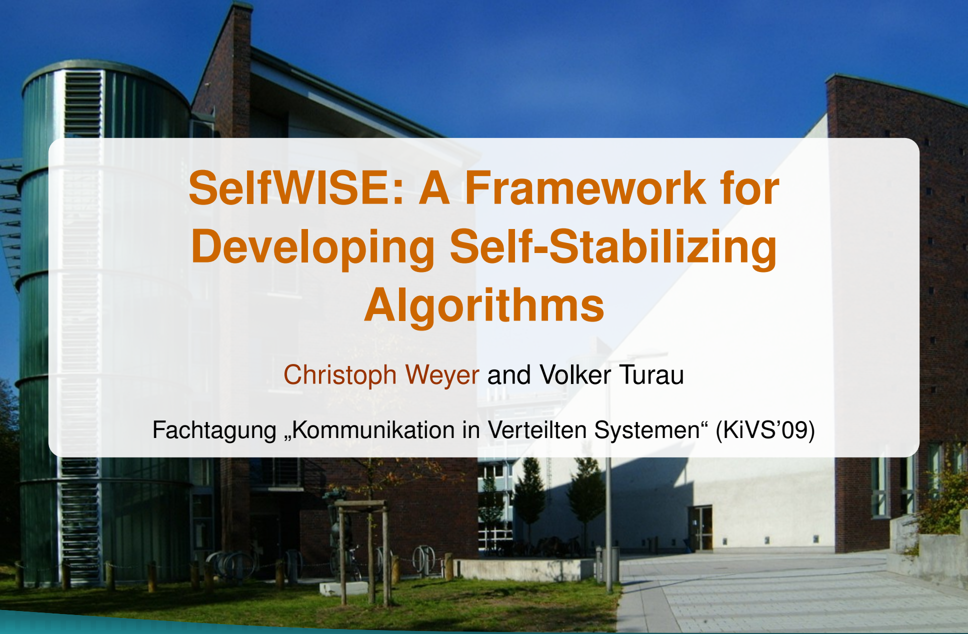


SelfWISE: A Framework for Developing Self-Stabilizing Algorithms

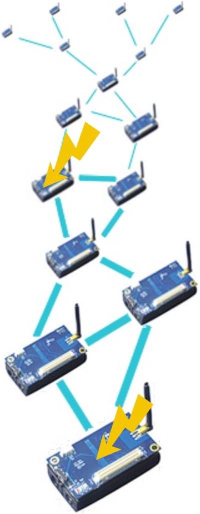
Christoph Weyer and Volker Turau

Fachtagung „Kommunikation in Verteilten Systemen“ (KiVS’09)

Institute of Telematics
Hamburg University of Technology
TUHH

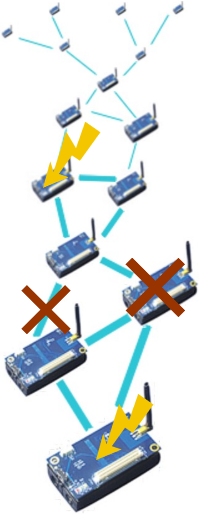
Self-Stabilization – A Child's Play?

Fault Model



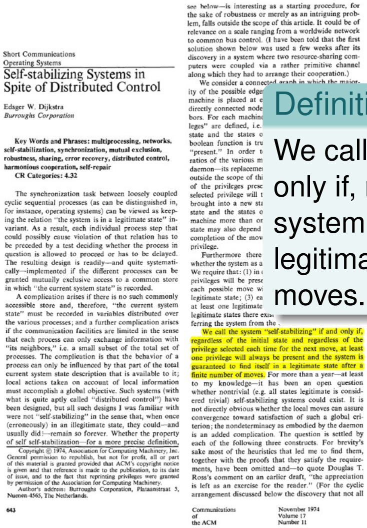
- Transient faults
- Caused by environmental influences
  - Wireless channel characteristics
  - Cosmic rays
  - . . .
- Lasting effect on state of the network
  - Message loss or corruption
  - Reset of nodes
  - Corruption of memory
- Other faults like:
  - Depleted nodes
  - Broken links
- Can be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

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:
  - Depleted nodes
  - Broken links
- Can be modeled as transient faults

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 major-
ity of the possible edge
machine is placed at e
directly connected node
bors. For each machin
leges" are defined, i.e.
state and the states o
boolean function is tru
"present." In order t
ratios of the various m
daemon—its replaceme
outside the scope of thi
of the privileges prese
selected privilege will t
brought into a new sta
state and the states o
machine more than or
state may also depend
completion of the mov
privilege.

