Programming Wireless Sensor Networks in a Self-Stabilizing Style

C. Weyer[1] V. Turau[1] A. Lagemann[2] J. Nolte[2]

[1]Institute of Telematics
Hamburg University of Technology

[2]Distributed Systems/Operating Systems
Brandenburg University of Technology Cottbus

The Third International Conference on Sensor Technologies and Applications, 2009

Outline

Motivation

Short Introduction to Self-Stabilization

Self-Stabilization in WSN

Main Results

Self-Stabilization

Short Communications
Operating Systems

Self-stabilizing Systems in Spite of Distributed Control

Edsger W. Dijkstra
Burroughs Corporation

Key Words and Phrases: multiprocessing, networks, self-stabilization, synchronization, mutual exclusion, robustness, sharing, error recovery, distributed control, harmonious cooperation, self-repair
CR Categories: 4.32

The synchronization task between loosely coupled cyclic sequential processes (as can be distinguished in, for instance, operating systems) can be viewed as keeping the relation "the system is in a legitimate state" invariant. As a result, each individual process step that could possibly cause violation of that relation has to be preceded by a test deciding whether the process in question is allowed to proceed or has to be delayed. The resulting design is readily—and quite systematically—implemented if the different processes can be granted mutually exclusive access to a common store in which "the current system state" is recorded.

A complication arises if there is no such commonly accessible store and, therefore, "the current system state" must be recorded in variables distributed over the various processes; and a further complication arises if the communication facilities are limited in the sense that each process can only exchange information with "its neighbors," i.e. a small subset of the total set of processes. The complication is that the behavior of a process can only be influenced by that part of the total current system state description that is available to it; local actions taken on account of local information must accomplish a global objective. Such systems (with what is quite aptly called "distributed control") have been designed, but all such designs I was familiar with were not "self-stabilizing" in the sense that, when once (erroneously) in an illegitimate state, they could—and usually did!—remain so forever. Whether the property of self self-stabilization—for a more precise definition, see below—is interesting as a starting procedure, for the sake of robustness or merely as an intriguing problem, falls outside the scope of this article. It could be of relevance on a scale ranging from a worldwide network to common bus control. (I have been told that the first solution shown below was used a few weeks after its discovery in a system where two resource-sharing computers were coupled via a rather primitive channel along which they had to arrange their cooperation.)

We consider a connected graph in which the majority of the possible edges are missing and a finite state machine is placed at each node; machines placed in
directly conne
bors. For eac
leges" are de
state and th
boolean func
"present." I
ratios of the
daemon—its
outside the sc
of the privil
selected privi
brought into
state and th
machine mor
state may als
completion o
privilege.

Furtherm
whether the s
We require th
privileges wi
each possibl
legitimate st
at least one
legitimate states there exists a sequence of moves transferring the system from the one into the other.

We call the system "self-stabilizing" if and only if, regardless of the initial state and regardless of the privilege selected each time for the next move, at least one privilege will always be present and the system is guaranteed to find itself in a legitimate state after a finite number of moves. For more than a year—at least to my knowledge—it has been an open question whether nontrivial (e.g. all states legitimate is considered trivial) self-stabilizing systems could exist. It is not directly obvious whether the local moves can assure convergence toward satisfaction of such a global criterion; the nondeterminacy as embodied by the daemon is an added complication. The question is settled by each of the following three constructs. For brevity's sake most of the heuristics that led me to find them, together with the proofs that they satisfy the requirements, have been omitted and—to quote Douglas T. Ross's comment on an earlier draft, "the appreciation is left as an exercise for the reader." (For the cyclic arrangement discussed below the discovery that not all

---

Author's address: Burroughs Corporation, Plataanstraat 5, Nuenen-4565, The Netherlands.

