Composable Formal Models for High-Assurance Fault Tolerant Networks

Mark-Oliver Stehr
University of Illinois at Urbana-Champaign

Carolyn Talcott
SRI International

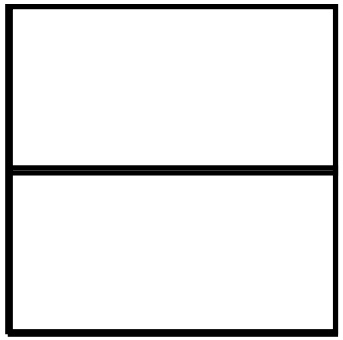
Overview

- The Maude Methodology
- Why Composable Formal Methods ?
- Real-World Case Studies:
  - Group Communication (Spread)
  - Group Key Agreement (Cliques)
  - Secure Group Communication (Secure Spread)
- Flavors of Composition
- Conclusion and Future Work

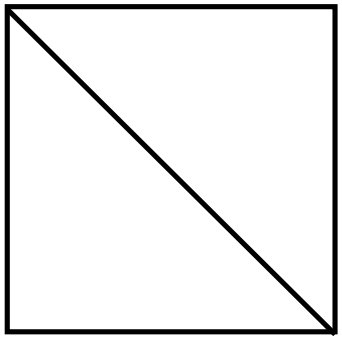
The Maude Methodology

- 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

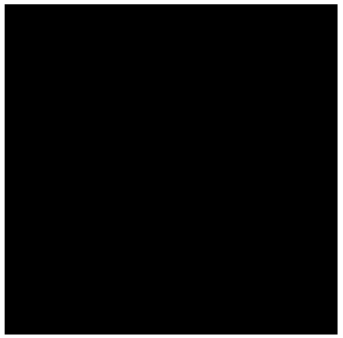
Specification vs. Implementation



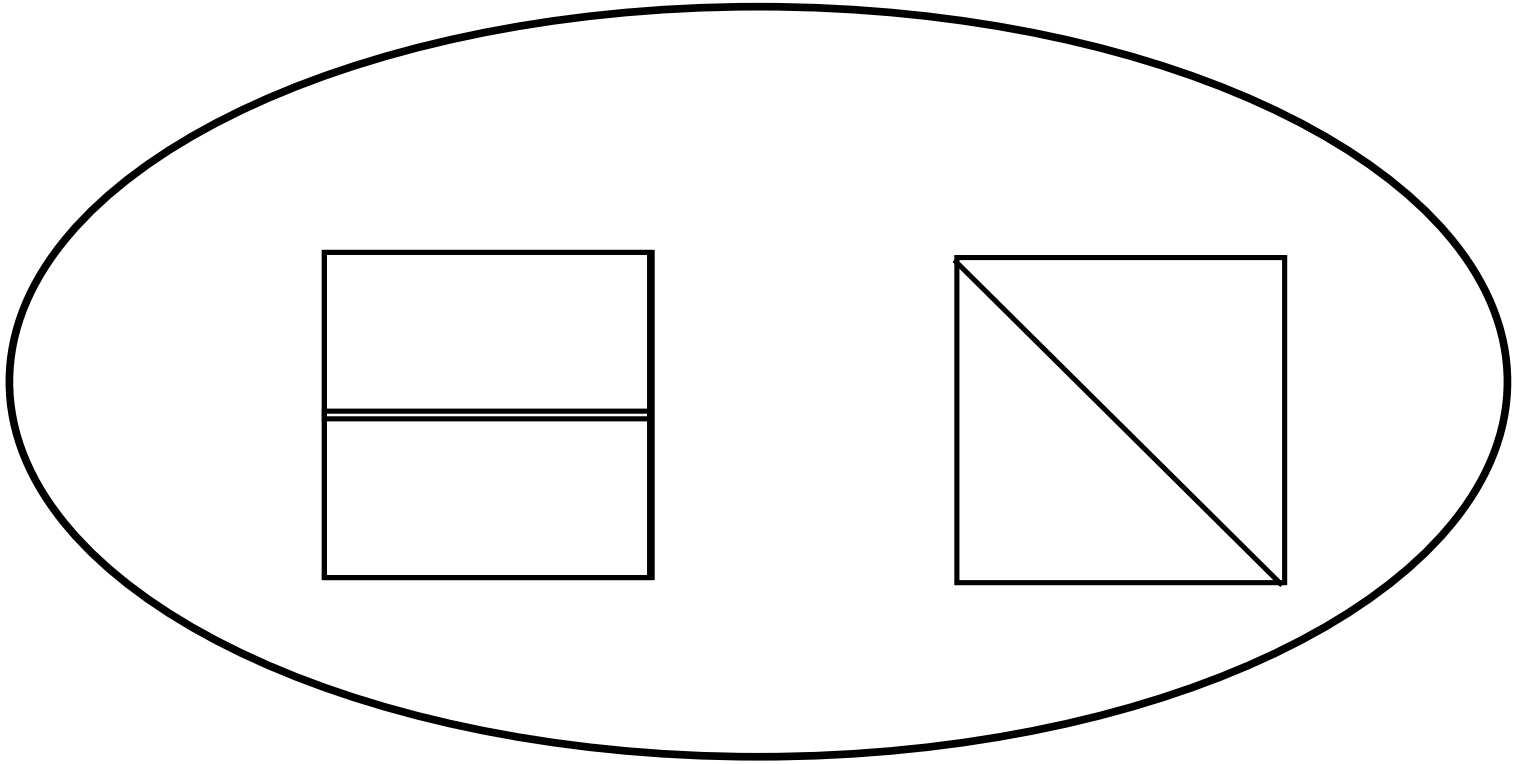
and



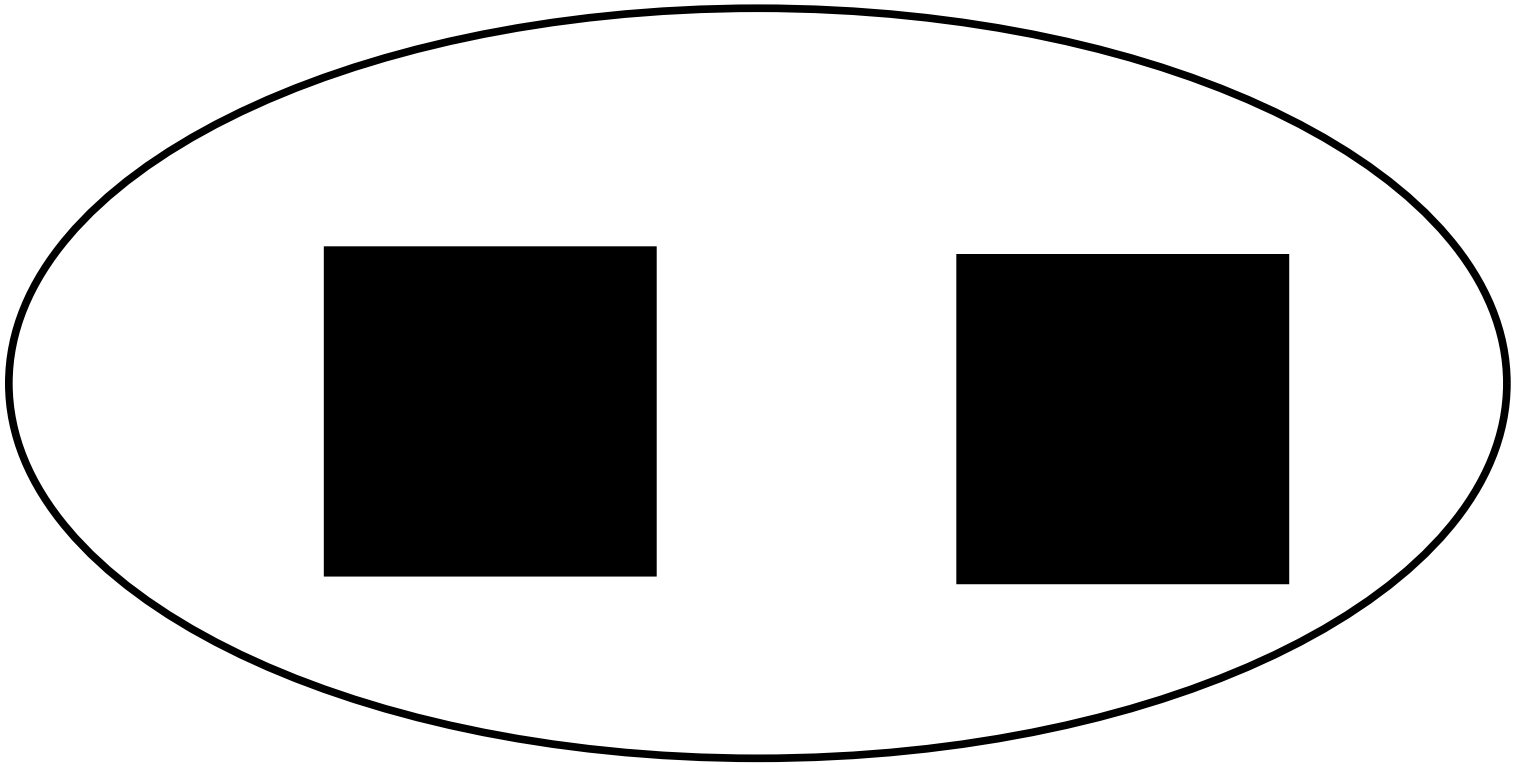
are implementations of the specification



