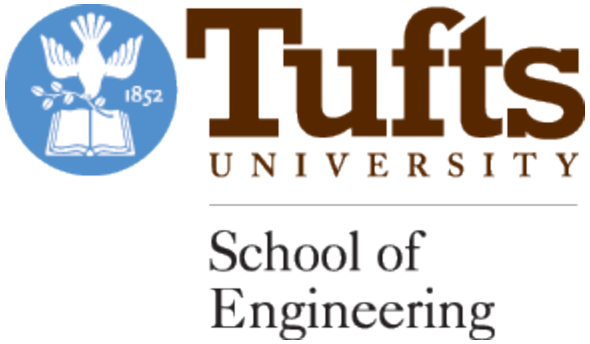


A theory of closure operators

Alva L. Couch

Marc A. Chiarini

Tufts University

Convergent operators

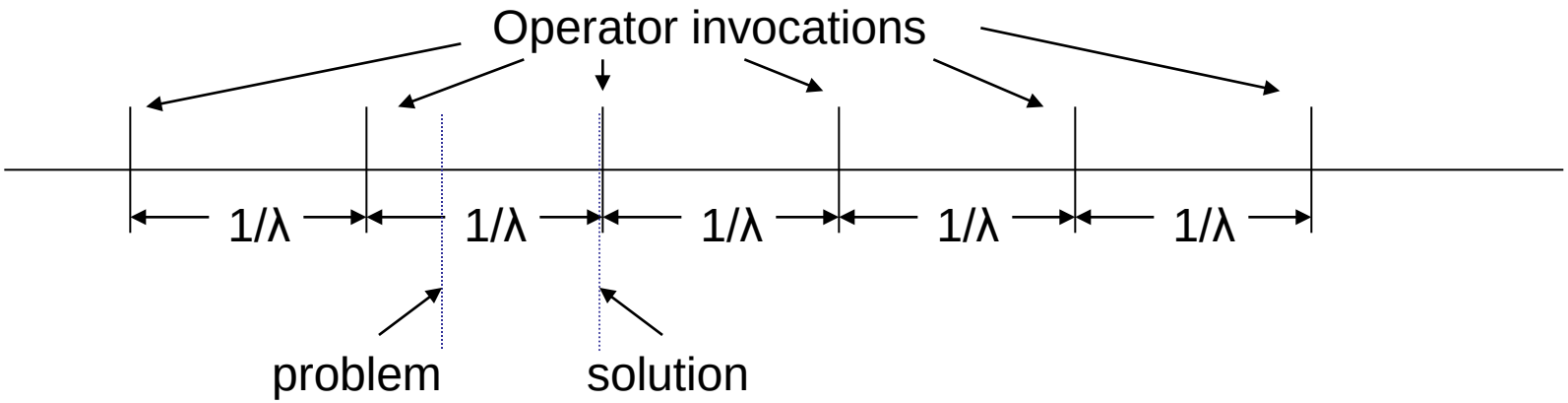
- A *convergent operator* assures a specific network state and is idempotent if that state exists already.
- Example 1: set a parameter to a value.
- Example 2: deploy a service.
- CFEngine implements a set of convergent operators for system management.

What does convergence mean?

- Convergent operators “immunize” the system against harmful degradations.
- Example 1: if a parameter ever changes to a less desirable value, change it back.
- Example 2: if a service stops working, either restart or redeploy it.

CFEngine “immunization”

- Repeatedly invoke operators at some (approximate) rate λ.
- Problems have max. lifetime of about 1/λ.