643

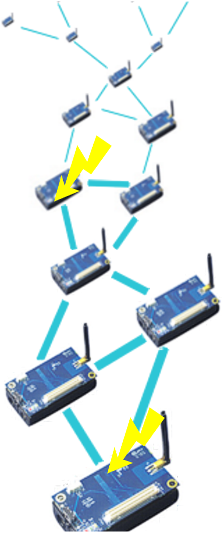
Communications
of
the ACM

November 1974
Volume 17
Number 11

Definition (Dijkstra 1974)

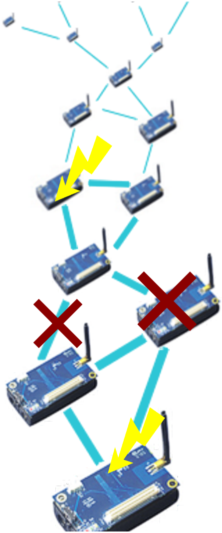
We call the system "self-stabilizing" if and only if, regardless of the initial state [. . .] the system is guaranteed to find itself in a legitimate state after a finite number of moves.

Fault Model



- Transient faults
- Caused by environmental influences
  - Wireless channel characteristics
  - Cosmic rays
  - ...
- Lasting effect on state of the network
  - Message loss or corruption
  - Reset of nodes
  - Corruption of memory

Fault Model

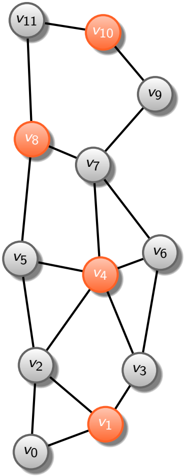


- Transient faults
- Caused by environmental influences
  - Wireless channel characteristics
  - Cosmic rays
  - . . .
- Lasting effect on state of the network
  - Message loss or corruption
  - Reset of nodes
  - Corruption of memory
- Other faults like:
  - Discharged nodes
  - Broken links
- Can be modeled as transient faults

Benefits of Self-Stabilization

- Inherent non-masking fault tolerance
- Formally verifiable
- Proofs are based on simple model
- Transformation to realistic model possible
- While preserving self-stabilization property

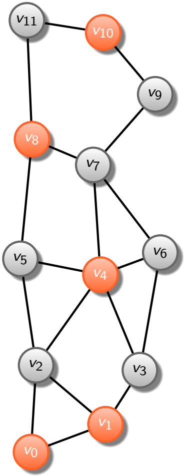
Maximal Independent Set



Example (Maximal Independent Set)

```
    public bool in;
rule R1:
    in = false and forall(Neighbors v : v.in  =  false)
-> in := true;
rule R2:
    in = true and exists(Neighbors v : v.in  =  true) ->
in := false;
```

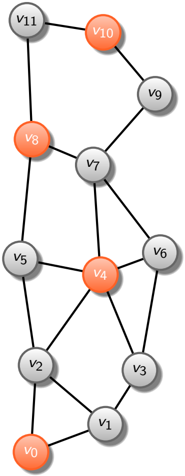
Maximal Independent Set



Example (Maximal Independent Set)

```
    public bool in;
rule R1:
    in = false and forall(Neighbors v : v.in = false)
-> in := true;
rule R2:
    in = true and exists(Neighbors v : v.in = true) ->
in := false;
```

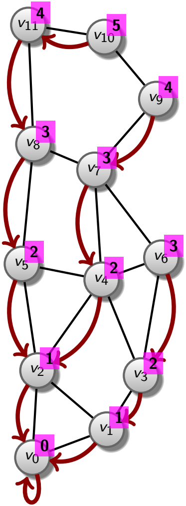
Maximal Independent Set



Example (Maximal Independent Set)

```
    public bool in;
rule R1:
    in = false and forall(Neighbors v : v.in = false)
-> in := true;
rule R2:
    in = true and exists(Neighbors v : v.in = true) ->
in := false;
```

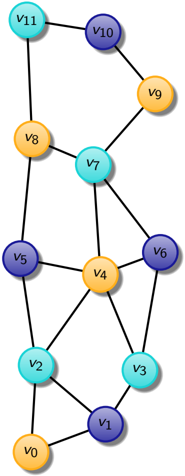
Spanning Tree



Example (Dolev 2000)