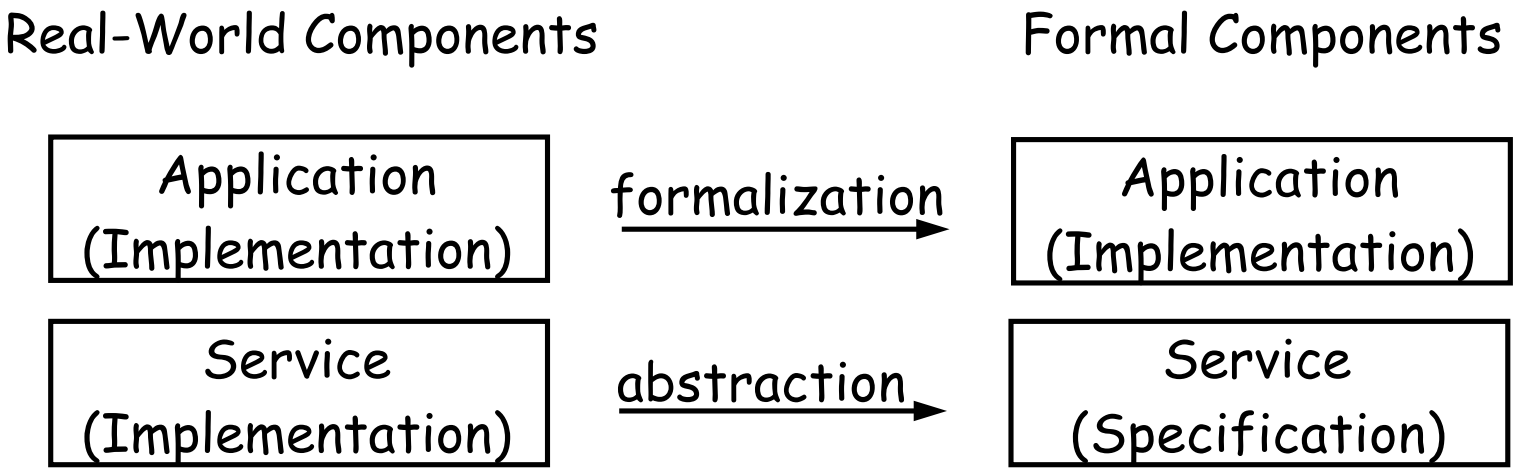
Composition



Abstraction



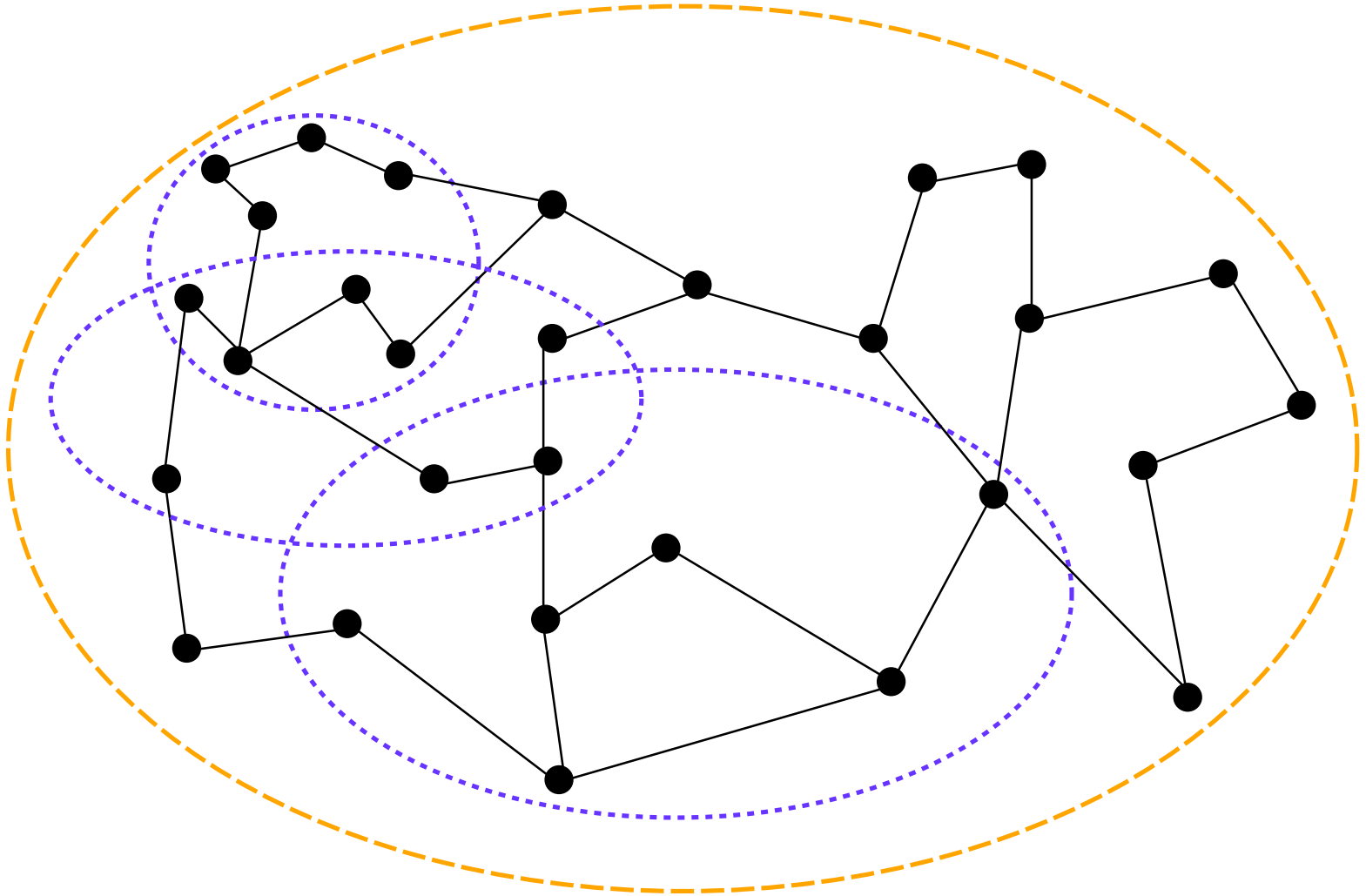
Composable Formal Methods



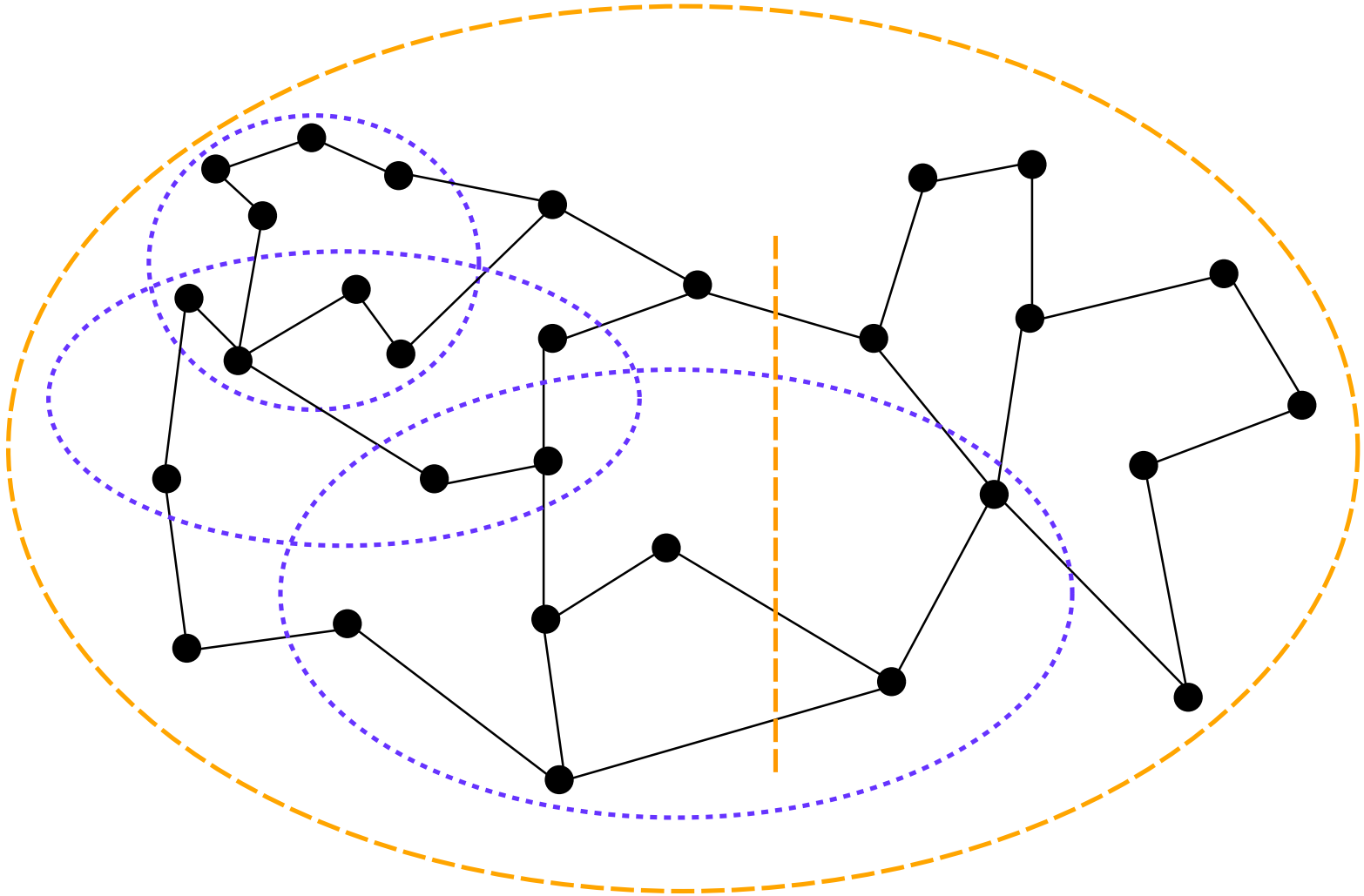
Group Communication: Spread

- Developed by Yair Amir and Jonathan Stanton
  (Johns Hopkins University)
- Evolved from Transis and Totem
- Allows Network Partitioning and Merging
- Extended Virtual Synchrony Semantics
- 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)