Furthermore there
whether the system as a
We require that: (1) in
privileges will be prese
each possible move wi
legitimate state; (3) ea
at least one legitimate
legitimate states there exis
ferring the system from the

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.

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.

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

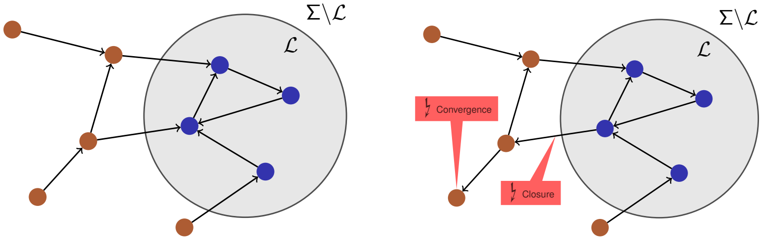
$$
\begin{array}{l}
guard_1 \longrightarrow statement_1 \\
guard_2 \longrightarrow statement_2 \\
\dots
\end{array}
$$

Main Definition

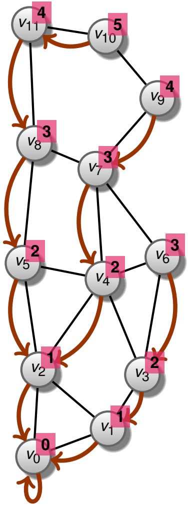
Definition (Self-Stabilization)

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*)



Spanning Tree



Example (Dolev 2000)

$$minNei(w, v) \equiv w \in N(v) \land \forall x \in N(v) : w.dist \leq x.dist$$

$$minDist(v) \equiv \min\{w.dist \mid w \in N(v)\}$$

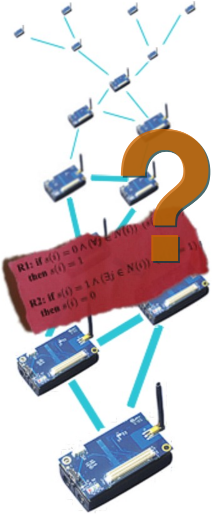
Root node

```
$\neg(parent = null \land dist = 0) \longrightarrow$
    $parent := null$
    $dist := 0$
```

Other node

```
$\neg(minNei(parent, v) \land dist = minDist(v) + 1) \longrightarrow$
    choose $w \in N(v)$ with $minNei(w, v)$
    $parent := w$
    $dist := minDist(v) + 1$
```

Adapting to Wireless Networks

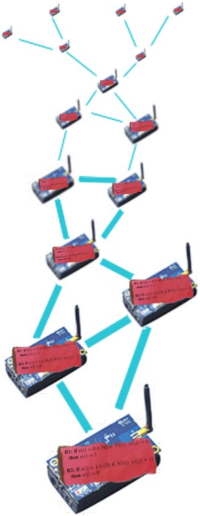


- Algorithms defined for abstract models
    - Shared memory (node state exchange)
    - Central Daemon (serial execution)
- Not suitable for wireless networks
- Transformations preserving self-stabilizing
- Existing transformations
    - Each node broadcasts its state
    - Nodes cache state of each neighbor
    - Randomized execution to break symmetry
- Still open research area

Next Part

SelfWISE Framework

Motivation for SelfWISE



- Need for simplifying the programming
  - Hide low-level details
  - Abstraction of accessing the wireless channel
  - Overcome limitation of resources
- Facilitate development of self-stabilizing algorithms
  - Integrated support for debugging and evaluation
  - Simulating behavior in different topologies
- Standard way for applying transformations
- Comparable statistics

SelfWISE – Language (I)

- Based on formal specification of algorithms
- Language is restricted to self-stabilizing algorithms
- Basic structure of an algorithm

```
algorithm name
  variable declarations
  macro definitions
rule name:
  guard -> statements
```

- Declaration of variables
  - Basic data types (e.g., `bool` or `int`) are supported

    ```
    public int dist;
    ```
  - Special data types `Node` and `NodeID`

    ```
    public Node parent;
    ```
  - Mapping of platform specific elements

    ```
    public map NodeID Platform.ID as ID;
    ```

SelfWISE – Language (I)

- Based on formal specification of algorithms
- Language is restricted to self-stabilizing algorithms
- Basic structure of an algorithm

```
algorithm name
  variable declarations
  macro definitions
rule name:
  guard -> statements
```

- Declaration of variables
  - Basic data types (e.g., **`bool`** or **`int`**) are supported

    ```
    public int dist;
    ```
  - Special data types **`Node`** and **`NodeID`**

    ```
    public Node parent;
    ```
  - Mapping of platform specific elements

    ```
    public map NodeID Platform.ID as ID;
    ```

SelfWISE – Language (II)