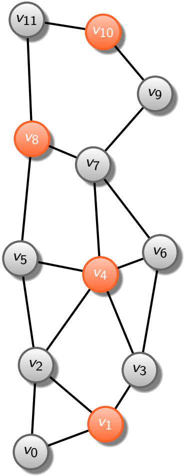
```
    public map NodeID Platform.ID as ID;
    public int dist;
    public NodeID parent;
    declare int minD := min(v.dist | Neighbors v);
rule R1:
     ID = 0 and !(parent = null and dist = 0) ->
          parent := null;
          dist := 0;
rule R2:
     ID != 0
     and !(parent in (v.ID | Neighbors v : v.dist  =  minD)
     and dist = minD + 1) ->
          parent := choose(v.ID | Neighbors v : v.dist = minD);
          dist := minD + 1;
```

Vertex Coloring



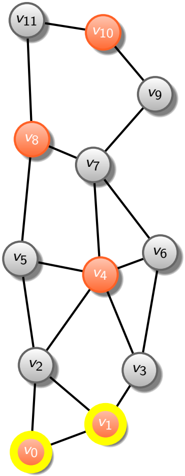
```
 public map int Neighborhood.numOfNeigh as d;
 public int c;
 declare set int colors := (1:d);
 declare bool
    B1 := c in (v.c|Neighbors v) or c>d+1;
 declare bool
    B2 := colors = (v.c|Neighbors v);
rule R1:
  B1 and B2 ->
      c := d + 1;
rule R2:
  B1 and !B2 ->
      c := choose(colors \ (v.c|Neighbors v));
```

Execution Model – Central Daemon Scheduler



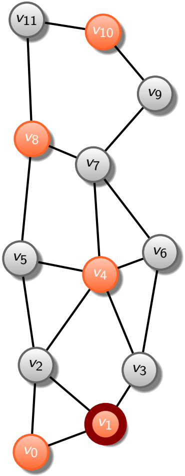
- Central entity (called daemon) assumed
- Algorithm execution is divided into rounds
- Daemon selects exactly one node
- Selection is fair
- Basically a serialization

Execution Model – Central Daemon Scheduler



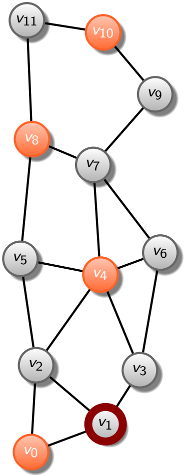
- Central entity (called daemon) assumed
- Algorithm execution is divided into rounds
- Daemon selects exactly one node
- Selection is fair
- Basically a serialization

Execution Model – Central Daemon Scheduler



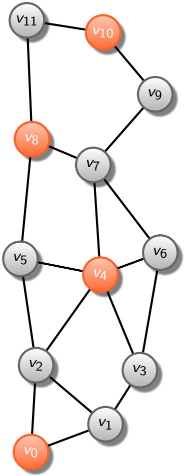
- Central entity (called daemon) assumed
- Algorithm execution is divided into rounds
- Daemon selects exactly one node
- Selection is fair
- Basically a serialization

Execution Model – Central Daemon Scheduler



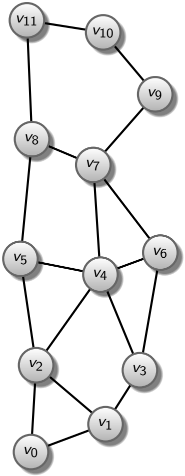
- Central entity (called daemon) assumed
- Algorithm execution is divided into rounds
- Daemon selects exactly one node
- Selection is fair
- Basically a serialization

Execution Model – Central Daemon Scheduler



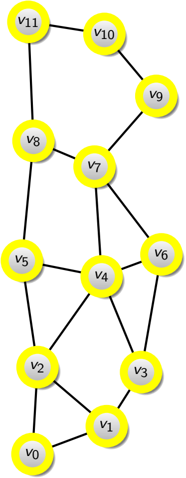
- Central entity (called daemon) assumed
- Algorithm execution is divided into rounds
- Daemon selects exactly one node
- Selection is fair
- Basically a serialization

Execution Model – Synchronous



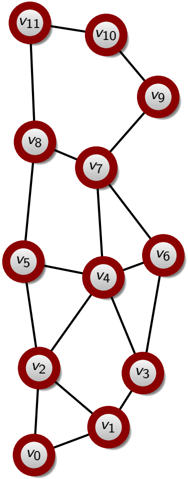
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