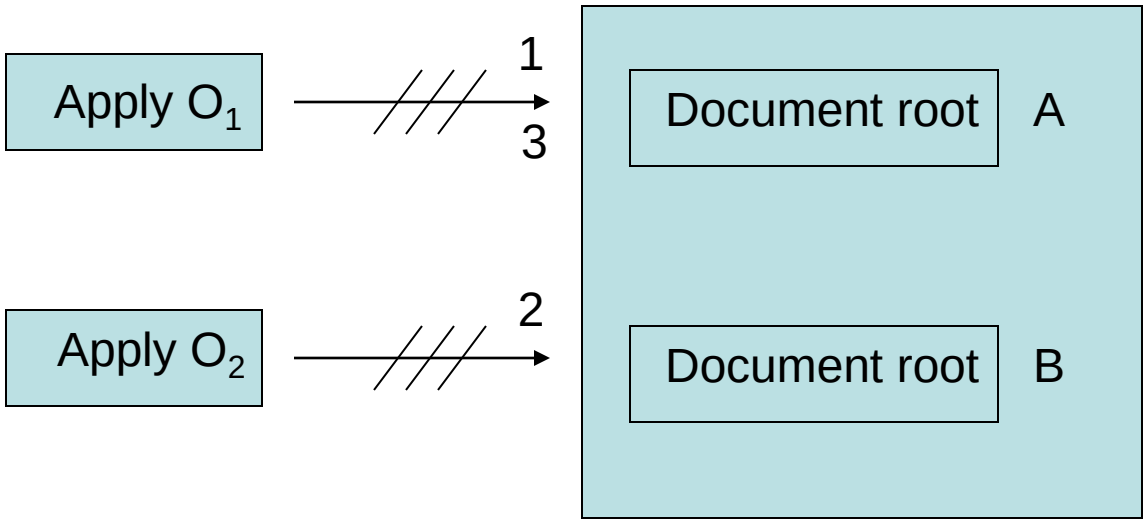


Assurance and acceptance

- **CFEngine operators have one limitation.**
- We say a state is *assured* by an operator if applying it changes the system to reflect that state.
- We say a state is *accepted* by an operator if it will not change that state to another.
- **Most CFEngine operators *assure exactly the states they accept.***
- This simplifies the *mathematics*, but creates some problems in *engineering* self-managing systems.

Dueling operators (in CFEngine)

- Suppose O_1 sets up a web server and O_2 tunes its performance.



