

**Formal Specification
of
Group Communication Middleware**

Mark-Oliver Stehr
University of Illinois at Urbana-Champaign

Joint Work with

Carolyn Talcott
SRI International

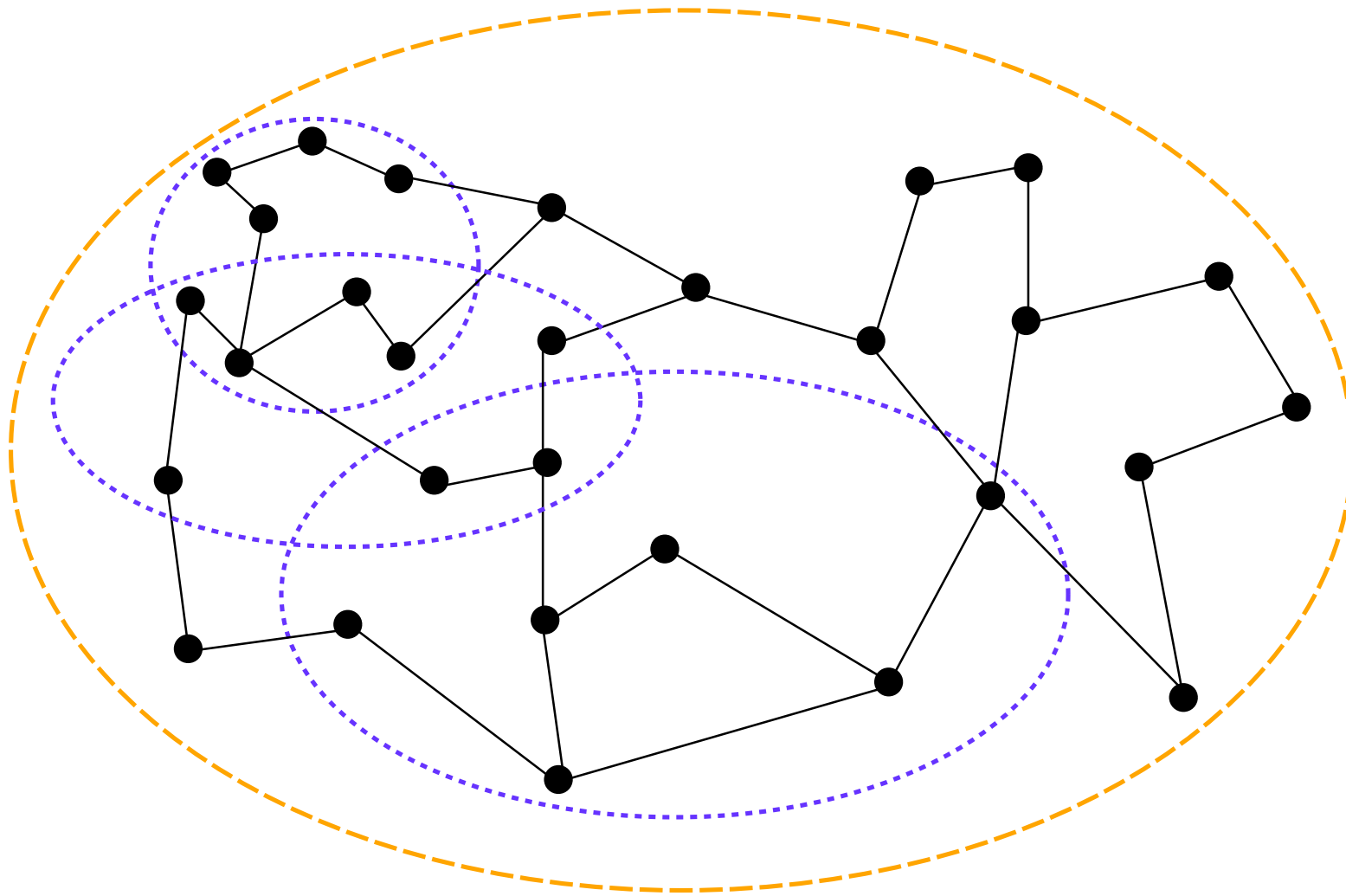
Introduction

- Typical applications:
 - distributed transactions, database replication
 - highly available servers
 - load balancing, system management/monitoring
 - groups as a basis for secure communication
 - collaborative computing
- Challenges:
 - Adaptability to failures of machines and connections
 - Adaptability to group membership changes
- Generations of group communication systems:
 - Assumption of static network topology:
 - Consul, xAMp
 - Possibility of network disconnections:
 - Isis, Phoenix
 - Possibility of network partitions:
 - Transis, Totem, Horus, RMP, Newtop, Relacs, **Spread**
- Key design principle: best-effort message delivery

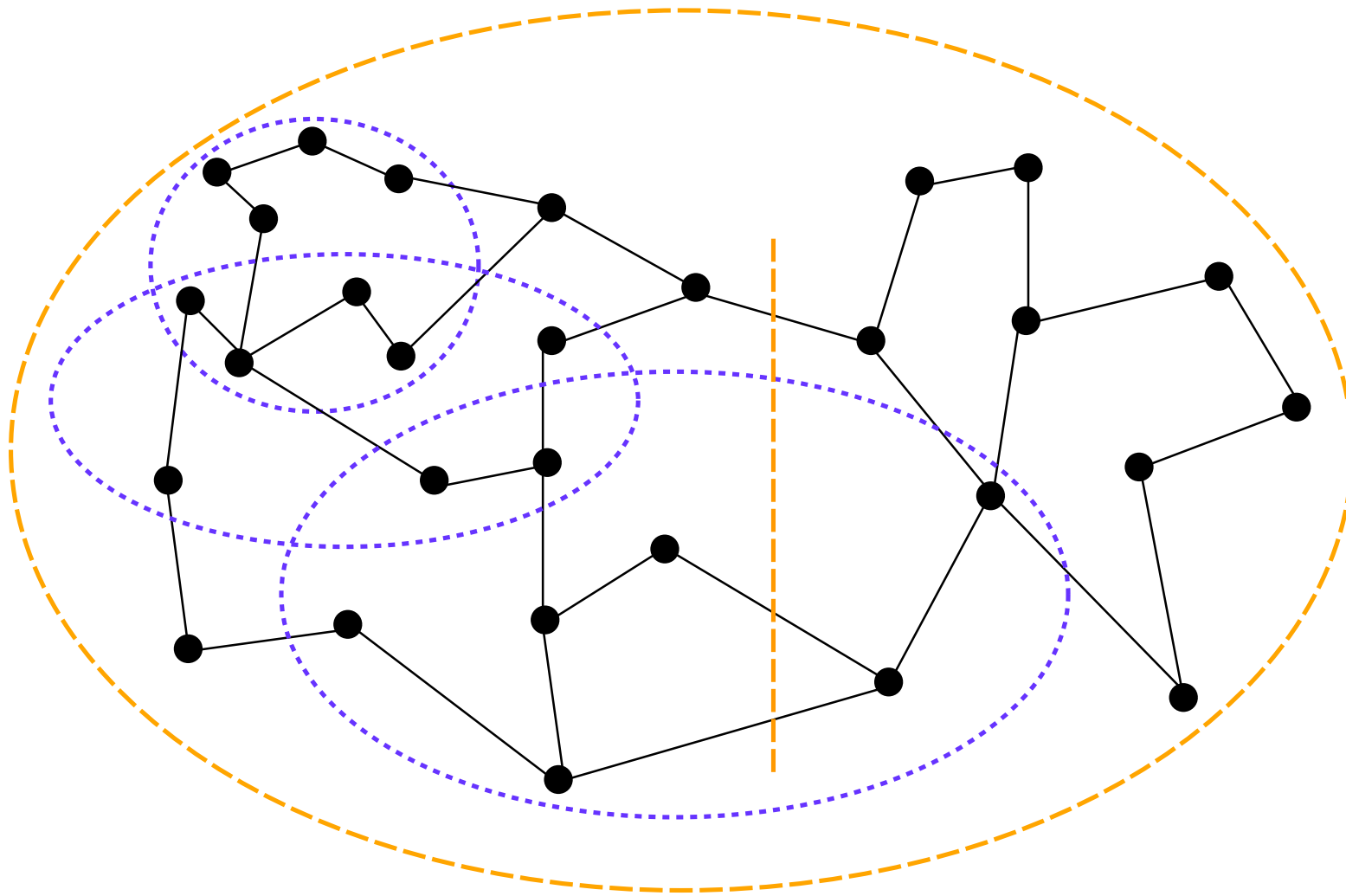
Spread

- Evolved from Transis and Totem
- Developed by Yair Amir, Jonathan Stanton
(Johns Hopkins University)
- Allows Network Partitioning and Merging
- Supports different Ordering Guarantees:
 - Unordered, FIFO Order, Causal Order, Total Order
- Supports different Reliability Guarantees:
 - Unreliable, Reliable, Safe Delivery
- Implementations exist for various systems
(UNIX and Windows)