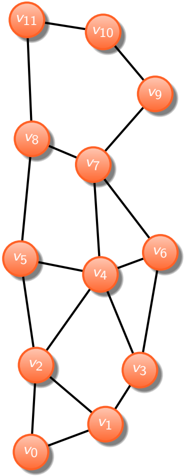
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



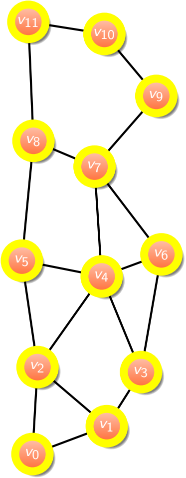
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



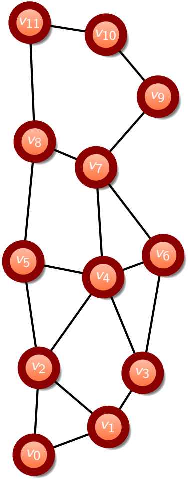
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



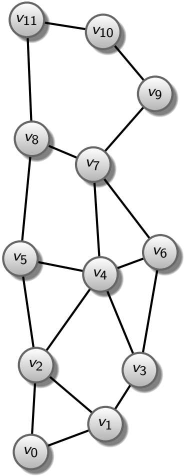
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



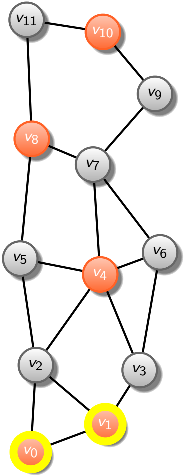
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Execution Model – Synchronous



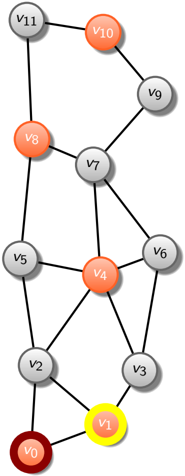
- Algorithm execution is divided into rounds
- Every enabled node is automatically activated
- Is not equivalent to central daemon model!

Transformations for WSN



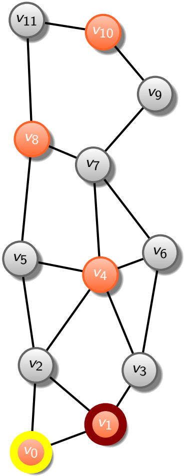
- Communication Model
  - Cached Sensornet Transformation (Herman 2003)
- Execution Model
  - Strict transformations
    - Deterministic conflict manager (Gradinariu, Tixeuil 2007)
    - BitToss (Goddard, Hedetniemi, Jacobs, Srimani 2008)
  - Weak transformations
    - Randomized conflict manager (Gradinariu, Tixeuil 2007)
    - Randomized transformation(Turau, Weyer 2006)

Transformations for WSN



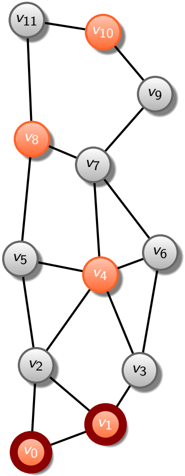
- Communication Model
  - Cached Sensornet Transformation (Herman 2003)
- Execution Model
  - Strict transformations
    - Deterministic conflict manager (Gradinariu, Tixeuil 2007)
    - BitToss (Goddard, Hedetniemi, Jacobs, Srimani 2008)
  - Weak transformations
    - Randomized conflict manager (Gradinariu, Tixeuil 2007)
    - Randomized transformation(Turau, Weyer 2006)

Transformations for WSN



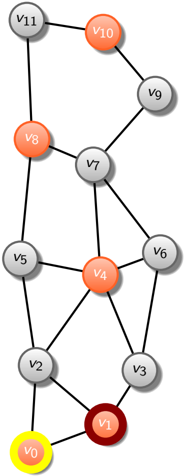
- Communication Model
    - Cached Sensornet Transformation (Herman 2003)
- Execution Model
    - Strict transformations
        - Deterministic conflict manager (Gradinariu, Tixeuil 2007)
        - BitToss (Goddard, Hedetniemi, Jacobs, Srimani 2008)
    - Weak transformations
        - Randomized conflict manager (Gradinariu, Tixeuil 2007)
        - Randomized transformation(Turau, Weyer 2006)