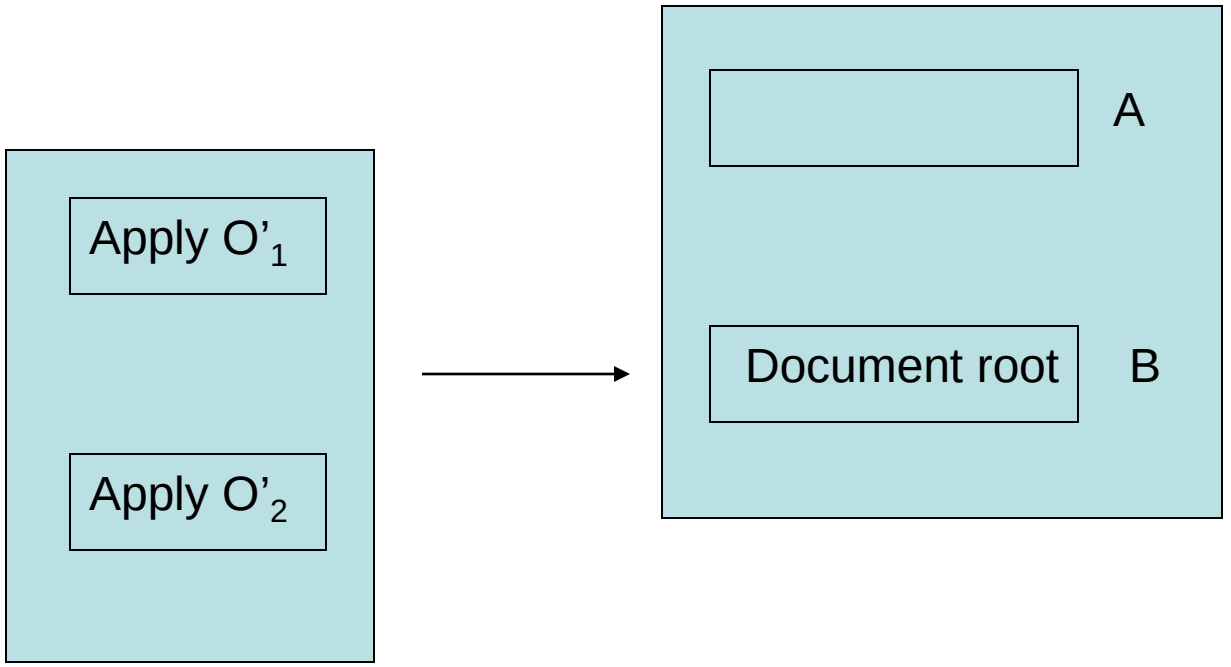
Chooses default document root

Oscillates forever between options A and B

Chooses faster document root

One way to resolve the conflict…

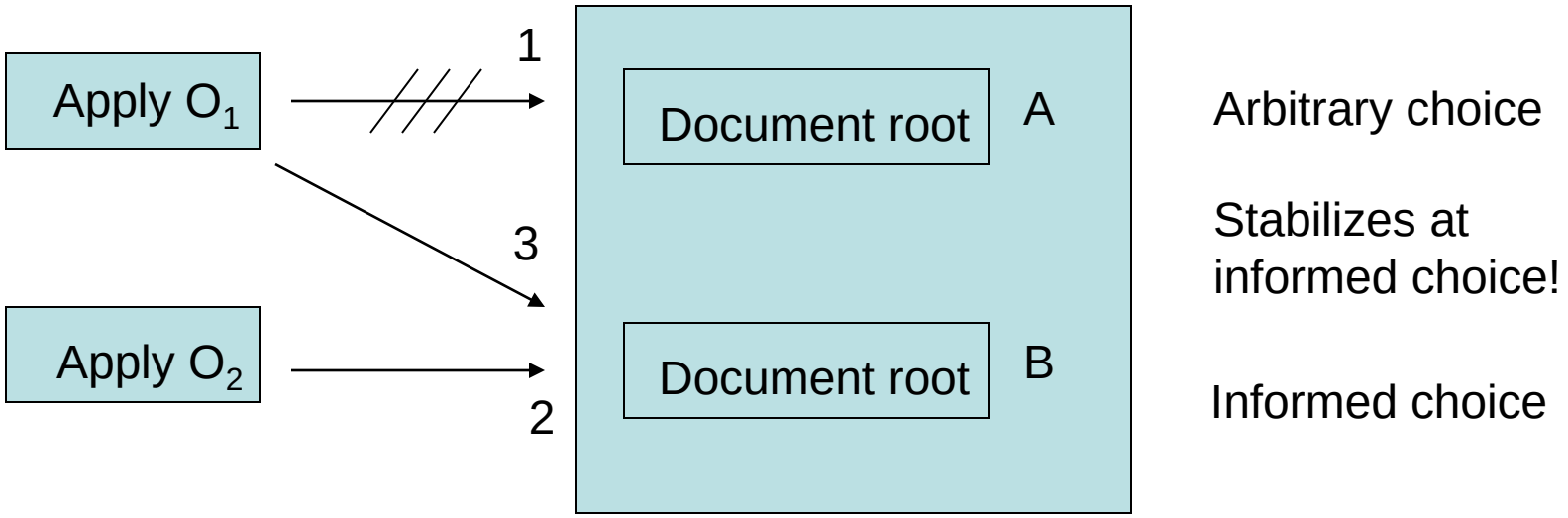
- Encapsulate O_1 and O_2 inside one operator.



Stable because of encapsulation

What we really want to happen:

- O_1 and O_2 “collaborate” and “share knowledge”:

