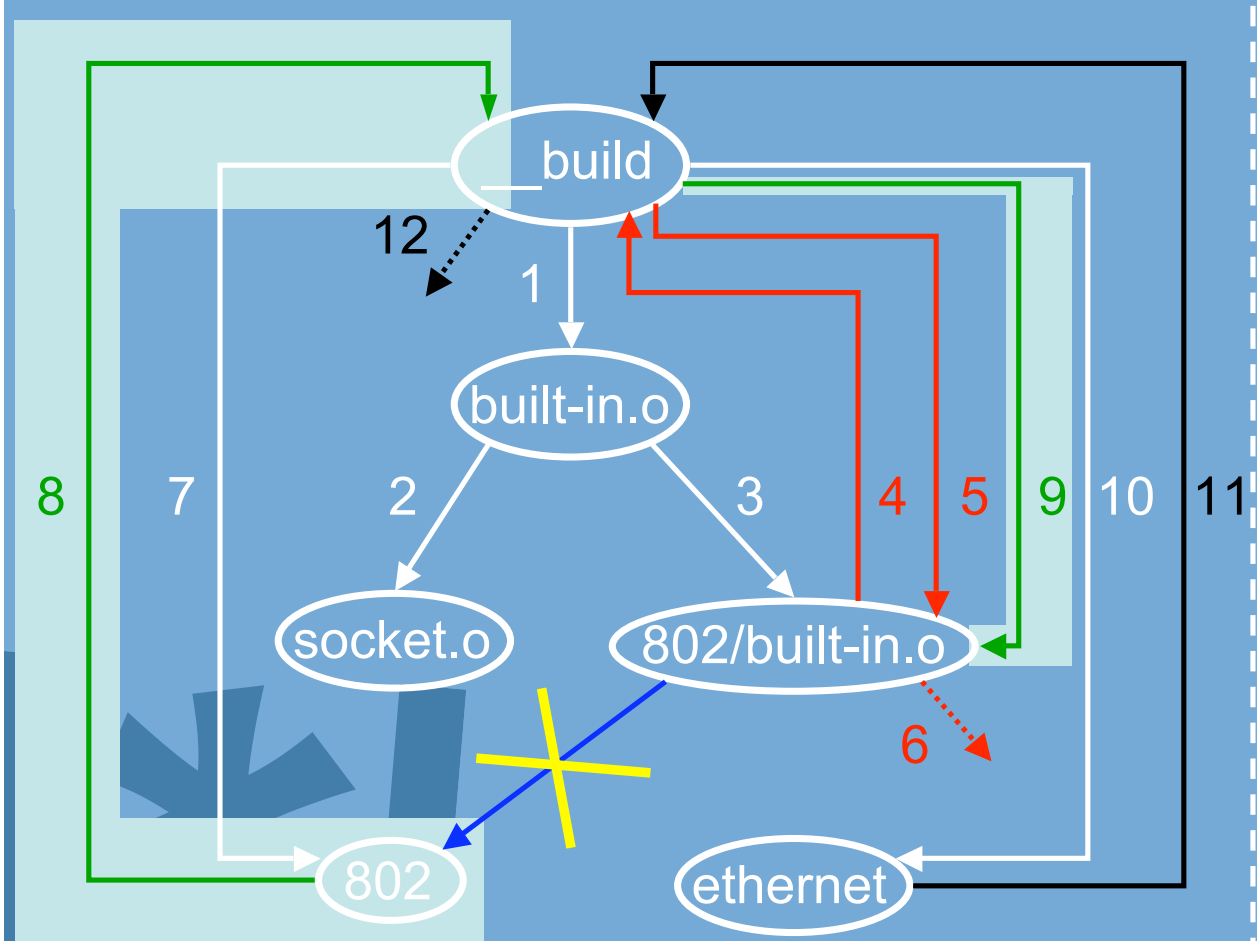
5d. Circular Dependency Chain (1)