Transformations for WSN



- Communication Model
  - Cached Sensornet Transformation (Herman 2003)
- Execution Model
  - Strict transformations
    - Deterministic conflict manager (Gradinariu, Tixeuil 2007)
    - BitToss (Goddard, Hedetniemi, Jacobs, Srimani 2008)
  - Weak transformations
    - Randomized conflict manager (Gradinariu, Tixeuil 2007)
    - Randomized transformation(Turau, Weyer 2006)

Transformations for WSN



- Communication Model
  - Cached Sensornet Transformation (Herman 2003)
- Execution Model
  - Strict transformations
    - Deterministic conflict manager (Gradinariu, Tixeuil 2007)
    - BitToss (Goddard, Hedetniemi, Jacobs, Srimani 2008)
  - Weak transformations
    - Randomized conflict manager (Gradinariu, Tixeuil 2007)
    - Randomized transformation(Turau, Weyer 2006)

Properties of Transformations

- Strict transformations
  - Advantage: equivalent to central daemon
  - Drawback: limited concurrent activity
- Weak transformations
  - Advantage: allow for more concurrency
  - Drawback: only probabilistic convergence

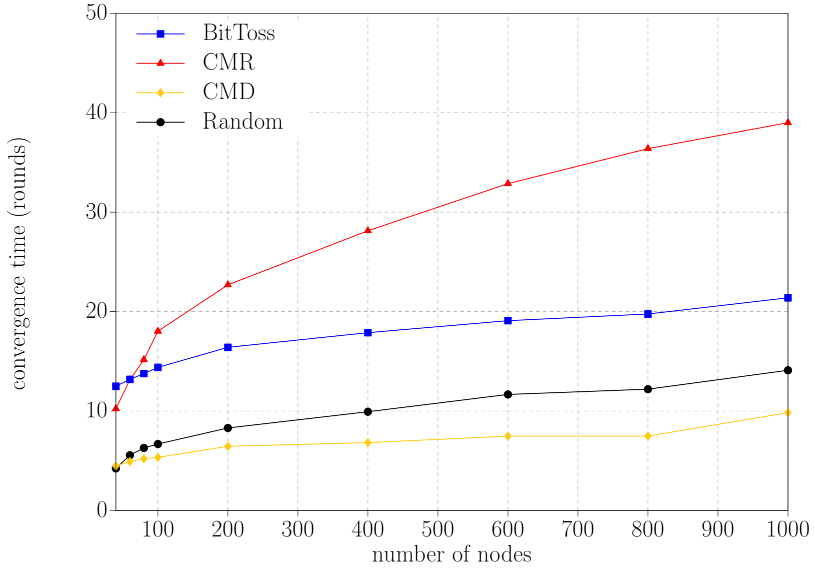
Major Concern: Convergence Time

- Represents responsiveness of algorithms
- High convergence time leads to low availability
- Major Question: Influence of transformations on convergence time?
- Do weak transformations reduce convergence time more than strict ones?