- Operations upon neighboring nodes
  - Set of all neighbors (`Neighbors`)
  - Iterator over neighborhood (`Neighbors v`)
  - Filtering neighborhood

    `(Neighbors v :  v.dist = minD)`
- Simple set operations
  - Choose one element in set

    `choose(Neighbors v :  v.dist = minD);`
  - Check if element is in set

    `parent in (Neighbors v :  v.dist = minD);`
  - Get minimum, maximum or average of a set

    `min(v.dist | Neighbors v);`
- Macro definition

```
declare int minD := min(v.dist | Neighbors v);
```

SelfWISE – Language (II)

- Operations upon neighboring nodes
  - Set of all neighbors `(`**`Neighbors`**`)`
  - Iterator over neighborhood `(`**`Neighbors`**` v)`
  - Filtering neighborhood

    `(`**`Neighbors`**` v :  v.dist = minD)`
- Simple set operations
  - Choose one element in set

    `choose``(`**`Neighbors`**` v :  v.dist = minD);`
  - Check if element is in set

    `parent `**`in`**` (`**`Neighbors`**` v :  v.dist = minD);`
  - Get minimum, maximum or average of a set

    `min``(v.dist | `**`Neighbors`**` v);`
- Macro definition

  `declare int`` minD := `**`min`**`(v.dist | `**`Neighbors`**` v);`

SelfWISE – Language (II)

- Operations upon neighboring nodes
  - Set of all neighbors (`Neighbors`)
  - Iterator over neighborhood (`Neighbors v`)
  - Filtering neighborhood

    `(Neighbors v :  v.dist = minD)`
- Simple set operations
  - Choose one element in set

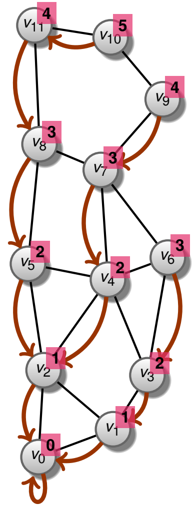
    `choose(Neighbors v :  v.dist = minD);`
  - Check if element is in set

    `parent in (Neighbors v :  v.dist = minD);`
  - Get minimum, maximum or average of a set

    `min(v.dist | Neighbors v);`
- Macro definition

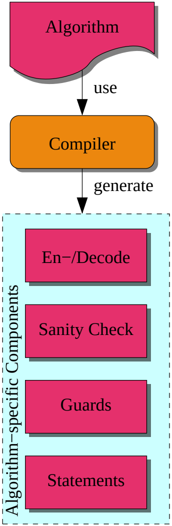
```
declare int minD := min(v.dist | Neighbors v);
```