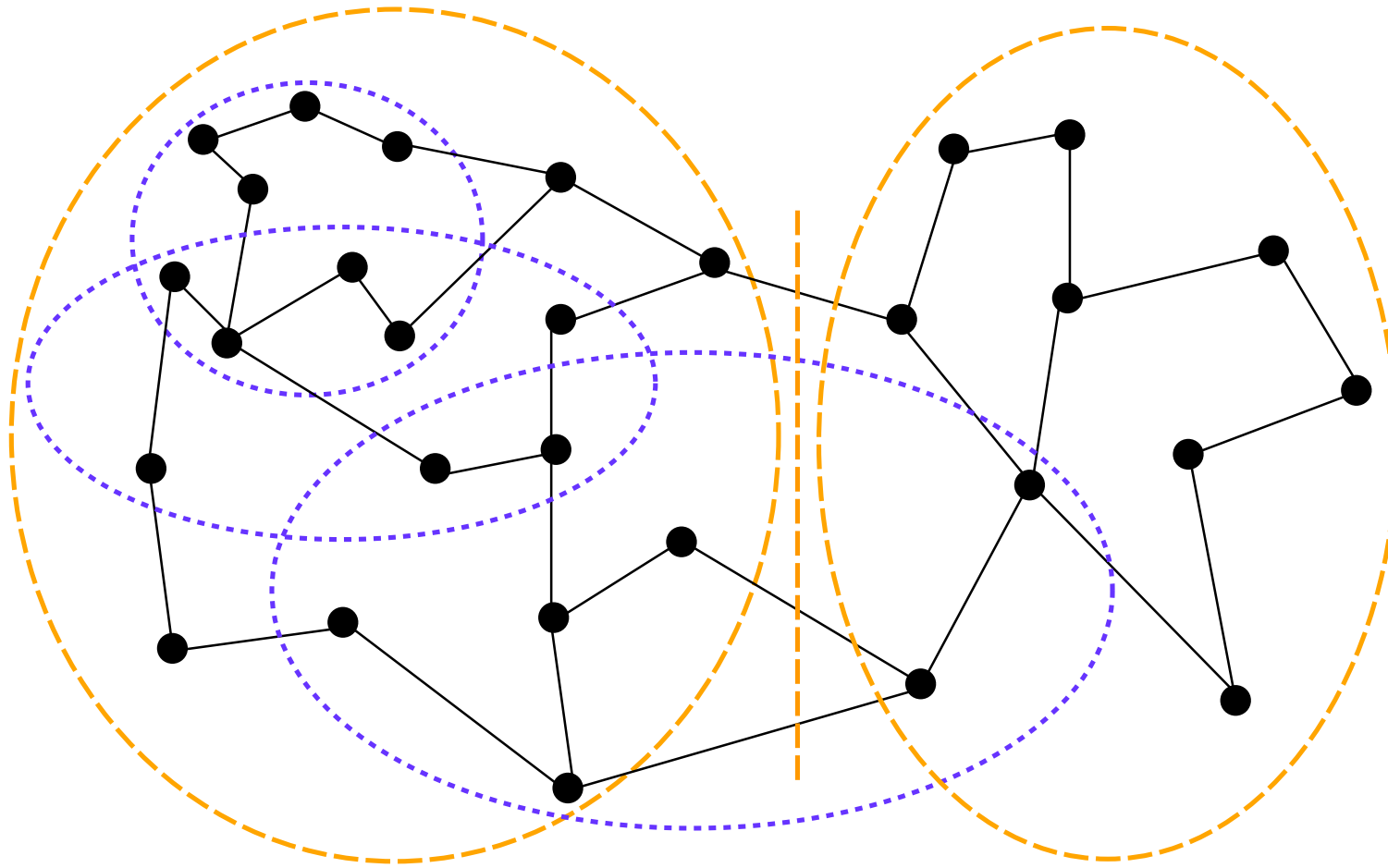
Configurations & Groups



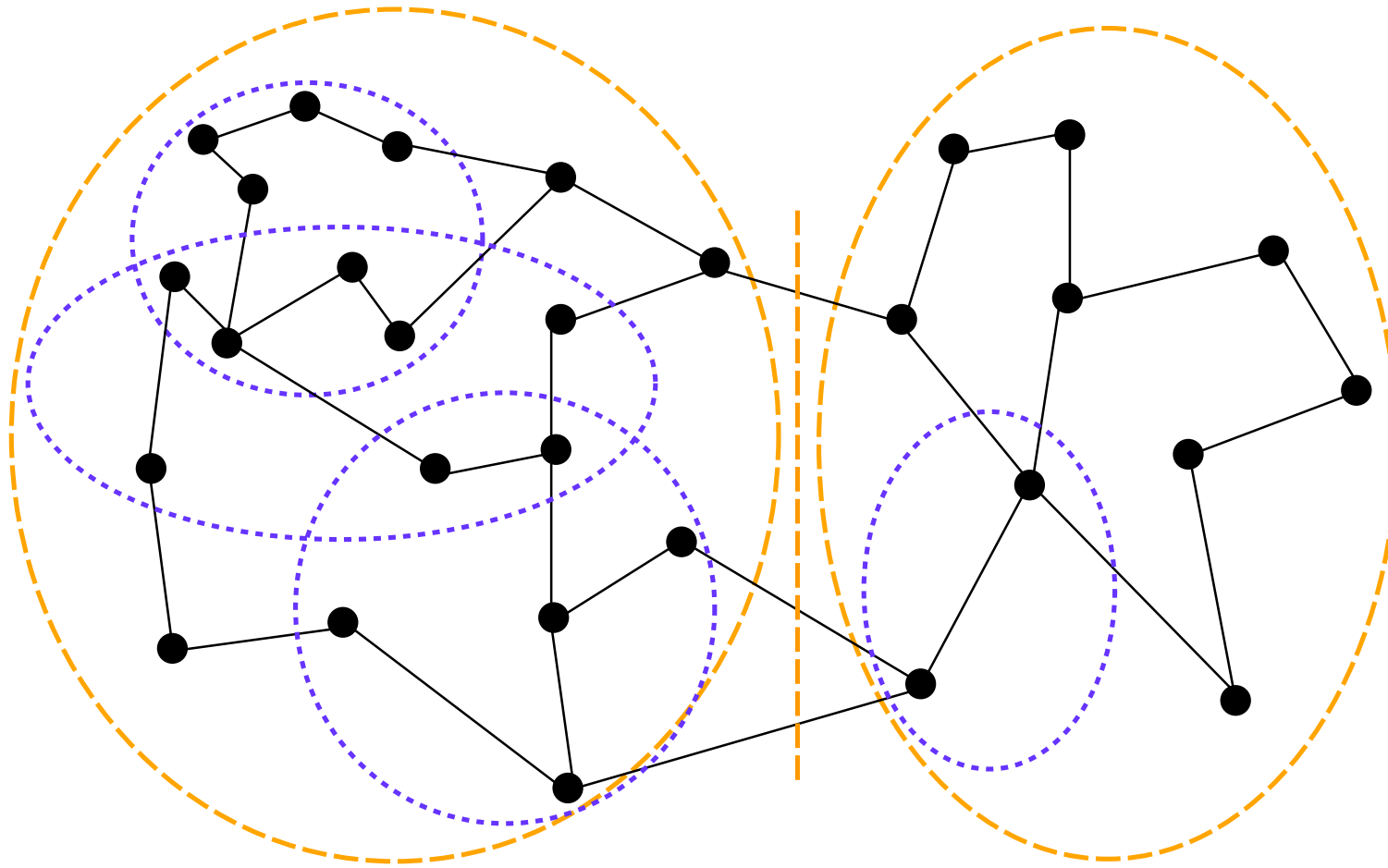
Configurations & Groups



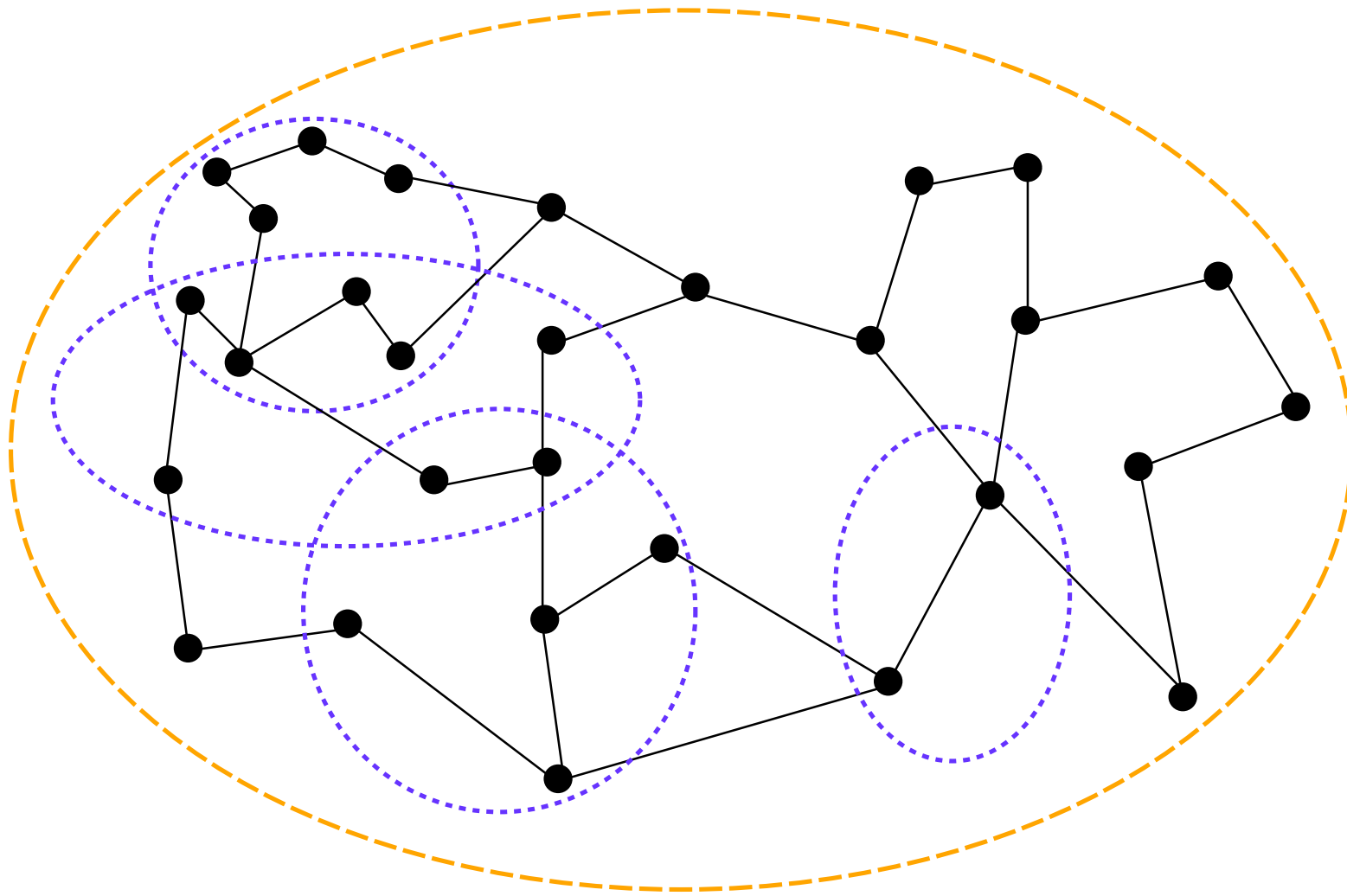
Configurations & Groups



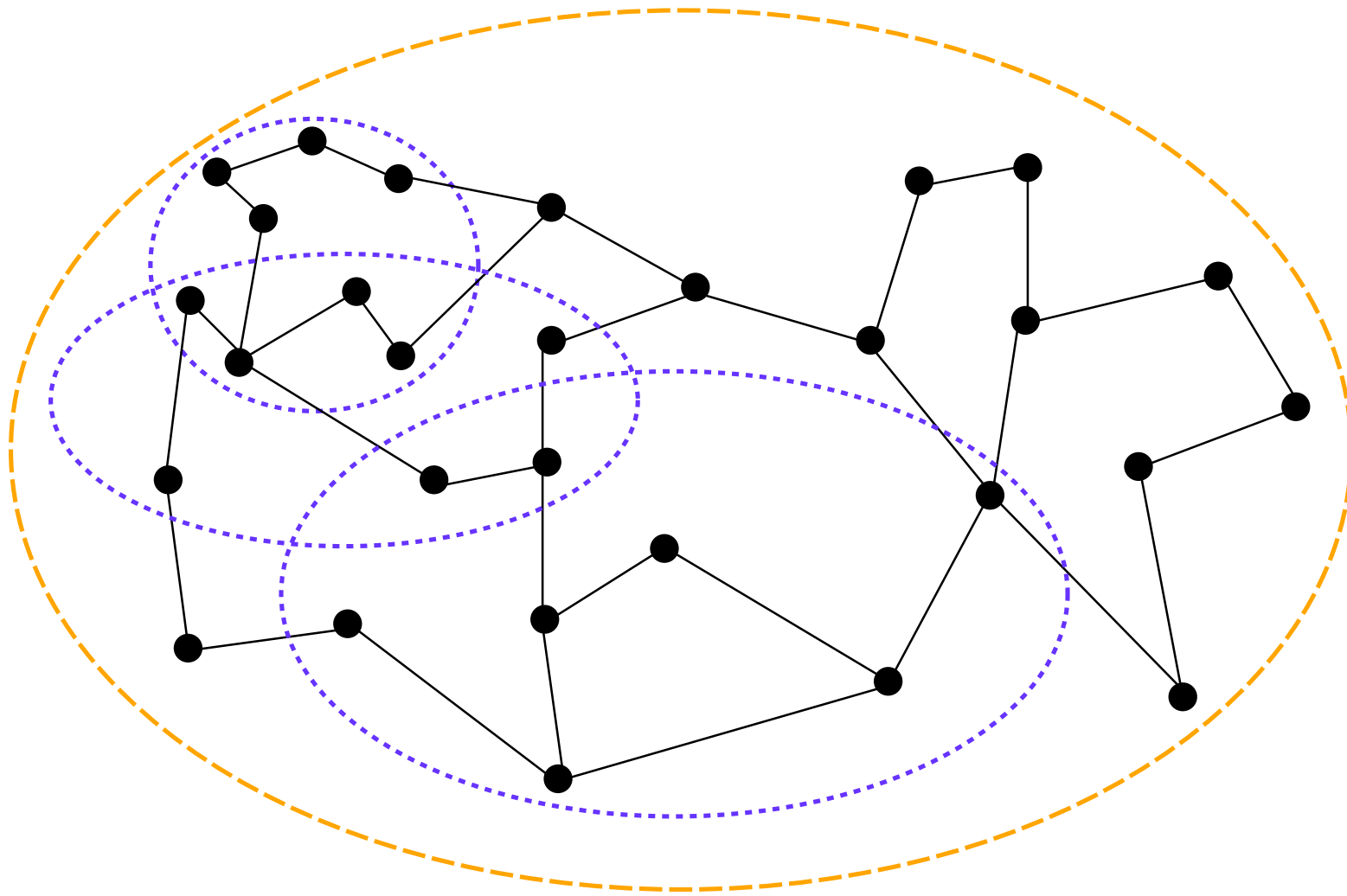
Configurations & Groups



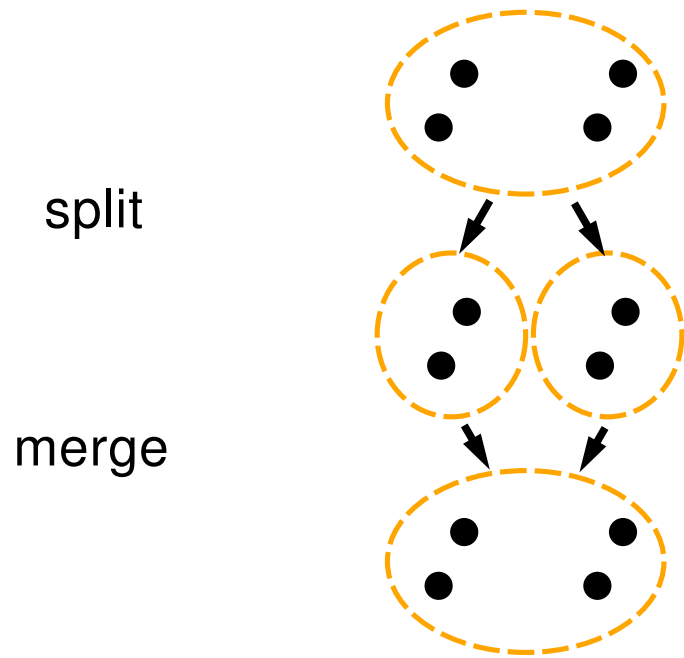
Configurations & Groups



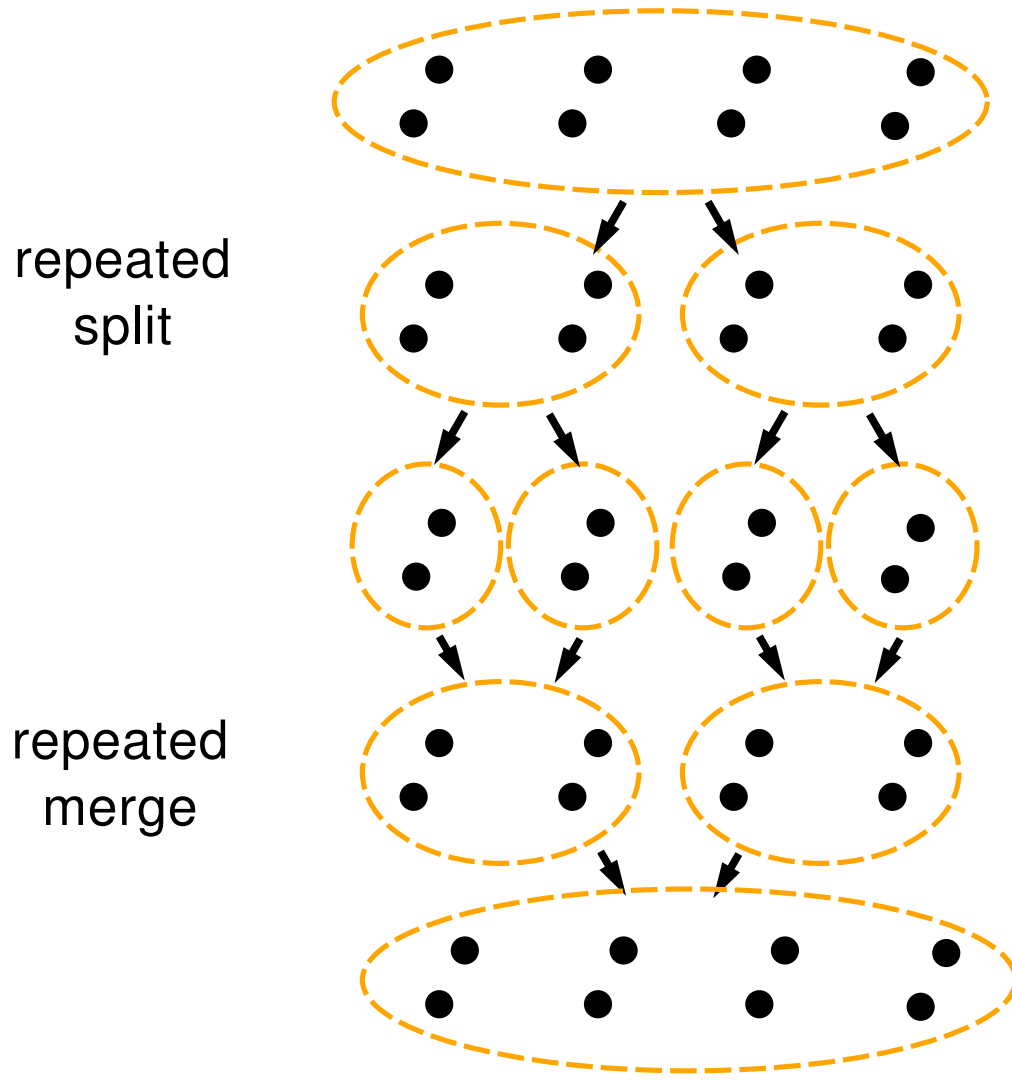
Configurations & Groups



Partial Order of Configurations

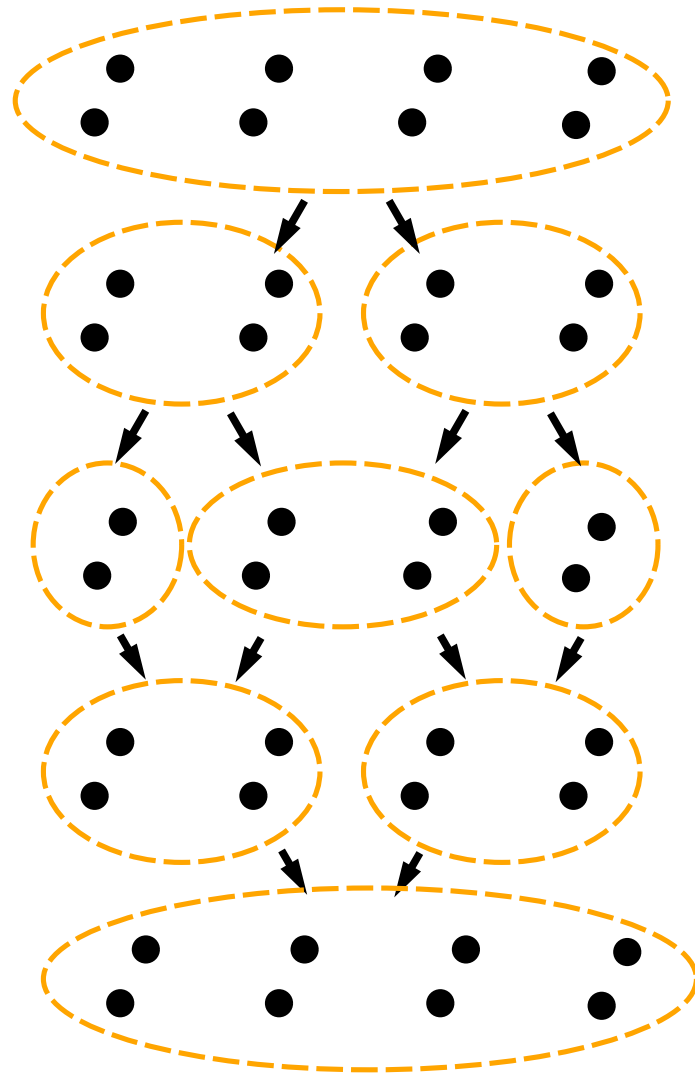


Partial Order of Configurations

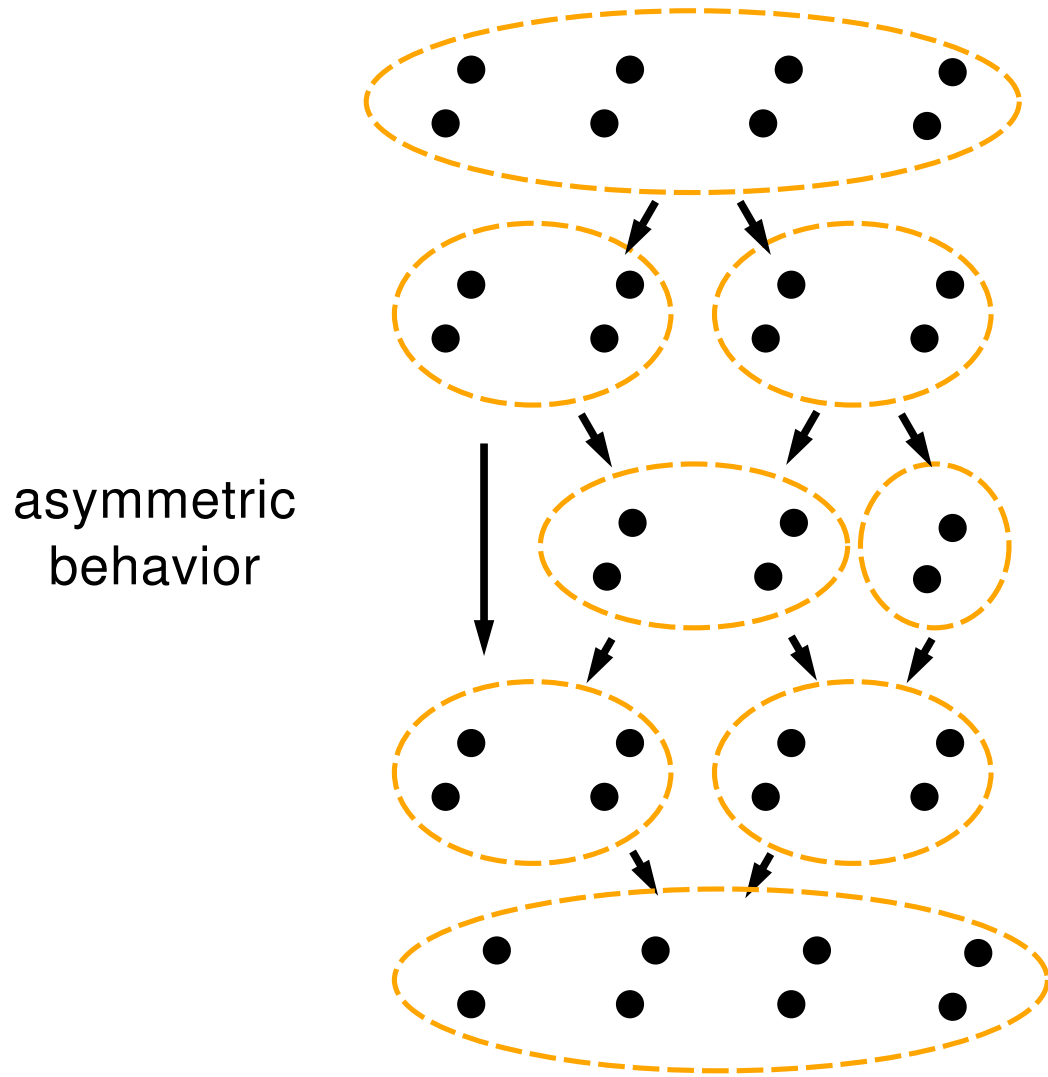


Partial Order of Configurations

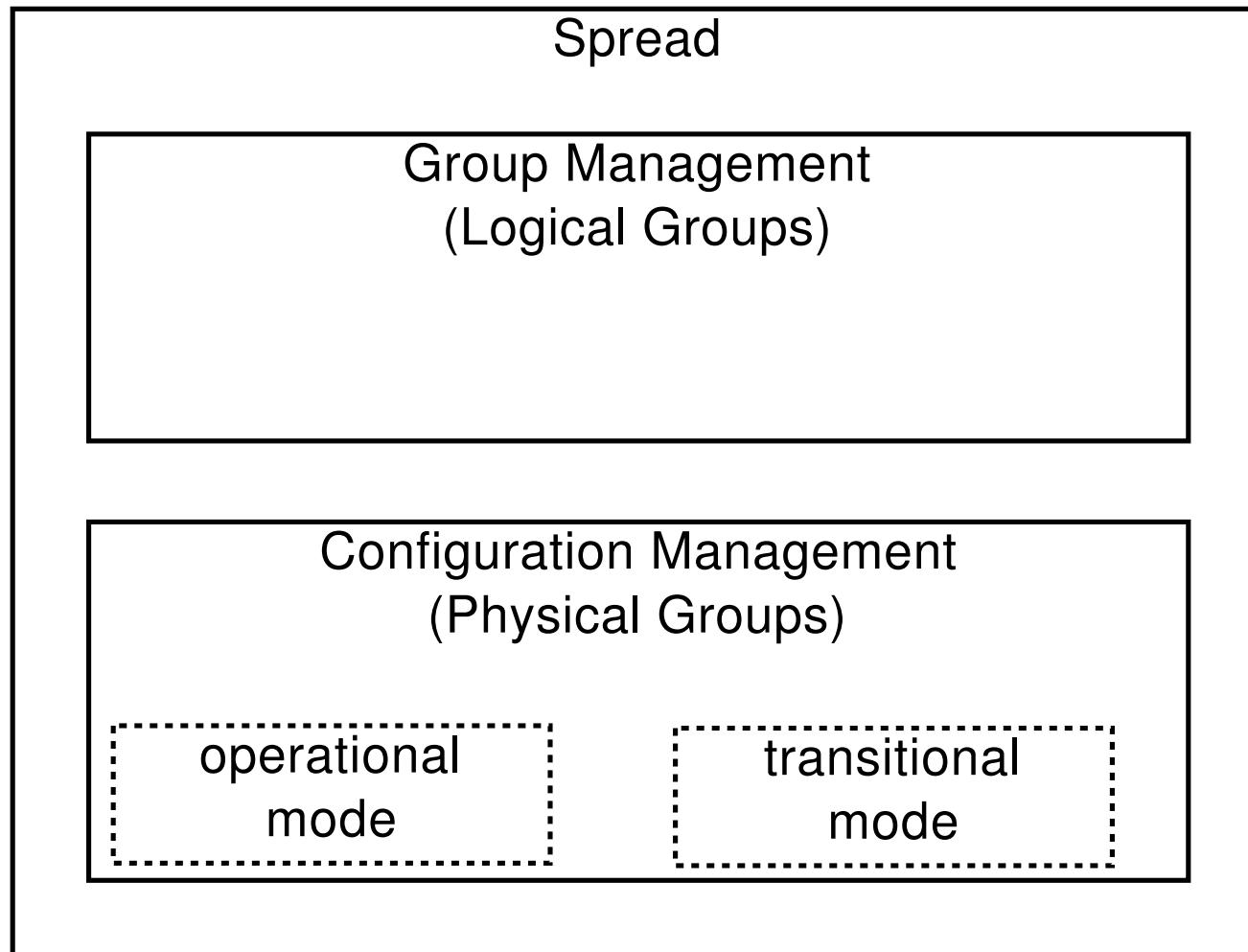
concurrent
split & merge



Partial Order of Configurations



Spread Architecture



The Maude Methodology

- Formal modelling using Maude within rewriting logic and its membership equational sublogic
- Key questions:
 1. Does the formal model adequately capture intended model ?
 2. Does the formal model have the desired properties ?