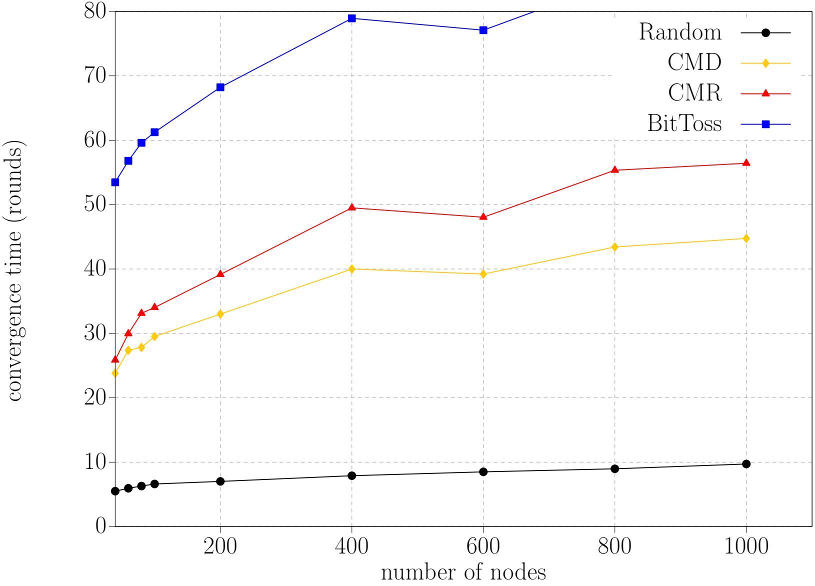
Upper Bounds vs. Average

- Determining convergence time analytically yields upper bounds
- Analytical determination of average is prohibitive ⇒ large value space
- Only practical method: simulation
- Contribution: analysis of convergence time of three algorithms central to WSN applications

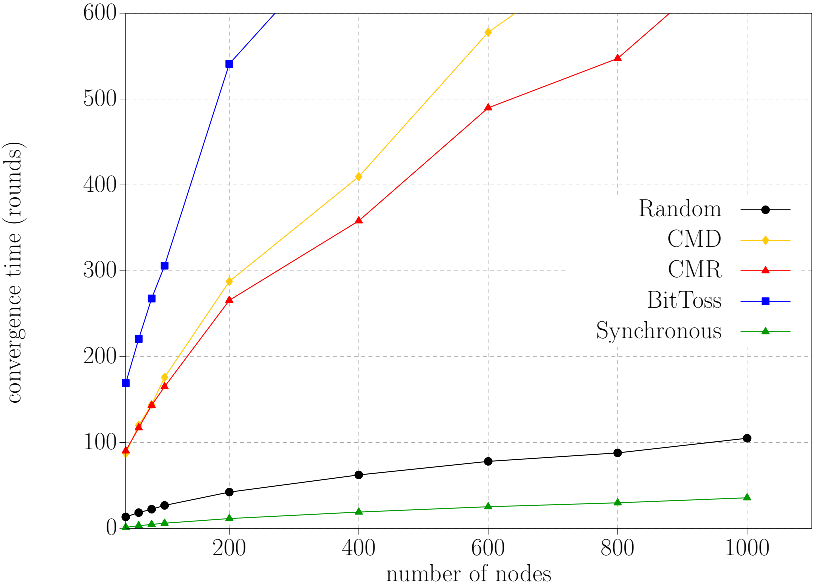
Maximal Independent Set (Density 9)



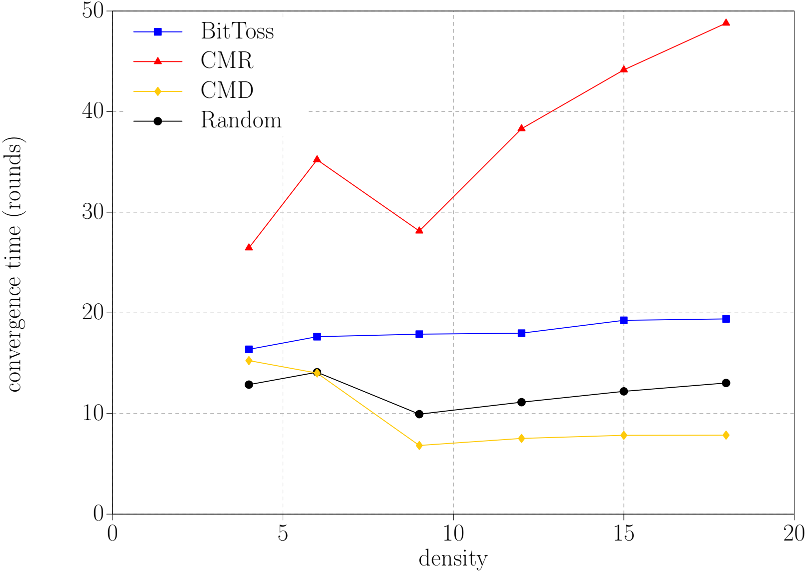
Vertex Coloring (Density 9)