Example: Spanning Tree



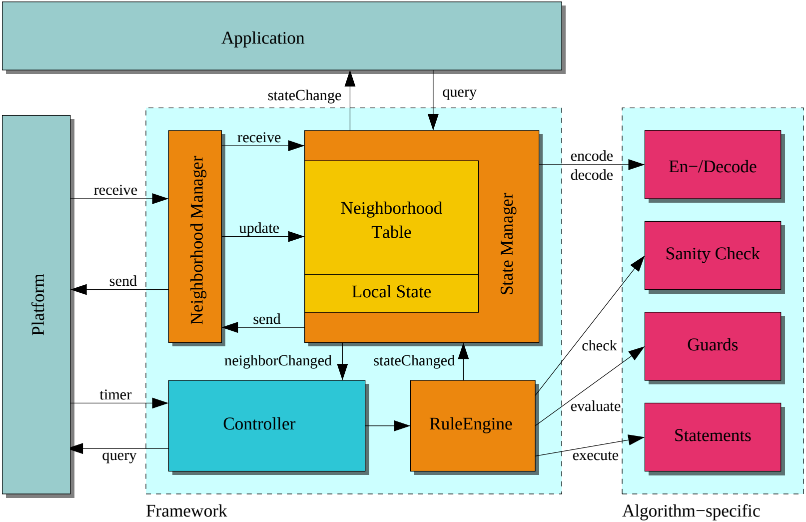
```
algorithm SpanningTree;
 public map NodeID Platform.ID as ID;
 public Node parent;
 public int dist;
 declare int minD := min(v.dist|Neighbors v);
rule R1:
  ID = 0 and !(parent = null and dist = 0) ->
    parent:=null;
    dist:=0;
rule R2:
  ID != 0 and
   !((parent in (Neighbors v:v.dist=minD))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

SelfWISE – Compiler

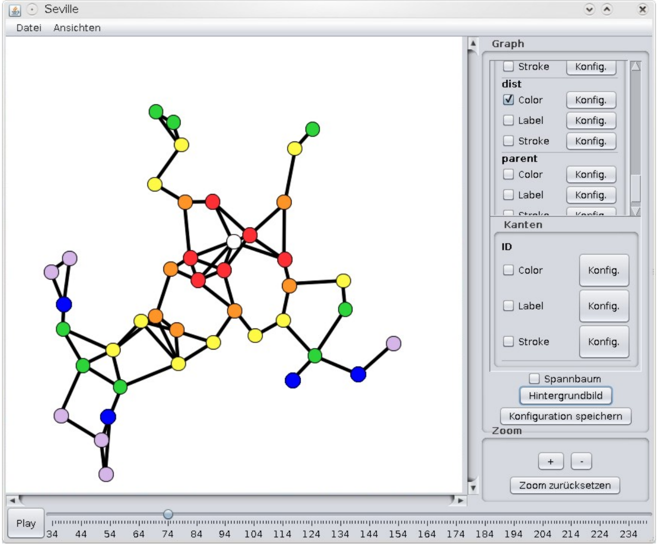


- Create different components
  - Separate each rule into guard and statement
  - Initialization and sanity checks
  - Encoding and decoding for network representation
- Must preserve self-stabilizing properties

SelfWISE – Architecture



SelfWISE – Visualization



Current SelfWISE Implementations

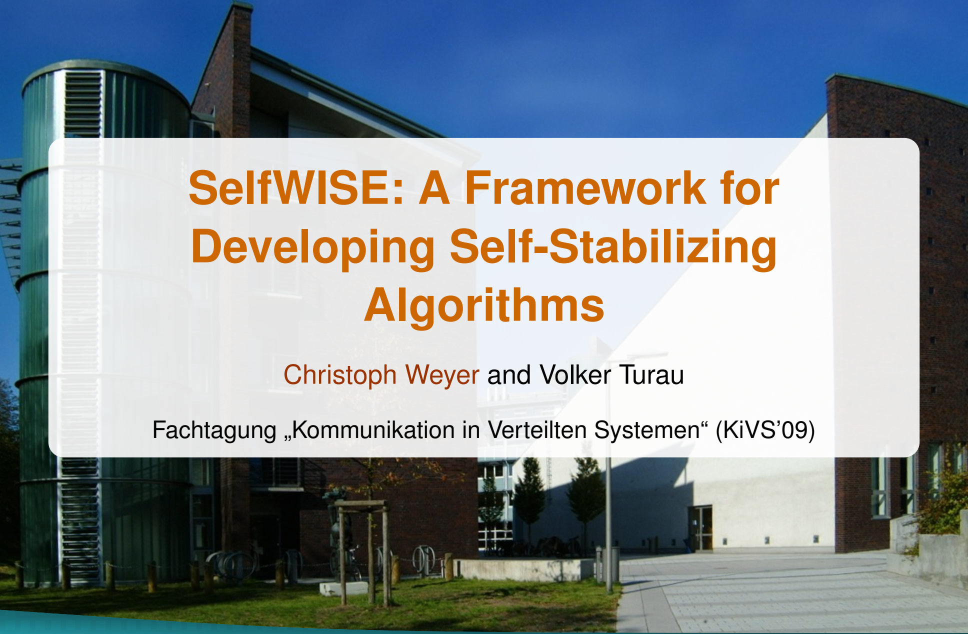
- SelfWISE framework
  - TU Hamburg-Harburg: TinyOS/TOSSIM
  - BTU Cottbus: Reflex/OMNeT++
- Implementation state
  - Six different transformations
  - Several algorithms
    - spanning tree, vertex coloring, clustering, . . .
  - First comparisons of transformation performance
- Current focus
  - Performance improvements by porting to ns2
  - Integration of fault injection facility

Conclusion

- Realization preserves self-stabilization properties
- Framework is suitable for limited memory
  - Around 20 kB (ROM) and 110 Byte (RAM)
- SelfWISE helps the evaluation of self-stabilizing algorithms

Next steps

- Investigation of different transformations
- Using self-stabilization in a real deployment



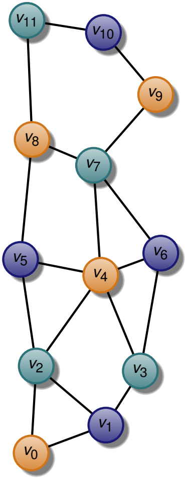
SelfWISE: A Framework for Developing Self-Stabilizing Algorithms

Christoph Weyer and Volker Turau

Fachtagung „Kommunikation in Verteilten Systemen“ (KiVS’09)

Institute of Telematics
Hamburg University of Technology
TUHH

Example: Vertex Coloring



```
algorithm VertexColoring;
 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));
```