- Light-weight techniques:
 - Execution
 - State space exploration
 - Model checking
- Heavy-weight Techniques:
 - Informal mathematical proofs
 - Rigorous formal proofs

Towards a Formal Model

- Abstract specification of group communication layer
- Spread is one possible implementation among others
- Use of abstract specification:
 - Verification of Spread itself
 - Verification of layers and applications on top of Spread
- First objective:
 - Specification of the configuration management layer
 - Specification of operational mode
 - Specification of transitional mode
- Challenge: Capture best-effort principle formally
 - Idea: Explicit representation of all delivery constraints
 - Deliver as much as possible under the given constraints

Sending Safe Message

```
rl operational(proc)
  network(everybody)
  localconf(proc,conf)
  localeqs(proc,conf,localeqs)
  knownseqs(proc,conf,knownseqs)
  freshseq(conf,seq)
  delivered(proc,messagelist)
  causalorder(conf,constraints)
  multicast-req(proc,safe,data) =>
  operational(proc)
  network(everybody)
  localconf(proc,conf)
  localeqs(proc,conf,localeqs sNatSet(seq))
  knownseqs(proc,conf,knownseqs sNatSet(seq))
  freshseq(conf,(s seq))
  delivered(proc,messagelist)
  causalorder(conf,(constraints mkConstraintSet(knownseqs,seq)))
  broadcast(everybody,msg-data(proc,safe,conf,seq,knownseqs,data,false))
  multicast-ack(proc) .
```

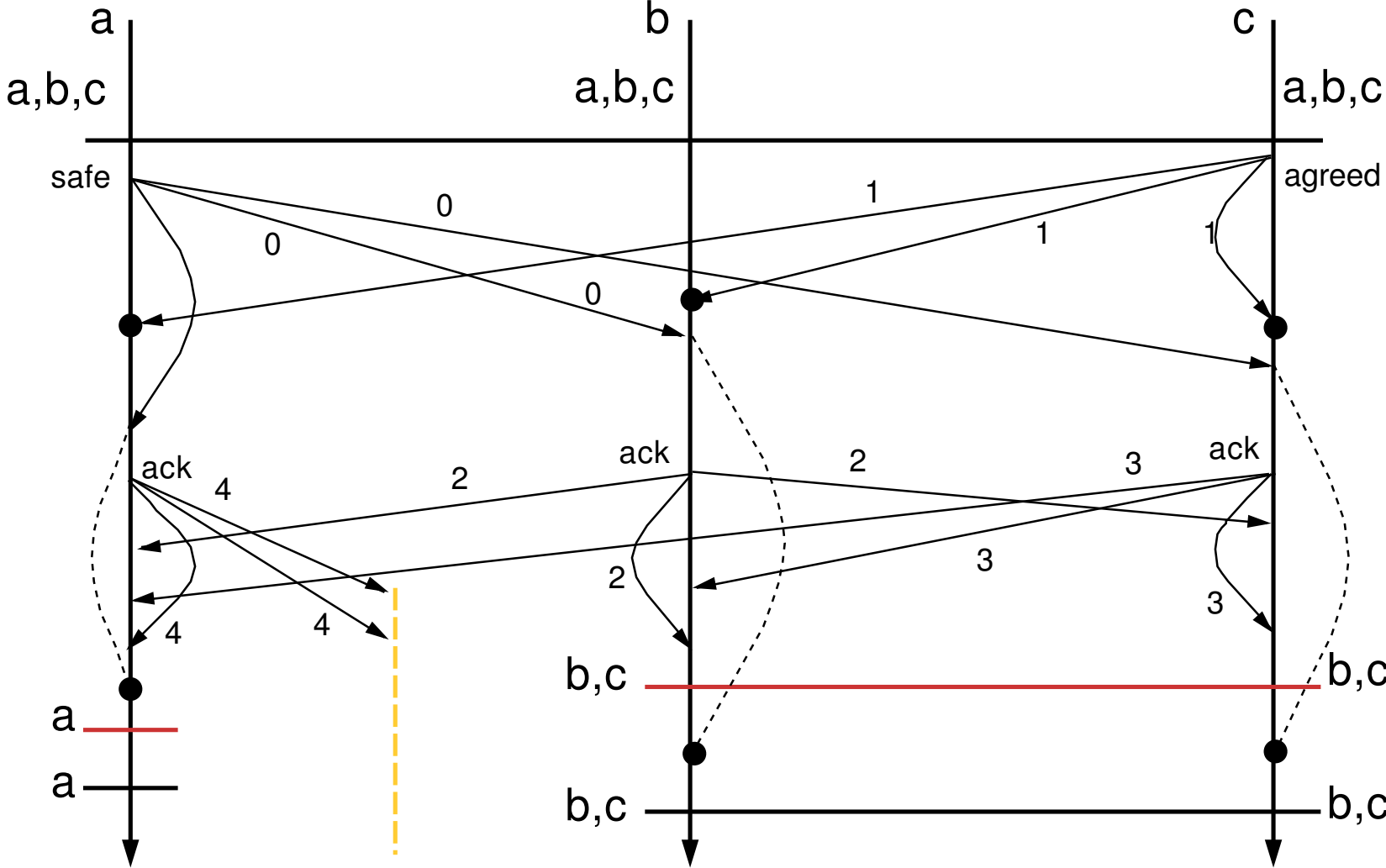
Receiving Safe Message

```
crl operational(proc)
  localconf(proc,conf)
  delivered(proc,delivered)
  knownseqs(proc,conf,knownseqs)
  freshseq(conf,seq)
  received(proc,(sMessageSet(message) received))
  causalorder(conf,constraints)
  totalorder(conf,events) =>
  operational(proc)
  localconf(proc,conf)
  knownseqs(proc,conf,(knownseqs knownseqs(message)))
  freshseq(conf,(s seq))
  received(proc,sMessageSet(setack(message)) received)
  delivered(proc,delivered)
  causalorder(conf,constraints)
  totalorder(conf,events)
  broadcast(members(conf),msg-ack(proc,conf,seq,seq(message)))
  if deliverable(proc,conf,received,delivered,message,constraints,events) and
    mode(message) == safe and not(isacked(message)) .
```

Delivering Safe Message

```
crl operational(proc)
  localconf(proc,conf)
  delivered(proc,delivered)
  knownseqs(proc,conf,knownseqs)
  received(proc,sMessageSet(message) received)
  causalorder(conf,constraints)
  totalorder(conf,events) =>
  operational(proc)
  localconf(proc,conf)
  knownseqs(proc,conf,(knownseqs knownseqs(message)))
  received(proc,received)
  delivered(proc,(delivered sMessageList(message)))
  causalorder(conf,constraints)
  totalorder(conf,(events sEventList(event(src(message),seq(message))))))
  if deliverablesafe(proc,conf,received,delivered,message,constraints,events) and
    mode(message) == safe and isacked(message) .
```