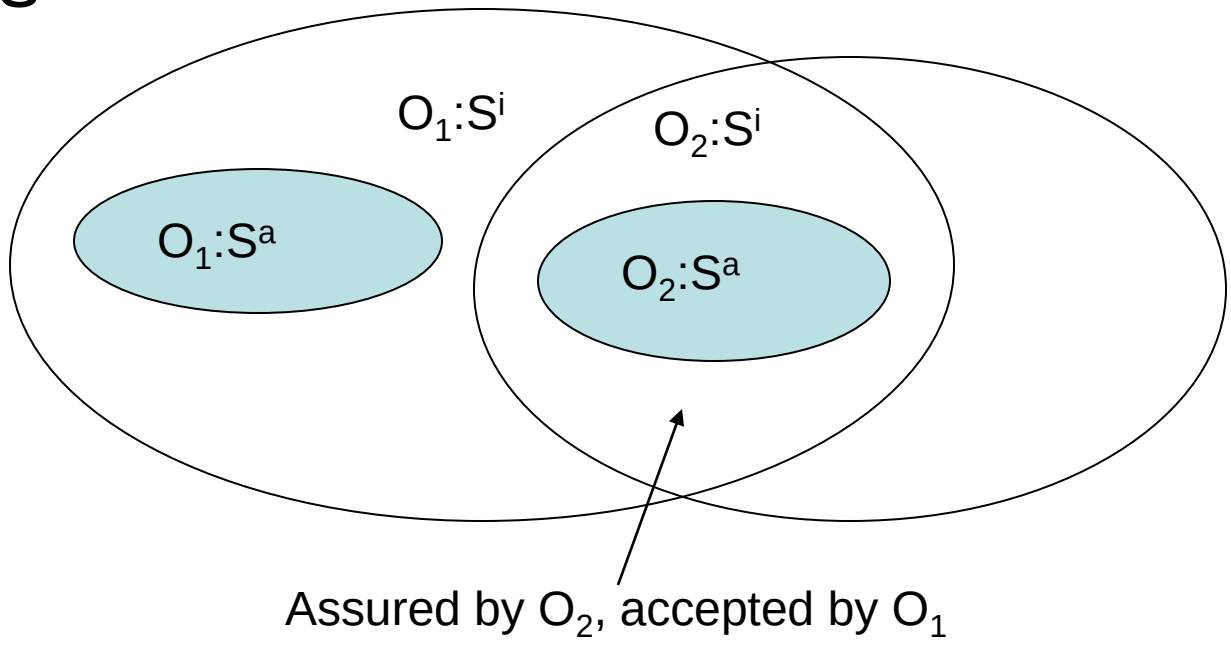


Collaboration is difficult

- For O_1 and O_2 to collaborate rather than dueling, O_1 must *accept more states than it assures*.
- This means that O_1 must base its actions on a *model of what a healthy webserver looks like*.

Statespace view

- O_1 deploys a web server, O_2 tunes it.
- S^a are assured states; S^i are accepted states



Closures

- A *closure* is a self-managing part of an otherwise (perhaps) open system.
- Key concepts:
  - Hides *control loops* inside a black box.
  - Hides *incidental complexity:* choices made for no justifiable reason need not be made by humans.
- We borrow ideas from closures to improve operators.

Closure operators

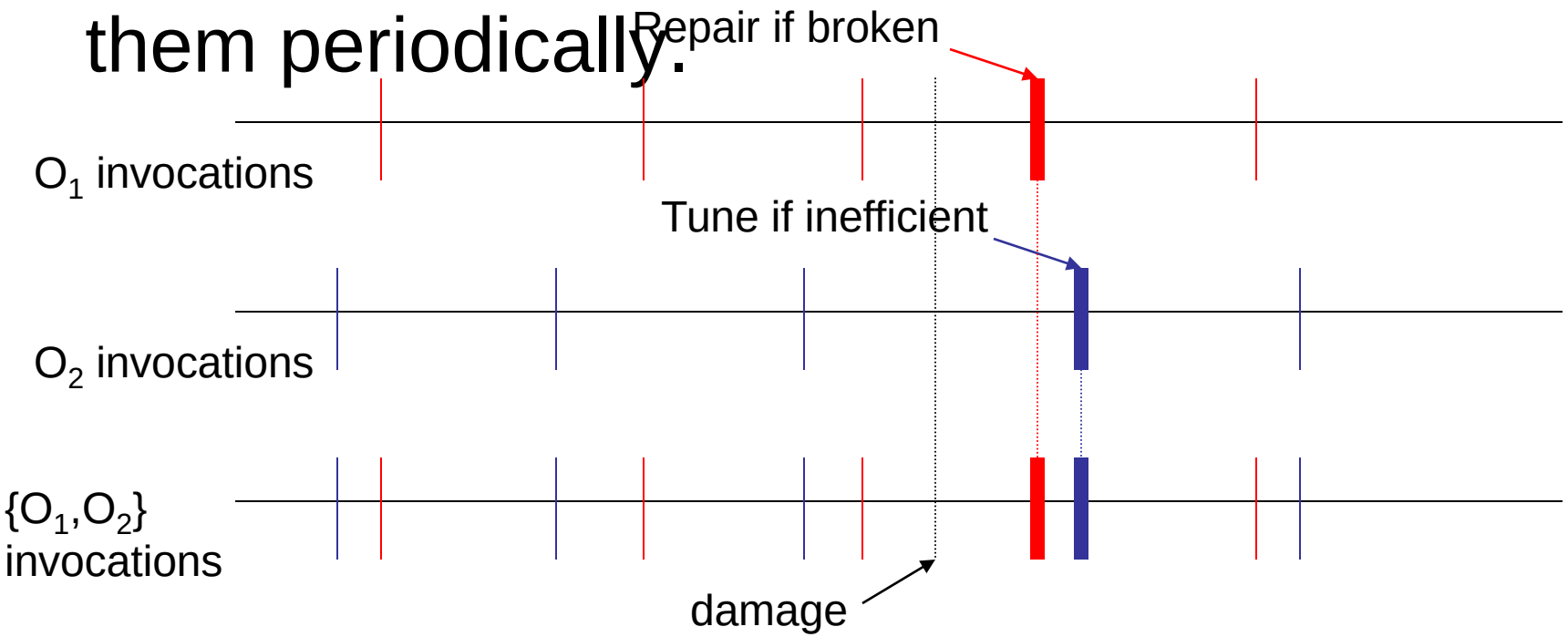
- A *closure operator* is a convergent operator that accepts more states S^i than the states S^a that it assures.
- The difference between sizes of S^i and S^a is a measure of the *incidental complexity* of the operator, i.e., the choices that it makes for which it does not have strong justifications.
- One operator's incidental choice may be another operator's informed decision.