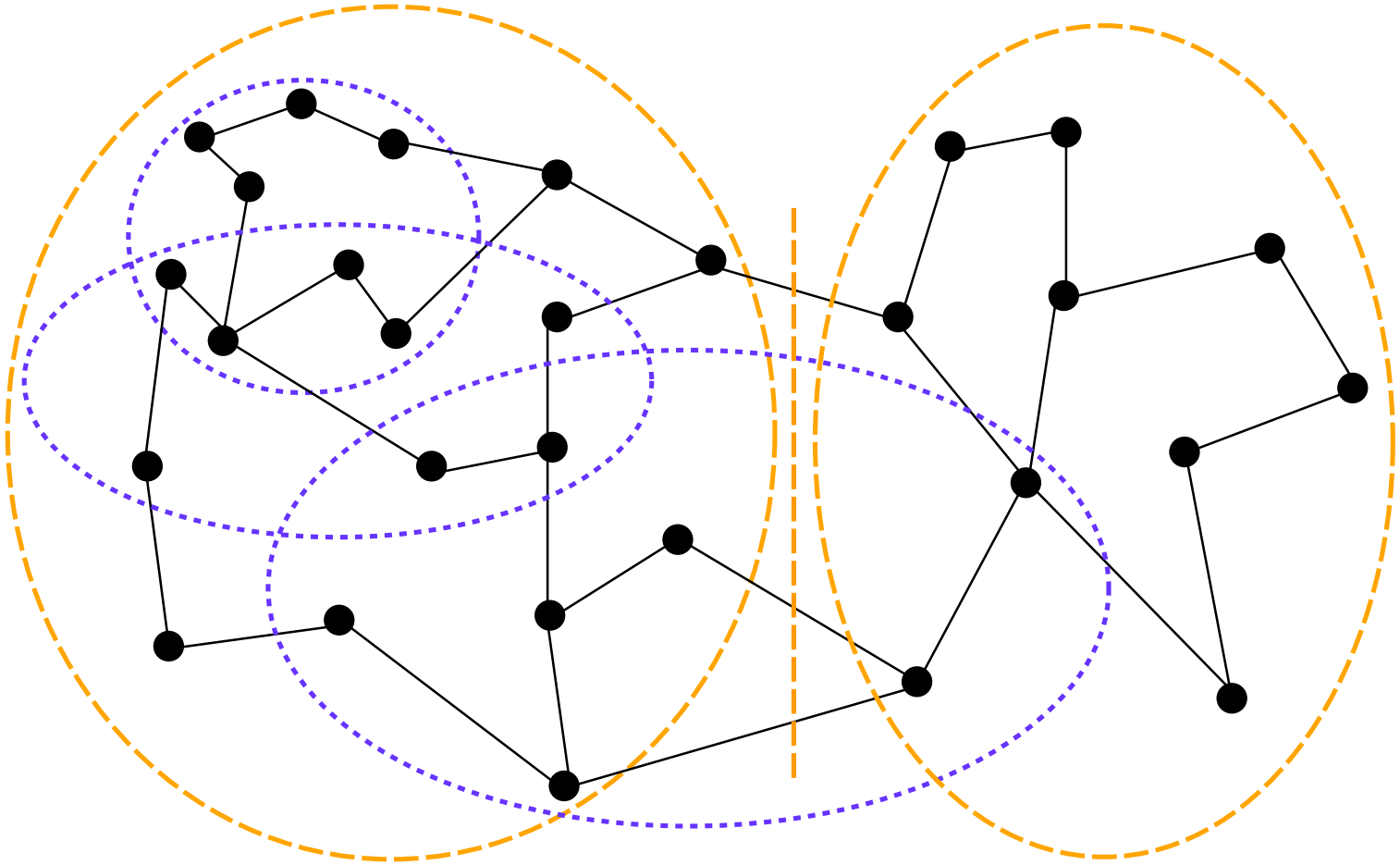
Configurations & Groups



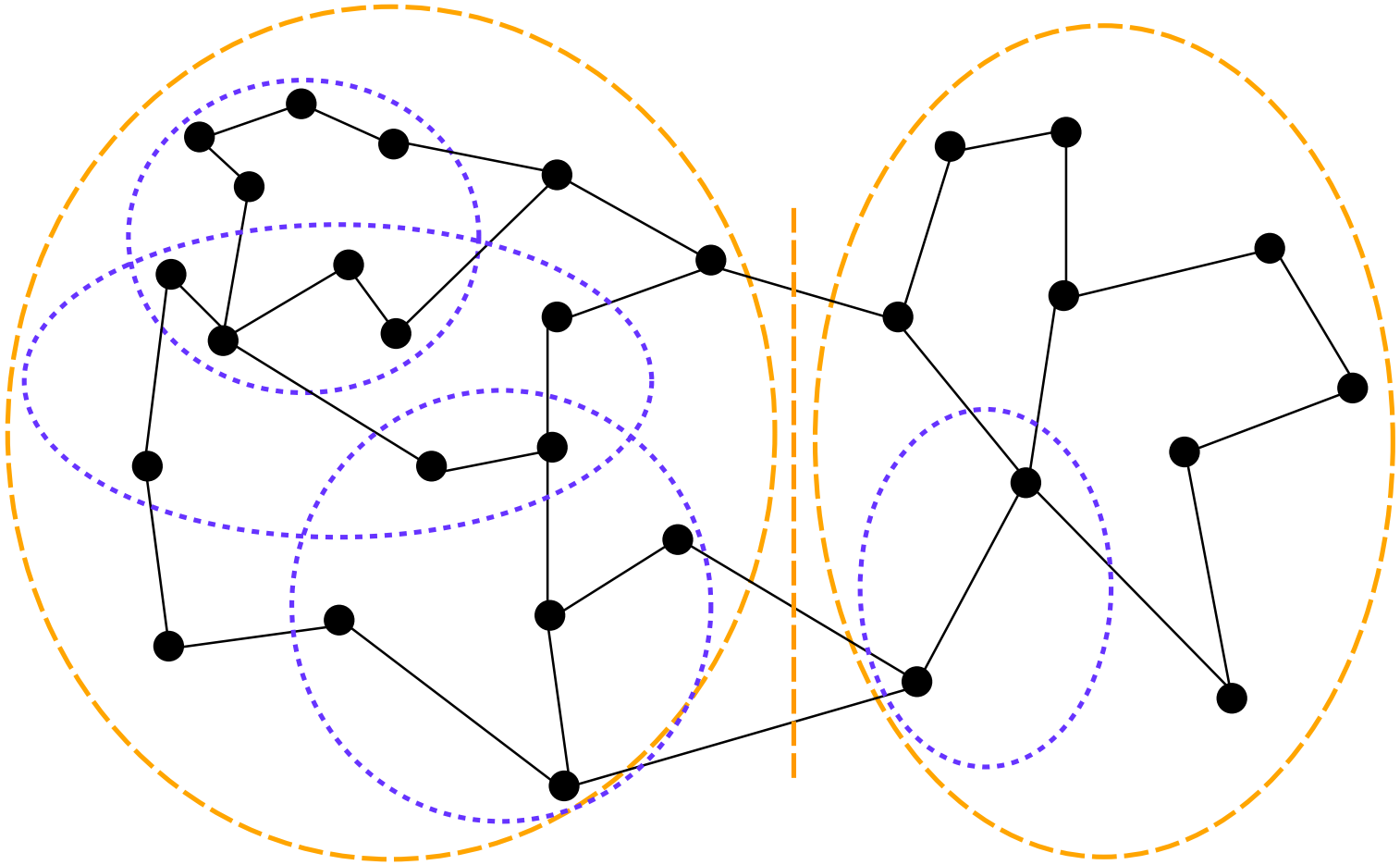
Configurations & Groups



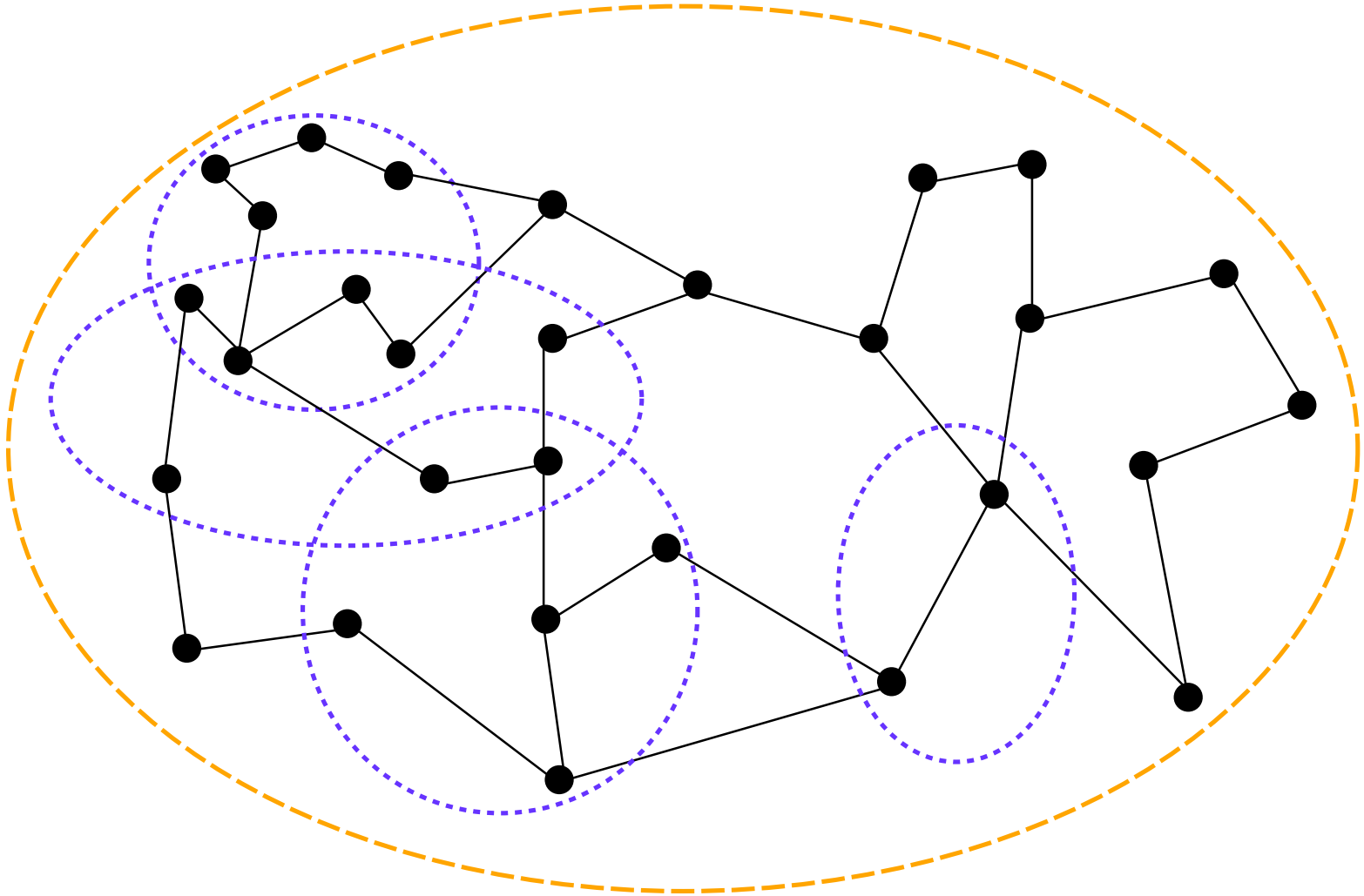
Configurations & Groups



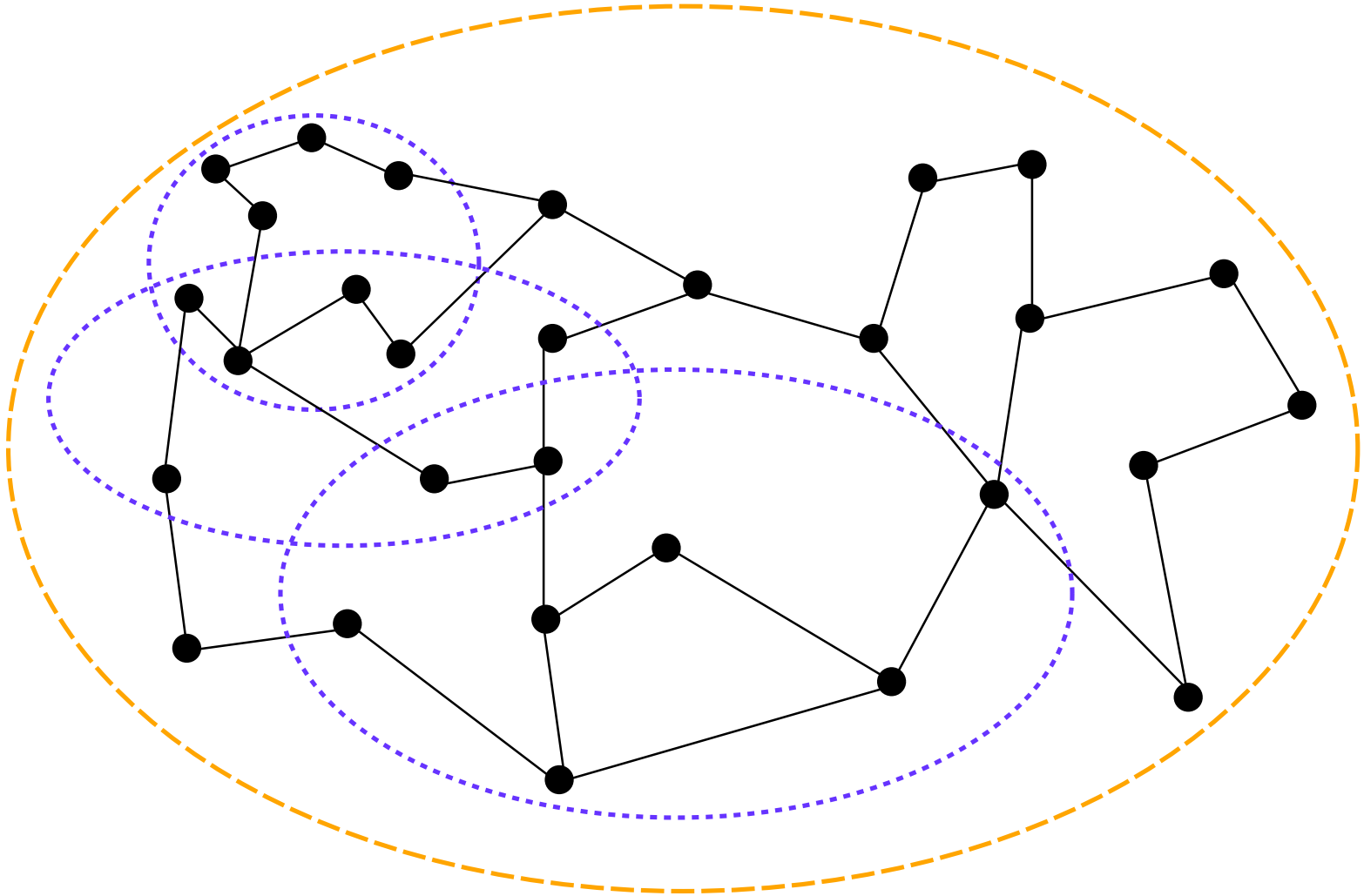
Configurations & Groups



Configurations & Groups



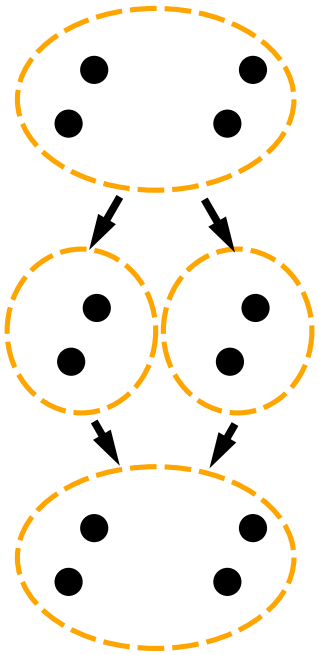
Configurations & Groups



Partial Order of Configurations

split

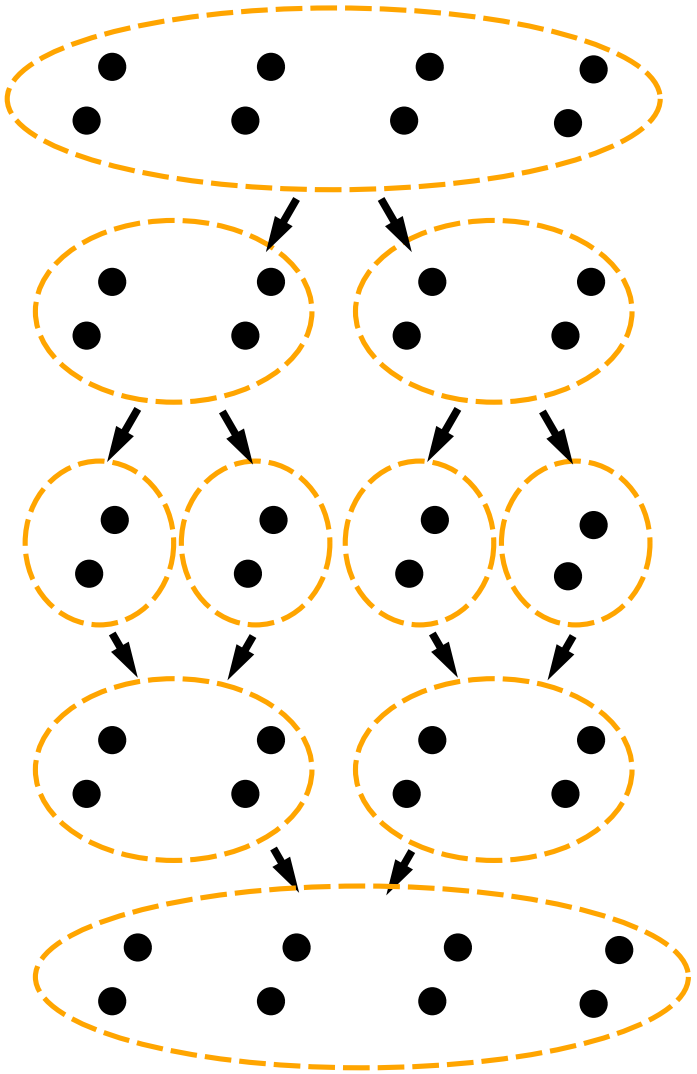
merge



Partial Order of Configurations

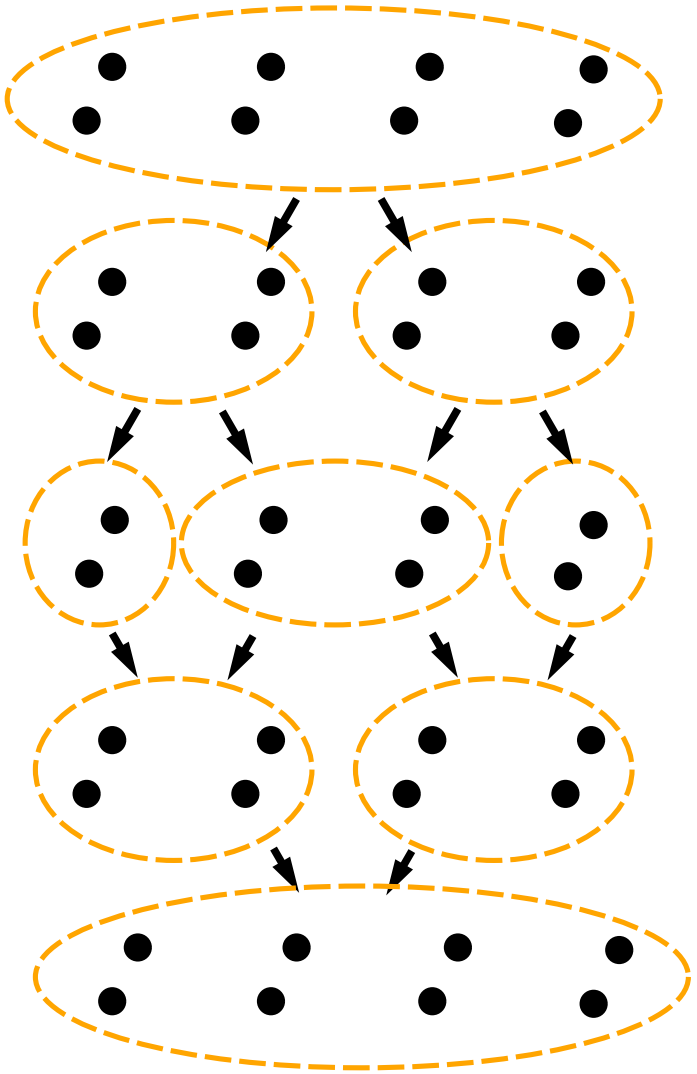
repeated split

repeated merge



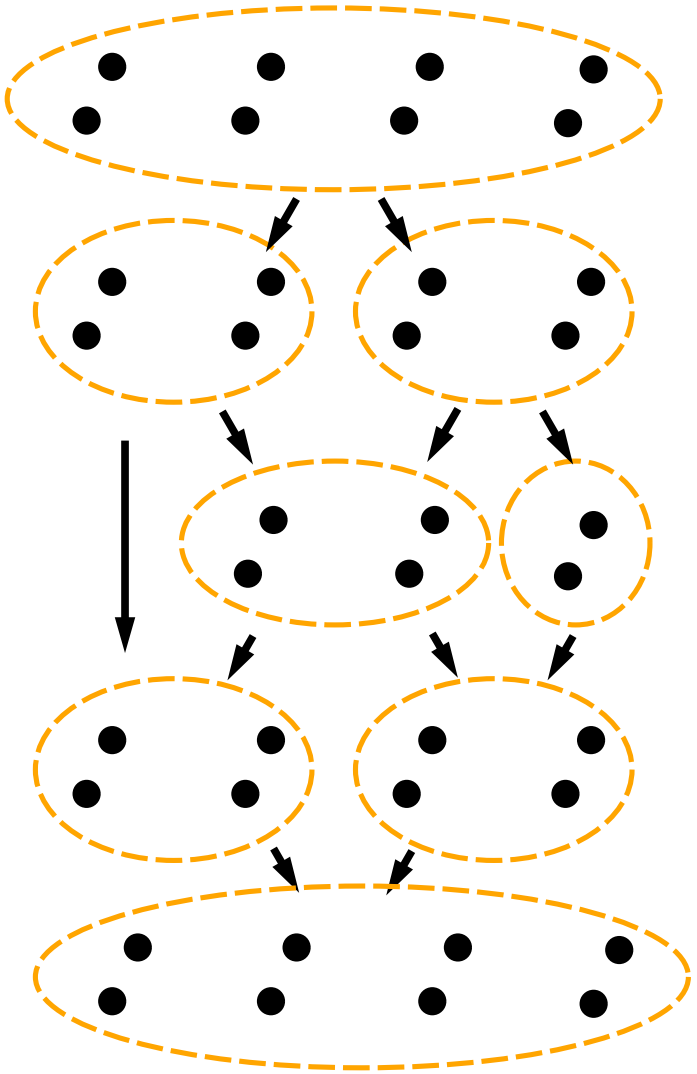
Partial Order of Configurations

concurrent
split & merge



Partial Order of Configurations

asymmetric behavior



Towards a Formal Model

- Objective:
  - Abstract specification of Spread
  - Very different from Implementation:
    Less complex but more general behaviour
- References:
  - User Guide (informal, many technical details)