Typical Scenario



Initial Configuration

```
network(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")))

operational(proc("a")) operational(proc("b")) operational(proc("c"))

reachable(proc("a"), sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")))
reachable(proc("b"), sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")))
reachable(proc("c"), sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")))

sent(proc("a"), eMessageSet)
sent(proc("b"), eMessageSet)
sent(proc("c"), eMessageSet)

received(proc("a"), eMessageSet)
received(proc("b"), eMessageSet)
received(proc("c"), eMessageSet)

delivered(proc("a"), eMessageList)
delivered(proc("b"), eMessageList)
delivered(proc("c"), eMessageList)

localconf(proc("a"), regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))
localconf(proc("b"), regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))
localconf(proc("c"), regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))

causalorder(regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), eConstraintSet)
totalorder(regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), eEventList)

multicast-req(proc("a"), safe, data(""))
multicast-req(proc("c"), agreed, data(""))
...
```

Final Configuration

```
network(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")))
```

```
operational(proc("a")) operational(proc("b")) operational(proc("c"))
```

```
reachable(proc("a"), sProcSet(proc("a")))
```

```
reachable(proc("b"), sProcSet(proc("b")) sProcSet(proc("c")))
```

```
reachable(proc("c"), sProcSet(proc("b")) sProcSet(proc("c")))
```

```
sent(proc("a"), eMessageSet)
```

```
sent(proc("b"), eMessageSet)
```

```
sent(proc("c"), eMessageSet)
```

```
received(proc("a"), eMessageSet)
```

```
received(proc("b"), eMessageSet)
```

```
received(proc("c"), eMessageSet)
```

```
delivered(proc("a"),
```

```
  sMessageList(msg-data(proc("c"), agreed, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 1, eNatSet, data(""), false))
```

```
  sMessageList(msg-data(proc("a"), safe, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 0, eNatSet, data(""), true))
```

```
  sMessageList(msg-trans(transconf(sProcSet(proc("a")), 1, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))))
```

```
  sMessageList(msg-conf(regconf(sProcSet(proc("a")), 2), sProcSet(proc("a")))))
```

```
delivered(proc("b"),
```

```
  sMessageList(msg-data(proc("c"), agreed, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 1, eNatSet, data(""), false))
```

```
  sMessageList(msg-trans(transconf(sProcSet(proc("b")) sProcSet(proc("c")), 3, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))))
```

```
  sMessageList(msg-data(proc("a"), safe, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 0, eNatSet, data(""), true))
```

```
  sMessageList(msg-conf(regconf(sProcSet(proc("b")) sProcSet(proc("c")), 4), sProcSet(proc("b")) sProcSet(proc("c"))))
```

```
delivered(proc("c"),
```

```
  sMessageList(msg-data(proc("c"), agreed, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 1, eNatSet, data(""), false))
```

```
  sMessageList(msg-trans(transconf(sProcSet(proc("b")) sProcSet(proc("c")), 3, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0))))
```

```
  sMessageList(msg-data(proc("a"), safe, regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), 0, eNatSet, data(""), true))
```

```
  sMessageList(msg-conf(regconf(sProcSet(proc("b")) sProcSet(proc("c")), 4), sProcSet(proc("b")) sProcSet(proc("c"))))
```

```
localconf(proc("a"), regconf(sProcSet(proc("a")), 2))
```

```
localconf(proc("b"), regconf(sProcSet(proc("b")) sProcSet(proc("c")), 4))
```

```
localconf(proc("c"), regconf(sProcSet(proc("b")) sProcSet(proc("c")), 4))
```

```
...
```

Restricting Behavior

- Problem for Analysis:
 - High degree of internal nondeterminism/concurrency
 - Strong dependency on environment (external nondeterminism)

=> State Space Explosion

Our Approach:

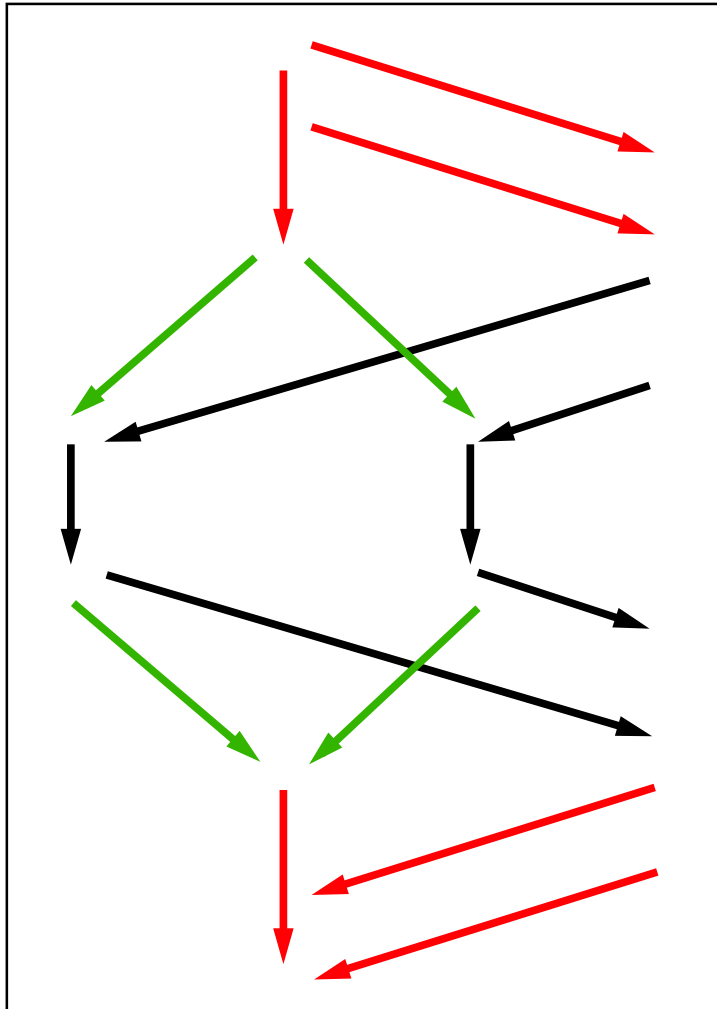
- Restrict behaviour by composing the system with a controller
- Controller is specified using a simple control language
- Semantics of control language is itself specified
in rewriting logic
- Can be used in combination with
Execution, State Space Exploration, and Model Checking

Controller for Previous Scenario

```
CONTROLLER(  
  PERFORM(SEND(proc("a"),0)) ; PERFORM(SEND(proc("c"),1))  
  ;  
  PERFORM(RECEIVE(proc("c"),1)) ; PERFORM(RECEIVE(proc("b"),1));  
  PERFORM(RECEIVE(proc("b"),0)) ; PERFORM(RECEIVE(proc("c"),0)) ;  
  PERFORM(RECEIVE(proc("a"),1)) ; PERFORM(RECEIVE(proc("a"),0)) ;  
  PERFORM(RECEIVE(proc("a"),2)) ; PERFORM(RECEIVE(proc("b"),2)) ;  
  PERFORM(RECEIVE(proc("c"),2)) ; PERFORM(RECEIVE(proc("a"),3)) ;  
  PERFORM(RECEIVE(proc("b"),3)) ; PERFORM(RECEIVE(proc("c"),3)) ;  
  PERFORM(RECEIVE(proc("a"),4))  
  ;  
  PERFORM(LOSE(proc("b"),4)) ; PERFORM(LOSE(proc("c"),4))  
  ;  
  ( PERFORM(CHANGE(proc("a"),sProcSet(proc("a")))) ||  
    PERFORM(CHANGE(proc("b"),(sProcSet(proc("b")) sProcSet(proc("c")))) ||  
    PERFORM(CHANGE(proc("c"),(sProcSet(proc("b")) sProcSet(proc("c")))) )  
  )  
  ;  
  ( PERFORM(EVS-START(proc("a"))) ||  
    PERFORM(EVS-START(proc("b"))) )  
  ;  
  ( PERFORM(EVS-SUCCESS(proc("a"),true)) ||  
    PERFORM(EVS-SUCCESS(proc("b"),true)) ||  
    PERFORM(EVS-SUCCESS(proc("c"),true)) )  
  )  
)
```

Transitional Mode

1. Agree on the set of new members of configurations



```
rl evs-start(proc,newmembers) =>  
  evs-start'(proc,newmembers,newmembers)  
  evs-req(proc,newmembers,newmembers) .
```

```
cr1 evs-req(proc, (members members'), newmembers) =>  
  evs-req(proc, members, newmembers)  
  evs-req(proc, members', newmembers)  
  if members != eProcSet and members' != eProcSet .
```

```
rl evs-start'(proc,newmembers,eProcSet) =>  
  eState .
```

```
cr1 evs-start'(proc,newmembers,(procset procset')) =>  
  evs-start'(proc,newmembers,procset)  
  evs-start'(proc,newmembers,procset')  
  if procset != eProcSet and procset' != eProcSet .
```

```
rl evs-start'(proc,newmembers,sProcSet(proc'))  
  evs-req(proc', sProcSet(proc), newmembers) =>  
  evs-start''(proc,newmembers,sProcSet(proc'))  
  evs-rep(sProcSet(proc), proc') .
```