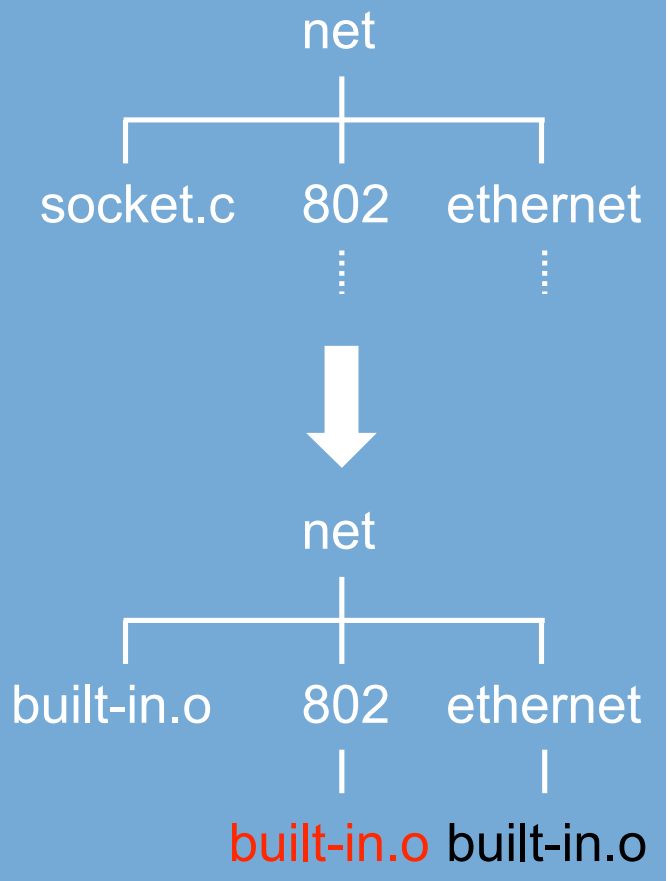




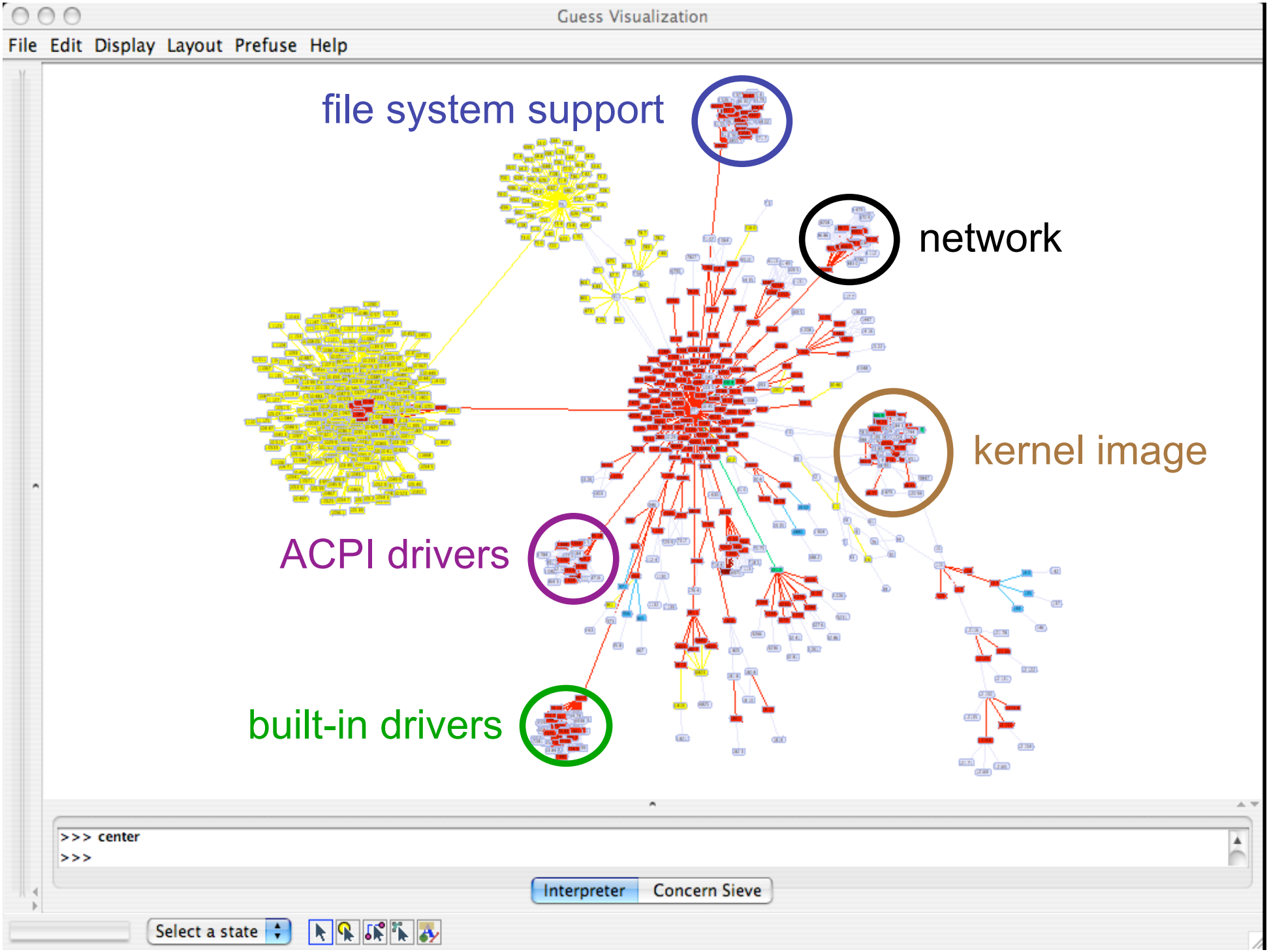

```
$(sort $(subdir-obj-y)): $(subdir-ym) ;
```

5d. Circular Dependency Chain (2)






```
$(sort $(subdir-obj-y)): ; $(Q)$(MAKE) $(build)=$(@D)
```


Guess Visualization
File Edit Display Layout Prefuse Help
file system support
network
kernel image
ACPI drivers
built-in drivers
>>> center
>>>
Interpreter
Concern Sieve
Select a state

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

6. Discussion: co-evolution?

Linux:

- build system clearly evolved to cope with developer needs
- safest migration path chosen ↔ at expense of speed and parallelism
- "correctness trumps efficiency"
- contacted build maintainers

In general:

- is Linux case representative for open/closed-source software?
- nature of changes:
  - correctness
  - speed
  - configurability
  - changes in source code?
- architecture mining?

other cases needed ↔ with developer docs, discussions, ...

Outline

1. Build System?
2. Co-evolution
3. MAKAO
4. Case Study
5. The 2.6 kernel:
    a) Dependency management
    b) FORCE
    c) Recursive vs. Non-Recursive make
    d) Circular Dependency Chain
6. Discussion
7. Conclusion

7. Conclusion and Future Work

Linux build system analysis:

- build is a valuable and crucial asset
- safety ↔ speed
- build idioms ↔ fundamental build change

} co-evolution

Lehman's laws of software evolution:

- build system evolves
- during evolution it grows in complexity
- maintenance performed to manage this complexity

TODO:

- causal link between source code re-engineering and build system changes?
- configuration-aware study, also on other systems