  - Research Papers (high-level axiomatic semantics)
  - Spread Source Code (C)
- Use of abstract specification:
  - Supplementary Documentation (for designers and users)
  - Verification of Spread itself
  - Verification of layers and applications on top of Spread
- Challenge: Capture best-effort principle formally
  - Idea: Explicit representation of all delivery constraints
  - Deliver as much as possible under the given constraints

Spread API Overiew

Messages

ops reliable fifo causal agreed safe : -> Mode .

op msg-data : Agent Group Data Mode -> Message .
op msg-trans : Group -> Message .
op msg-join : Agent Group AgentList Membership -> Message .
op msg-leave : Agent Group AgentList Membership -> Message .
op msg-disconnect : Agent Group AgentList Membership -> Message .
op msg-network : Group Membership AgentList AgentList -> Message .
op msg-self-leave : Group -> Message .

Join

op sp-join-req : Agent Group -> Configuration .
op sp-join-ack : Agent -> Configuration .
op sp-join-err : Agent JoinError -> Configuration .

Leave

op sp-leave-req : Agent Group -> Configuration .
op sp-leave-ack : Agent -> Configuration .
op sp-leave-err : Agent LeaveError -> Configuration .

Multicast

op sp-multicast-req : Agent Group Data Mode -> Configuration .
op sp-multicast-ack : Agent -> Configuration .
op sp-multicast-err : Agent MulticastError -> Configuration .

Receive

op sp-receive-req : Agent -> Configuration .
op sp-receive-ack : Agent Message -> Configuration .
op sp-receive-err : Agent ReceiveError -> Configuration .

Miscellaneous

op sp-connect-req : Agent -> Configuration .
op sp-connect-ack : Agent -> Configuration .
op sp-connect-err : Agent ConnectError -> Configuration .

op sp-disconnect-req : Agent -> Configuration .
op sp-disconnect-ack : Agent -> Configuration .
op sp-disconnect-err : Agent DisconnectError -> Configuration .