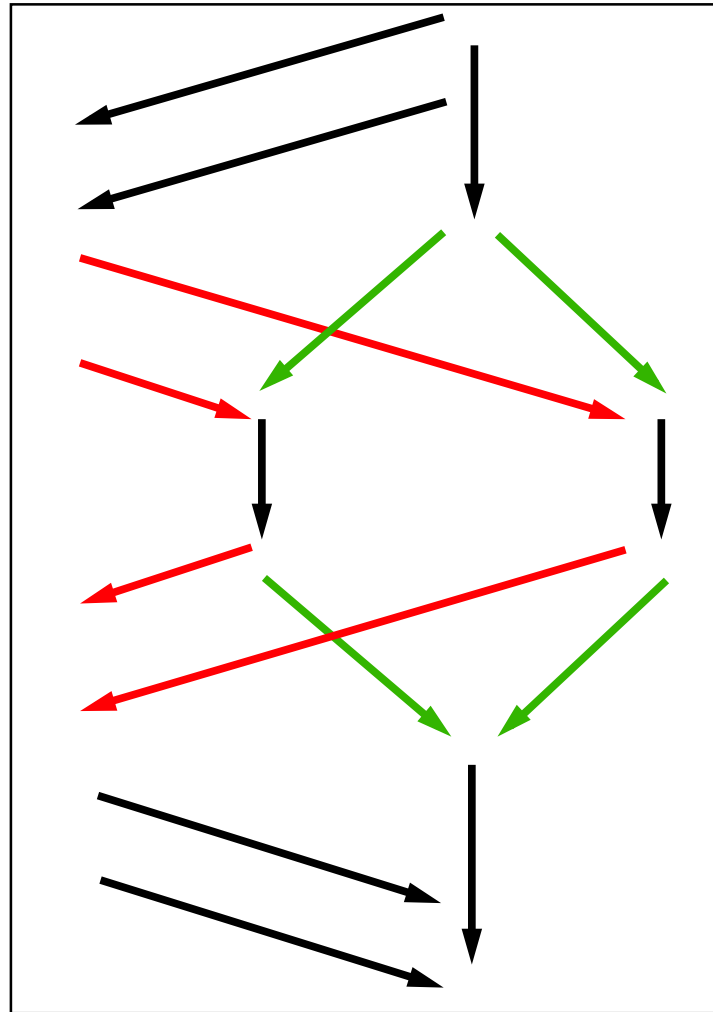
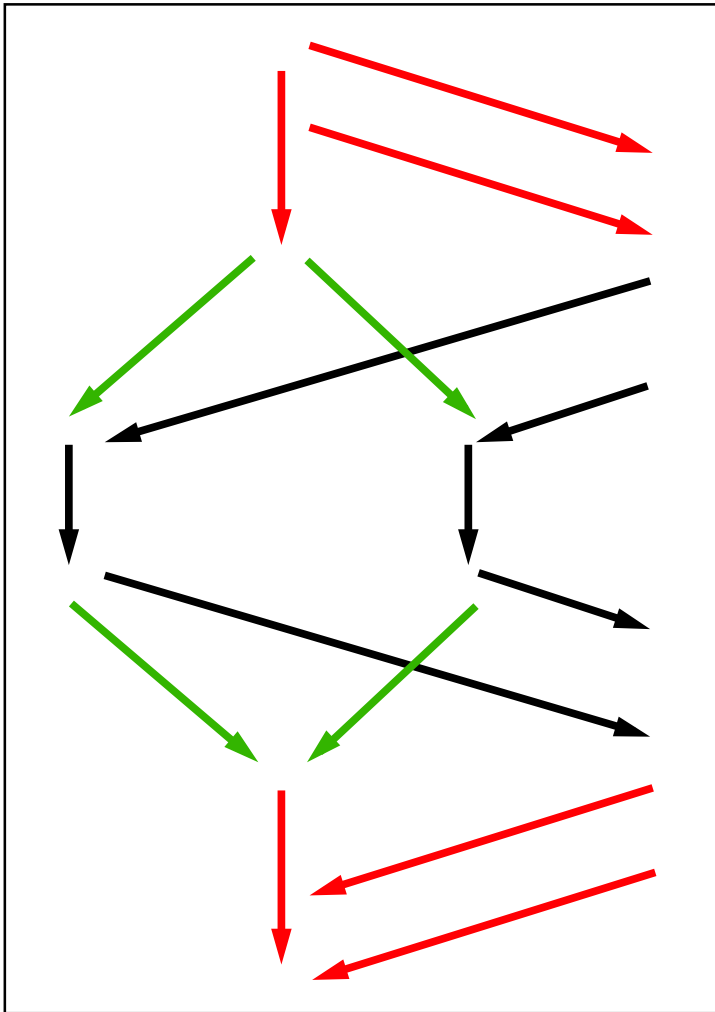
```
rl evs-start''(proc,newmembers,procset)  
  evs-start''(proc,newmembers,procset') =>  
  evs-start''(proc,newmembers,(procset procset')) .
```

```
rl evs-rep(members, proc)  
  evs-rep(members', proc) =>  
  evs-rep(members members', proc) .
```

```
rl evs-start'''(proc,newmembers,newmembers)  
  evs-rep(newmembers,proc) =>  
  evs-start'''(proc,newmembers) .
```

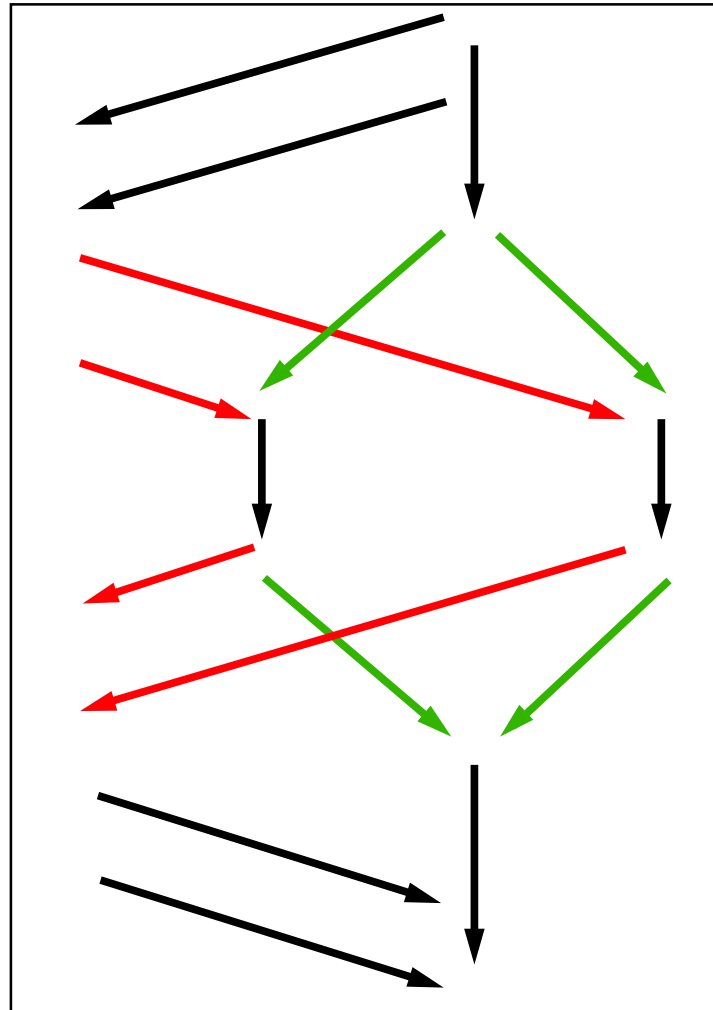
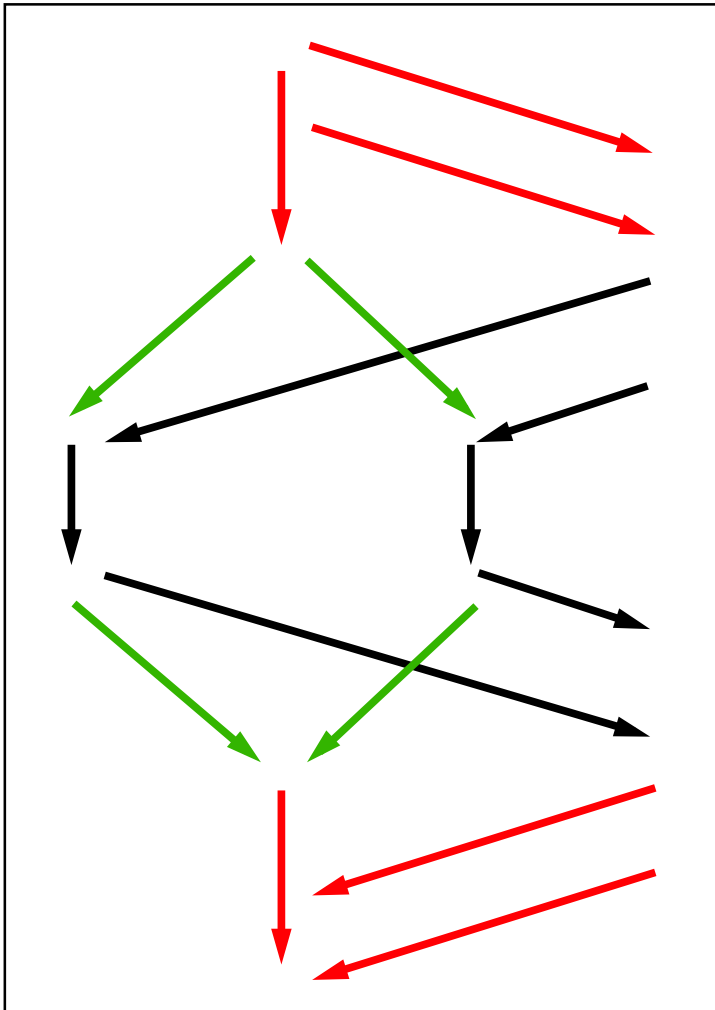
Transitional Mode