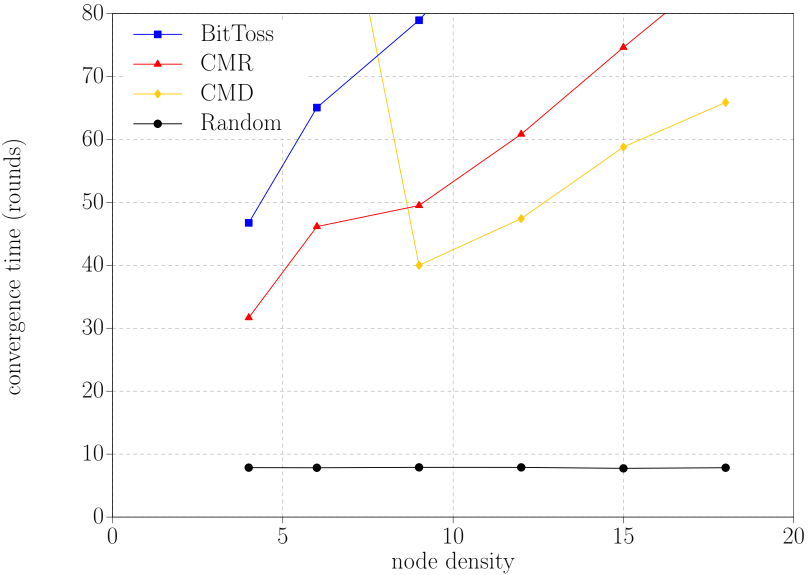
Spanning Tree (Density 9)



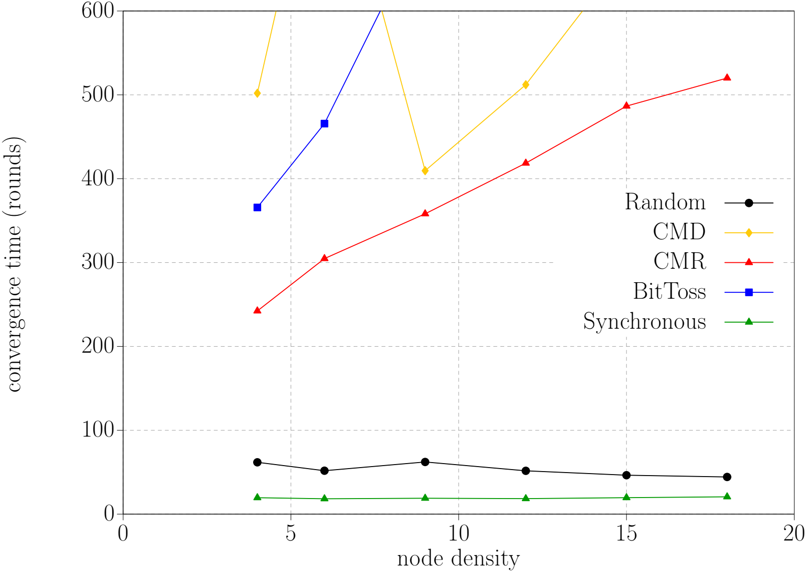
Maximal Independent Set (400 Nodes)



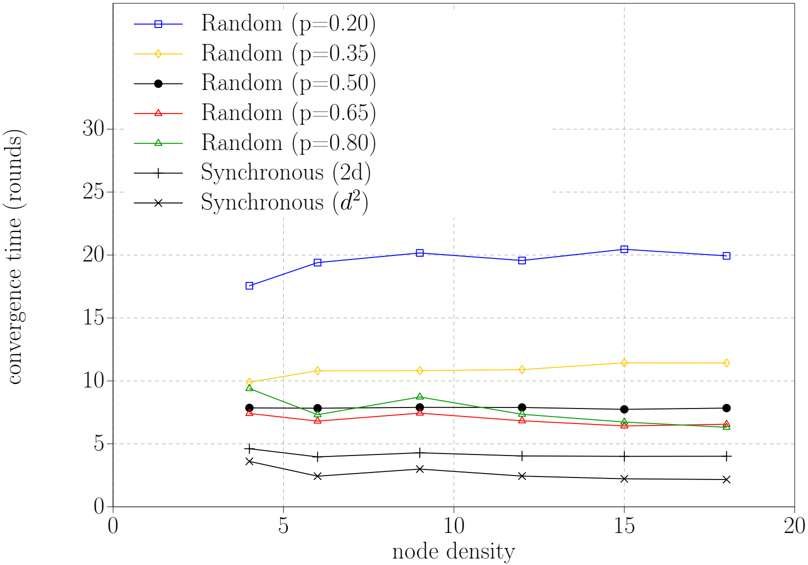
Vertex Coloring (400 Nodes)



Spanning Tree (400 Nodes)



Vertex Coloring (400 Nodes, Synchronous Version)



Summary

- average convergence time much better than upper bounds from literature
- randomized transformation very good performance
- with randomized transformation convergence time only depends on convergence time of original algorithm

- Outlook
  - Use SelfWISE on real sensor hardware (e. g. TMoteSky or SunSpot).
  - Determine the duration of a round under real conditions.