Goal of closure operator theory

- Allow each operator to make *incidental choices.*
- Allow other operators to replace incidental choices with *informed choices.*
- Applying a set of operators *composes knowledge* embodied in all of them.

Composing closure operators

- O_1 repairs a web server.
- O_2 tunes a web server.
- Their “composition” is to invoke both of them periodically.



(Relatively straightforward) properties of closure operators

- If a set of operators act on orthogonal subsystems, then their composition is a closure operator.
- If a set of operators shares the same acceptance set and a reachable fixed point, then their composition is a closure operator.

Modeling

- Difficulty in creating a closure: how does one define or specify the acceptance set?
- The assurance set is defined *procedurally:* “here’s how to create a state.”
- The acceptance set is defined *declaratively:* “these states are fine if they are present.”

Example: what is an appropriate document root?

- There must be a document root.
- It must appear in several places in the configuration file.
- The same document root must be specified everywhere it is needed.
- If O_1 understands this, then O_2's assured state will be accepted by O_1, and there will be no duel.

Future work

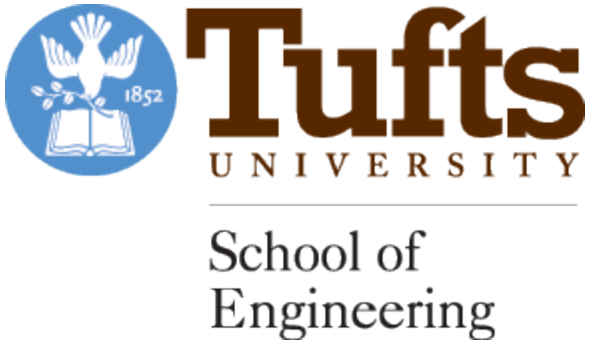
- We know how to construct “a few” closure operators with appropriate properties.
- Next step: design how to incorporate these concepts into CFEngine.
  - Use a modeling language to define CFEngine soft classes.
  - Use soft classes to invoke corrective actions.

Conclusions

- CFEngine operators currently assure what they accept.
- By using a constraint model, they can accept more than they assure.
- This can be used to *compose knowledge* of multiple operators.

Afterword: HotAC Outcome

- June 2, 2008, Wheeling IL,USA:
  Hot Autonomic Computing attendees identified *three grand challenges*.
- One of the agreed-upon challenges was *control loop composition*.
- Closure operators provide a mechanism for *composing control knowledge.*



Questions?

Alva L. Couch

Marc A. Chiarini

Tufts University