1. Agree on the set of new members of configurations



Transitional Mode

2. Determine set of transitional members

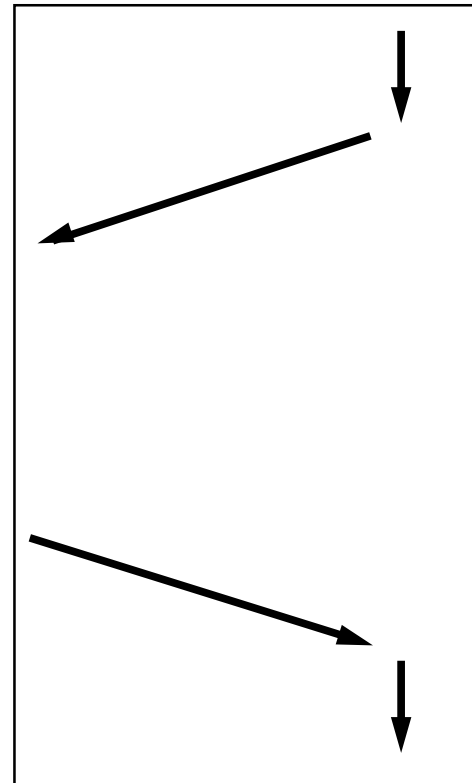
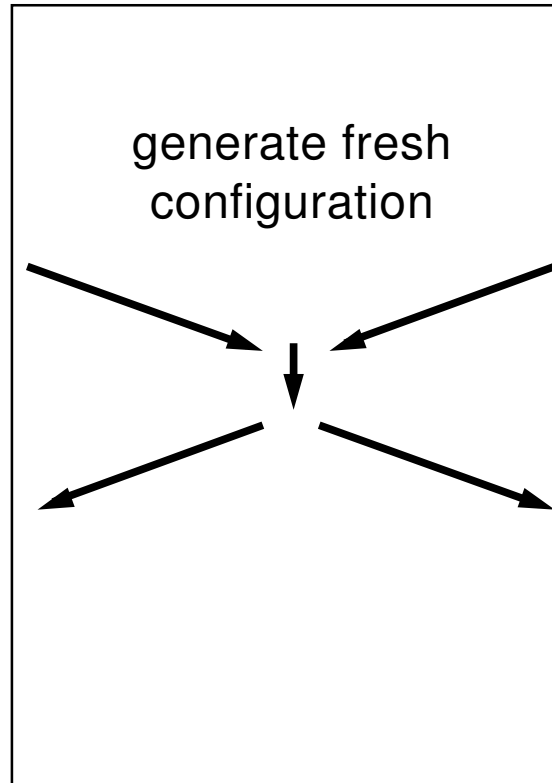
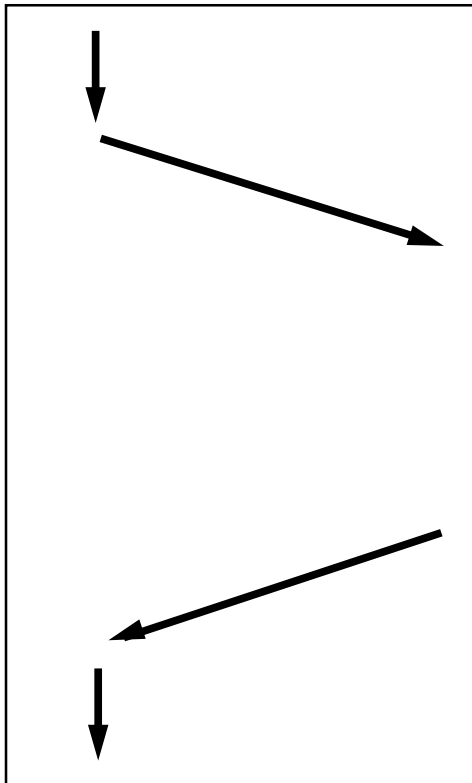


Transitional Mode

3. Deliver all messages that can be delivered



4. Generate and deliver the new transitional configuration

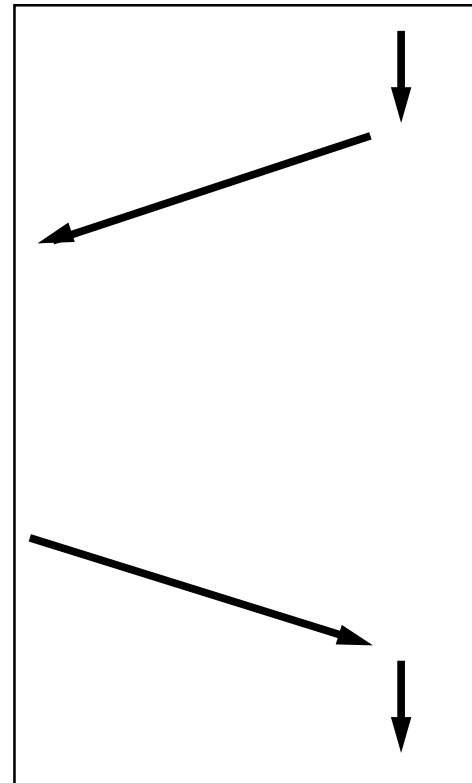
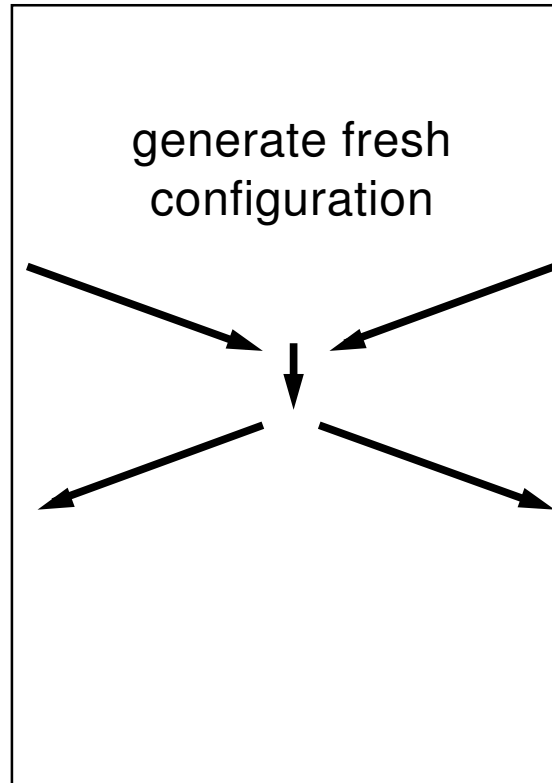
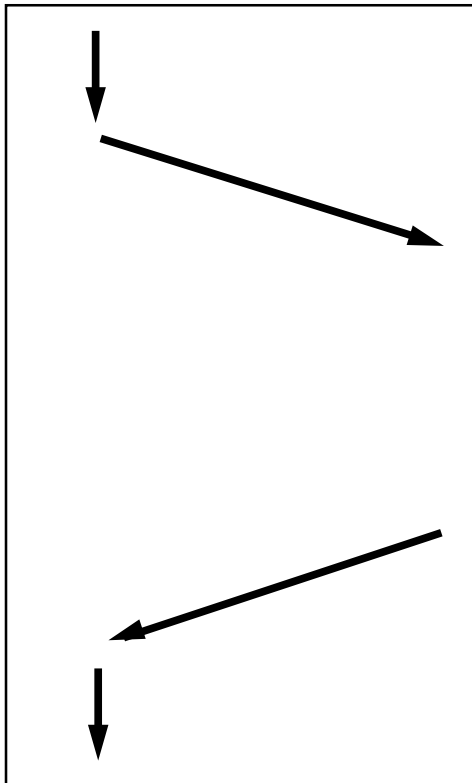


Transitional Mode

5. Deliver all messages that can be delivered



6. Generate and deliver the new regular configuration



Conclusion & Future Work

- Formal specification of group communication middleware is worth the effort because critical and reused by many applications
- Provides unambiguous documentation/clarification of the behavior of implementations in all conceivable circumstances
- High degree of data-driven concurrency and nondeterminism makes rewriting logic a natural choice as a specification language
- Next Objective:
 - Logical group management layer of Spread
 - Replicated database application on top of Spread
- Future Work:
 - Flush Spread on top of Spread
 - Secure Spread on top of Flush Spread
- Beyond Spread:
 - Unifying specification of group communication
- Beyond group communication:
 - A library of reusable, composable communication patterns