Thank You!

Questions ?

Basic Definitions

- The state of node is described by its variables
- Configuration c of network is tuple of node states
- Each node has strict local view upon network
    - Node can read/write own state
    - Node can read state of neighbors
- Absence of faults is defined by a predicate $\mathcal{P}$
- A configuration is legitimate if it satisfies $\mathcal{P}$
- A transition $c \rightarrow c'$ is caused by executing an algorithm
- An algorithm consists of rules of the following kind

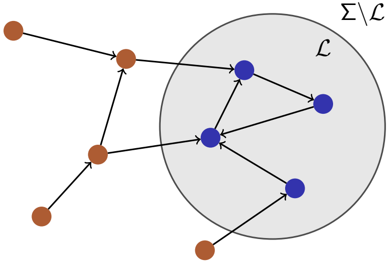
$$\begin{aligned} &guard_1 \longrightarrow statement_1 \\ &guard_2 \longrightarrow statement_2 \\ &\ldots \end{aligned}$$

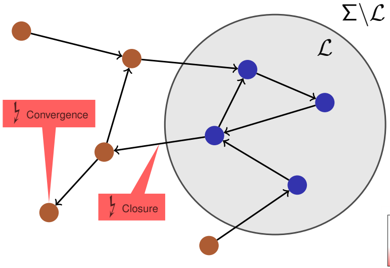
Main Definition

Definition (Self-Stabilization)

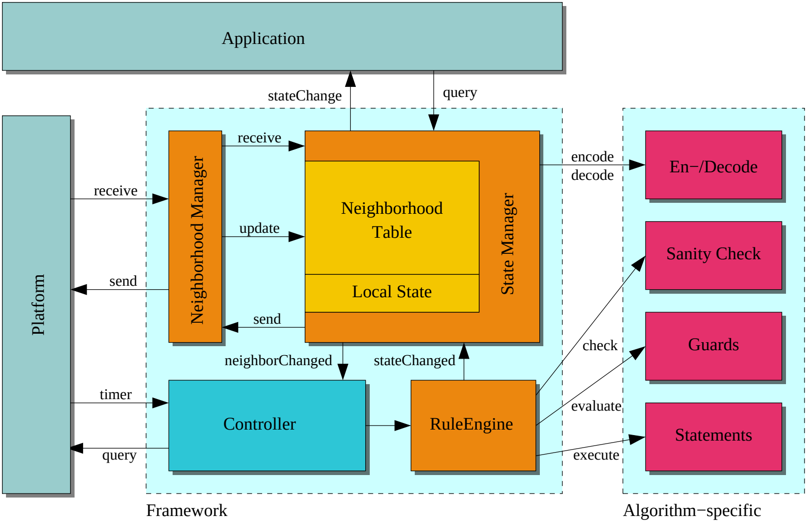
Let $\mathcal{L}$ be the set of all legitimate configurations relative to a predicate $\mathcal{P}$. A system is self-stabilizing with respect to $\mathcal{P}$ if:

1. If $c \in \mathcal{L}$ and $c \rightarrow c'$ then $c' \in \mathcal{L}$ (*closure property*)
2. Starting from any configuration every execution reaches $\mathcal{L}$ within a finite number of transitions (*convergence property*)





SelfWISE


Application
stateChange
query
Platform
receive
send
timer
query
Neighborhood Manager
receive
update
send
Neighborhood Table
Local State
State Manager
neighborChanged
stateChanged
Controller
RuleEngine
encode
decode
check
evaluate
execute
En−/Decode
Sanity Check
Guards
Statements
Framework
Algorithm−specific