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)  .
```

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))

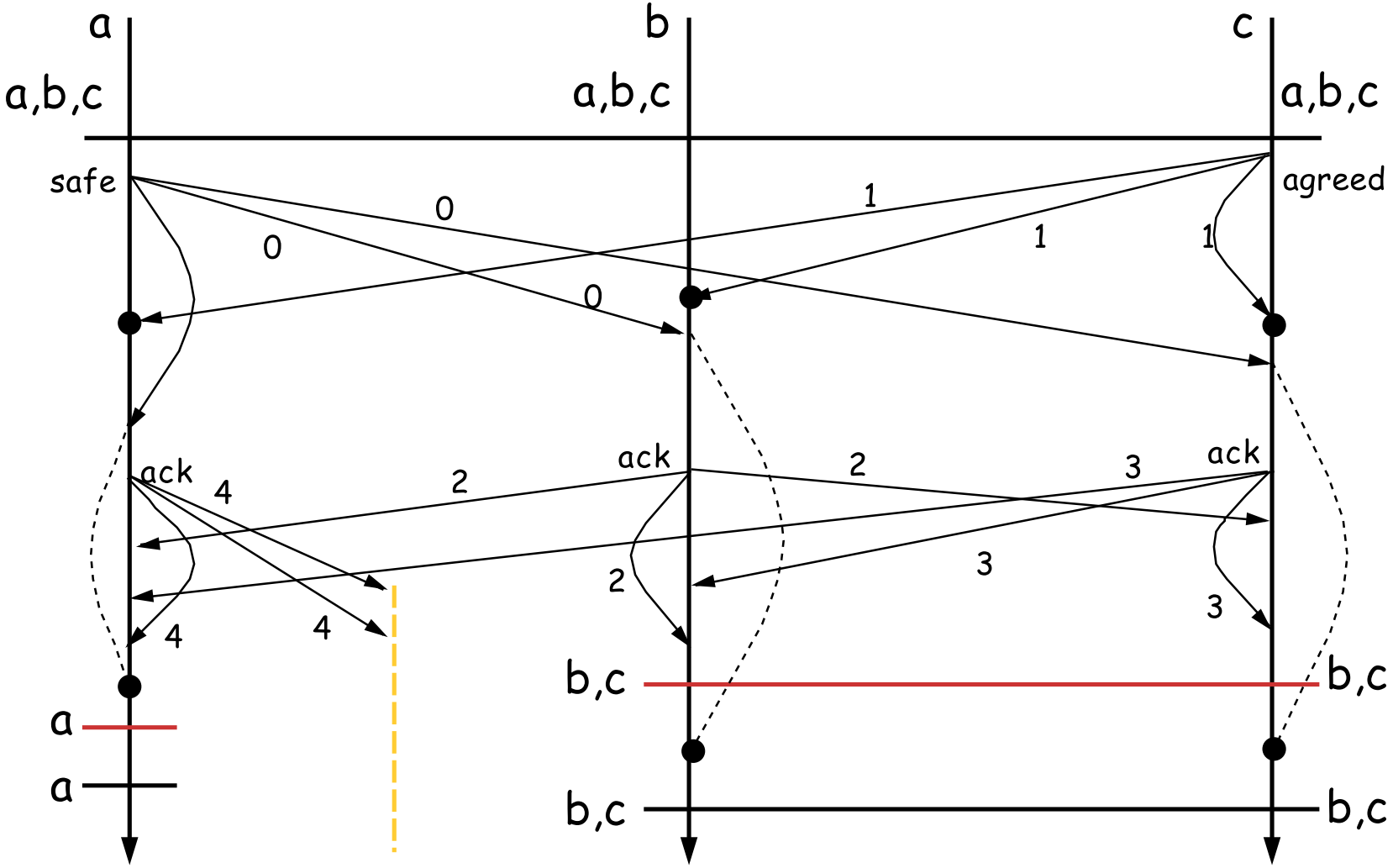
causalorder(regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), eConstraintSet)
totalorder(regconf(sProcSet(proc("a")) sProcSet(proc("b")) sProcSet(proc("c")), 0), eEventList)

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))

...
```

Symbolic Execution



Group Key Agreement: Cliques

- Objective:
  - Agree on shared secret in a peer group
  - Update shared secret if membership changes
- Existing Group Key Agreement Protocols:
  - Centralized Key Distribution
  - Burmester-Desmedt, 1994
  - Steer-Strawczynski-Diffie-Wiener, 1988
  - **Steiner-Tsudik-Waidner (Group Diffie-Hellman), 1996**
  - Kim-Perrig-Tsudik (Tree-Based Group Diffie-Hellman), 2000
- Cliques-Toolkit:
  - Implements all the above protocols
  - API independent of protocol and underlying GCS

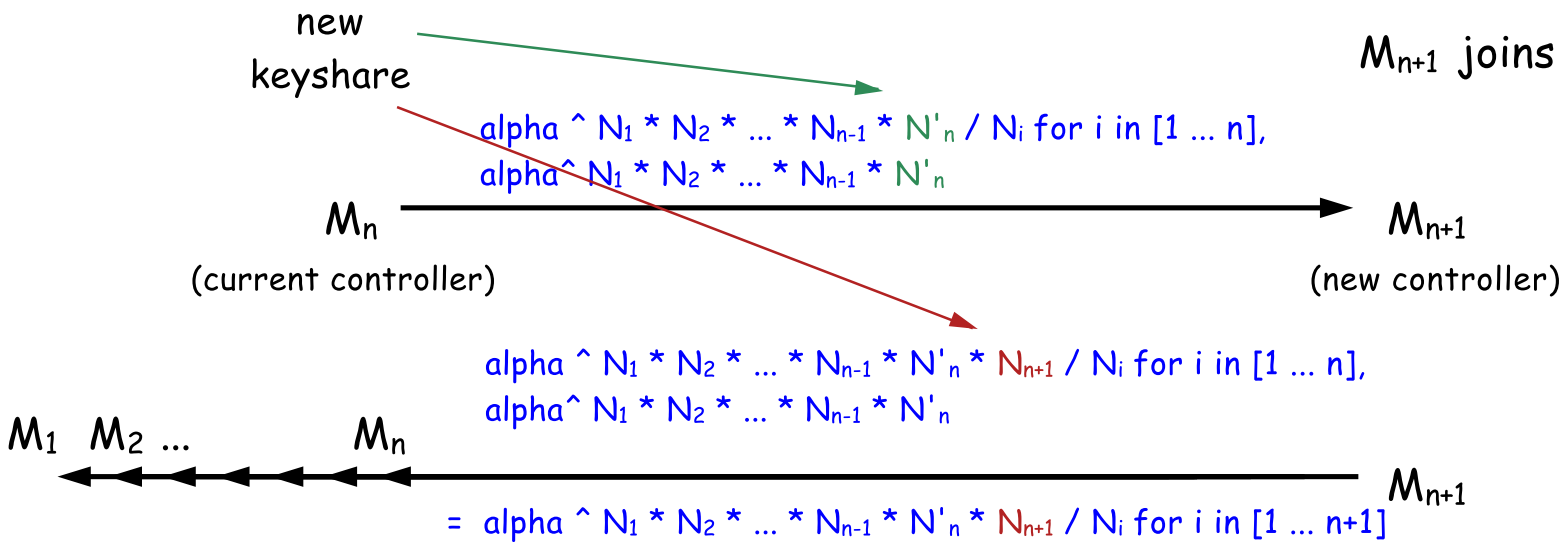
Towards a Formal Model

- Objective:
  - Abstract specification of Cliques toolkit
  - Focussing on Group Diffie-Hellman protocol
- Reference:
  - Research Papers (merge and leave protocols not covered)
  - Cliques Documentation (partly outdated, but useful source)
  - Cliques Source Code (C)
- Use of abstract specification:
  - Supplementary documentation
  - Verification of Cliques itself
  - Verification of layers and applications on top of Cliques
- Challenges:
  - Capture generic character of API
  - Abstract Specification of Cryptographic Primitives
  - Formal model obtained directly from the C code

Join Protocol

group $M_1 \ldots M_n$ with group key: alpha ^ $N_1 * N_2 * \ldots * N_{n-1} * N'_n$ for i in [1 ... n+1]

each member M_i knows: alpha ^ $N_1 * N_2 * \ldots * N_{n-1} * N_n / N_i$ for i in [1 ... n]



each member M_i knows: alpha ^ $N_1 * N_2 * \ldots * N_{n-1} * N'_n * N_{n+1} / N_i$ for i in [1 ... n+1]

each member M_i factors in its keyshare to obtain group key:

alpha ^ $N_1 * N_2 * \ldots * N_{n-1} * N'_n * N_{n+1}$ for i in [1 ... n+1]

Cyclic Groups

```
sort PartialKey .
op idPartialKey : -> PartialKey .
op _*_ : PartialKey PartialKey -> PartialKey
         [assoc comm id: idPartialKey] .
op inv : PartialKey -> PartialKey .
eq inv(x) * x = idPartialKey .
```

group of (partial) keys
(group keys)

```
op alpha : -> PartialKey .
```

group generator
represented as constant

```
sort KeyShare .
op idKeyShare : -> KeyShare .
op _*_ : KeyShare KeyShare -> KeyShare
         [assoc comm id: idKeyShare] .
op inv : KeyShare -> KeyShare .
eq inv(y) * y = idKeyShare .
```

group of key contributions
(key shares)

```
op _^_ : PartialKey KeyShare -> PartialKey .
eq  x ^ idKeyShare = x .
eq (x ^ y) ^ z = x ^ (y * z) .
```

exponential normal form
alpha ^ N_1 * N_2 * ... * N_{n-1} * N_n

```
op random : Nat -> KeyShare .
```

random keyshares
represented as constants

Cliques API Overiew

Single Member Join

op clq-proc-join-req : Agent Group Context Agent -> State .
op clq-proc-join-ack : Agent Group Context Token -> State .

op clq-join-req : Agent Group Token -> State .
op clq-join-ack : Agent Group Context Token -> State .

op clq-update-ctx-req : Agent Group Context Token -> State .
op clq-update-ctx-ack : Agent Group Context -> State .

Leave Operation

op clq-leave-req : Agent Group Context AgentList -> State .
op clq-leave-ack-controller : Agent Group Context Token -> State .
op clq-leave-ack-not-controller : Agent Group Context -> State .
op clq-leave-ack-not-member : Agent Group -> State .

op clq-refresh-key-req : Agent Group Context Token -> State .
op clq-refresh-key-ack : Agent Group Context -> State .

Merge Operation

op clq-update-key-first-req : Agent Group Context AgentList -> State .
op clq-update-key-first-ack : Agent Group Context Token -> State .
op clq-update-key-intermediate-req : Agent Group Context Token -> State .
op clq-update-key-intermediate-ack : Agent Group Context Token -> State .
op clq-update-key-last-req : Agent Group Context Token -> State .
op clq-update-key-last-ack : Agent Group Context Token -> State .

op clq-factor-out-req : Agent Group Context Token -> State .
op clq-factor-out-ack : Agent Group Context Token -> State .
op clq-factor-out-ack : Agent Group Context -> State .

op clq-merge-req : Agent Group Context Agent Token AgentSet -> State .
op clq-merge-ack : Agent Group Context AgentSet -> State .
op clq-merge-ack : Agent Group Context AgentSet Token -> State .

Other Operations

op clq-first-user-req : Agent Group -> State .
op clq-first-user-ack : Agent Group Context -> State .

op clq-new-user-req : Agent Group -> State .
op clq-new-user-ack : Agent Group Context -> State .

Symbolic Execution

```
rew fresh(0)
   start(agent("a"), group("G"))
   join(agent("b"), ...,  group("G"))
   join(agent("c"), ...,  group("G"))
   join(agent("d"), ...,  group("G"))
   leave(sAgentList(agent("d")), ..., group("G")) .

rewrites: 3508 in 20ms cpu (175400 rewrites/second)
result State: fresh(7)
ready(agent("a"), group("G"),
 context(random(0), alpha ^ random(0)*random(2)*random(5)*random(6), ...)
ready(agent("b"), group("G"),
 context(random(2), alpha ^ random(0)*random(2)*random(5)*random(6), ...)
ready(agent("c"), group("G"),
 context(random(6), alpha ^ random(0)*random(2)*random(5)*random(6), ...)
```

State Space Exploration

```
search fresh(0)
        start(agent("a"), group("G"))
        merge(sAgentList(agent("b")) sAgentList(agent("c")), ..., group("G"))
        =>!
        state:State .

Solution 1 (state 260645)
states: 260646  rewrites: 27859114 in 524600ms cpu (53105 rewrites/second)
state:State --> fresh(6)
ready(agent("a"), group("G"),
  context(random(0), alpha ^ random(0) * random(2) * random(4),  ...)
ready(agent("b"), group("G"),
  context(random(2), alpha ^ random(0) * random(2) * random(4), ...)
ready(agent("c"), group("G"),
  context(random(4), alpha ^ random(0) * random(2) * random(4), ...)

No more solutions.
states: 260646  rewrites: 27859114 in 524600ms cpu (53105 rewrites/second)
```

Modelchecking

```
op done : -> Prop .
ceq fresh(var)
      ready(agent("a"), group("G"), contexta, memberlista)
      ready(agent("b"), group("G"), contextb, memberlistb)
      ready(agent("c"), group("G"), contextc, memberlistc)
      |=
      done = true
      if groupsecret(contexta) == groupsecret(contextb) /\
        groupsecret(contextb) == groupsecret(contextc) .

op initial : -> State .
eq initial = fresh(0)
       start(agent("a"),group("G"))
       merge(sAgentList(agent("b")) sAgentList(agent("c")), ..., group("G")) .

red modelCheck(initial, <> done) .
rewrites: 27753469 in 176690ms cpu (157074 rewrites/second)
result Bool: true
```

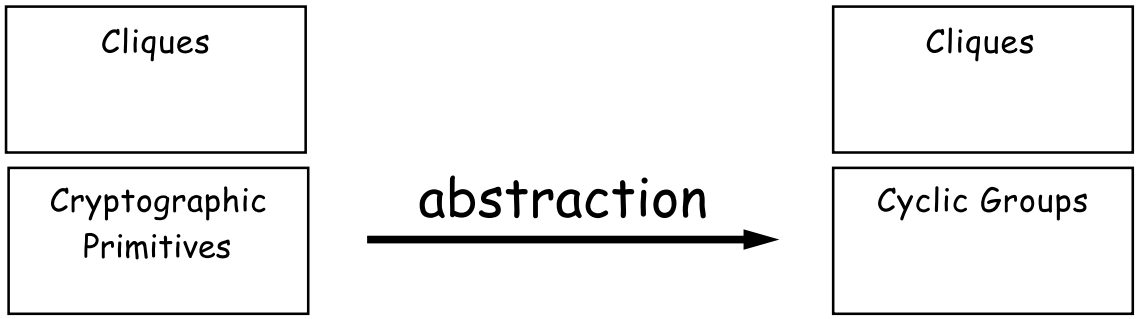
Flavors of Composition

Functional Composition:

- instead of composing the GDH specification
  with an abstract specification of cyclic groups

it can be composed
  with a concrete implementation of cryptographic primitives,
  while preserving all operational and logical properties

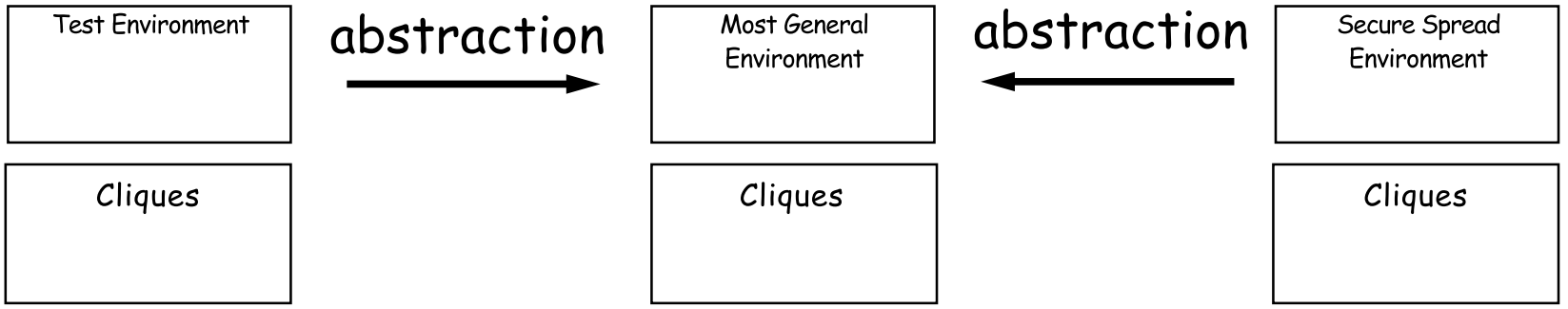
- interface is purely functional
- mathematically elegant and easy to handle



Flavors of Composition

Composition of Services:

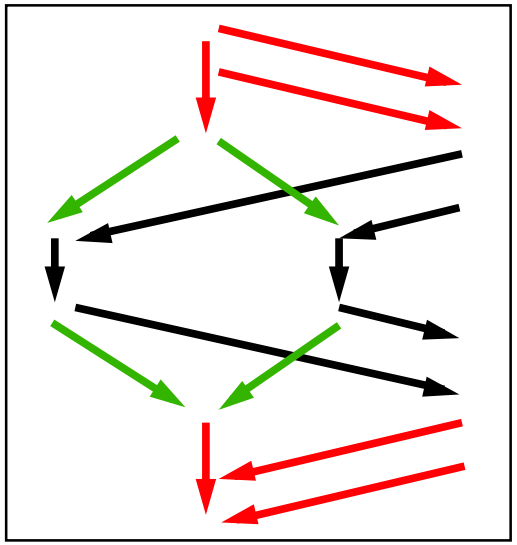
- Services of the Secure-Spread Layer are implemented using
  - the Spread Layers and
  - the Cliques-Toolkit
- interface consists of messages (typically request/reply pairs)
- allows concurrency and maintainence of state
- emphasizes the service provided by another component
- loosely coupled, asymmetric, coarse-grained

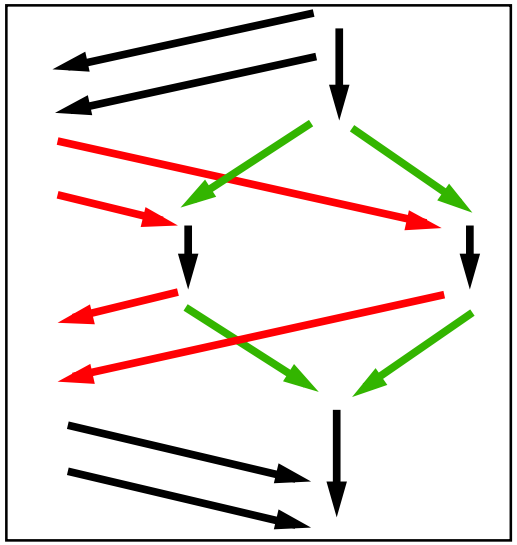


Flavors of Composition

Composition of Patterns:

- The Spread EVS algorithm can be specified
  as a composition of a few basic interaction patterns
- interface consists of tokens (synchronization messages)
- allows concurrency and complex patterns
- emphasizes the synchronization/coordination structure
- tighly coupled, symmetric, fine-grained





Conclusion

- Formal specification is a complex enterprise because:
  it must cover the behavior
    - of all conceivable implementations
    - under all conceivable circumstances

  and is based on informal documentation.

- But in this case it is worth the effort because:
  - middleware is critical and reused by many applications,
  - it forces us to make explict the essential ideas and
  - hence leads to a better understanding,
  - it provides unambigious documentation for designers and for users,
  - it is a reference for mathematical/formal verification.

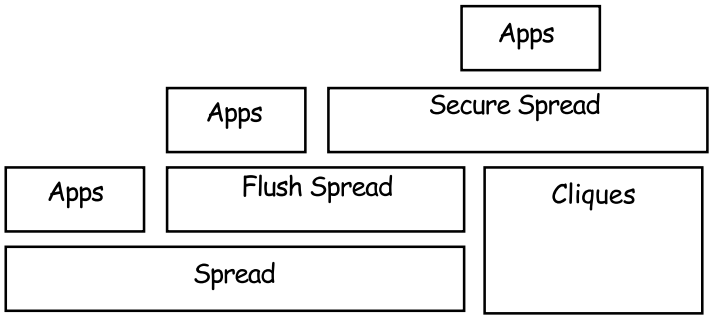
Conclusion

- Rewriting logic and its implementation in Maude is a meta-technology which is especially powerful in applications with a high degree of concurrency and nondeterminism.

- The executability of rewriting logic and its implementation in Maude
  - speeds up the development of formal specifications
  - and increases their quality by early validation

- Compositional approach combined with executable abstractions allows scaling up light-weight methods to complex systems

Future Work



- Logical group management layer of Spread
- Flush Spread on top of Spread
- Secure Spread on top of Flush Spread/Cliques
- Cliques: Authentication with Digital Signatures
- Applications