Programming Wireless Sensor Networks in a Self-Stabilizing Style

Christoph Weyer, Volker Turau
Institute of Telematics
Hamburg University of Technology
Hamburg, Germany
{c.weyer, turau}@tu-harburg.de

Andreas Lagemann, Jörg Nolte
Distributed Systems/Operating Systems
Brandenburg University of Technology Cottbus
Cottbus, Germany
{ae, jon}@informatik.tu-cottbus.de

Abstract

Wireless sensor networks operate in an unstable environment and thus are subject to arbitrary transient faults. Self-stabilization is a promising technique to add tolerance against transient faults in a self-contained non-masking way. A core factor for the applicability of a given self-stabilizing algorithm is its convergence time. This paper analyses the average stabilization time of three algorithms commonly regarded as central building blocks for wireless sensor networks. The analysis is accomplished with SelfWISE, a framework providing programming abstractions for self-stabilizing algorithms. The performed analysis considers the target models as well as network size and density. This demonstrates the usability of SelfWISE for evaluating self-stabilizing algorithms under a wide range of models.

Keywords: Self Stabilization, Wireless Sensor Networks, Model Transformation

1. Introduction

The characteristics of the wireless communication medium and other environmental influences (natural as well as artificial) lead to the frequent occurrence of transient faults [1]. Therefore, algorithms for wireless sensor networks (WSNs) need to be fault-tolerant especially against this kind of faults; otherwise operation over longer periods are not feasible. Self-stabilizing techniques [9] provide non-masking fault-tolerance in a self-contained manner. The applicability of these techniques to wireless sensor networks (WSNs) has therefore been a hot research topic in recent years. One important measure for the suitability of a self-stabilizing algorithm is its convergence time, because it expresses the time the service is not available. However, analytical studies of convergence time yield upper bounds only, which may be far from the average case.

What makes the property of self-stabilization outstanding, is that it can be formally verified in mathematical proofs. The complexity of such a proof directly depends on the assumptions made about the order and level of concurrency in which nodes may execute the algorithm. To keep things simple, the majority of algorithms use a sequential execution model, where exactly a single node executes at a time. This allows to concentrate on issues directly concerned with the self-stabilization property and leave aside concurrency issues. It is legitimate to do so, because it has been shown that algorithms written for one model can be adapted to run under another model. This is accomplished by generic transformers, which preserve the self-stabilizing properties, without detailed knowledge of the particular algorithm. To use self-stabilizing algorithms in sensor networks it is necessary to adapt them to a suitable model.

SelfWISE is a framework, which provides a programming abstraction that allows a formal specification of algorithms. These are automatically compiled into WSN applications based on a runtime environment where the developer can choose among a set of common transformations. This paper uses SelfWISE to assess the average convergence time of three algorithms with relevance to WSNs: Maximal independent set - used in clustering -, vertex coloring - used to construct TDMA schedules -, and breadth first spanning tree - used in routing algorithms. The results are compared with upper bounds from theoretical analyzes. Insights into the influence of different model adaptations to the resulting convergence time are provided.

The paper is structured as follows: First basic terms and concepts of self-stabilization in WSNs are introduced. Then we describe the SelfWISE programming abstraction and runtime environment. The algorithms analyzed in this paper are introduced using the formal notation provided by the SelfWISE language. Section 4 presents results and an evaluation. The paper concludes with a recapitulation of the results and by defining goals for future work.

2. Self-Stabilization in WSNs

Applying self-stabilizing algorithms in the field of WSNs to increase the fault-tolerance is currently an active research area [6, 7, 8, 10, 13, 14]. A tremendous advantage of self-stabilization is that it does not handle individual failures separately. Instead of modeling individual errors that may occur and providing corresponding recovery routines, self-stabilizing systems are based on a description of the error-free system and rules to reach and maintain this state. This avoids the drawback of fault masking approaches that handle a priori known faults only. Rules are based upon the states of a node and its direct neighbors. This strictly local view leads to scalable solutions with immanent integrated fault-tolerance regarding transient faults that may corrupt the state of some nodes. Common examples of such transient faults are: message loss or corruption, memory corruption, or node resets.

algorithm MaximalIndependentSet;
  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**;

Figure 1. Maximal Independent Set [4]

Figure 1 shows a simple example for a self-stabilizing algorithm: it constructs a maximal independent set (MIS) as described in [4]. The language of SelfWISE (see Section 3) is used to express this algorithm. The membership of a node in the independent set is indicated by setting the variable `in` to true. Such an algorithm typically consists of a set of rules, which in turn consist of a guard and a statement which shall be executed when the guard evaluates to true. Commonly their structure is very simple and they are easy to understand. Self-stabilizing algorithms are defined more formally in the following section.

2.1. Models of Self-Stabilization

A wireless communication network is represented by a graph $G = (V, E)$. Vertices $v_i \in V$ of the graph represent network nodes and edges $e_i \in E$ represent bidirectional communication links between those. The diameter of the graph is denoted by Δ. The *state* s_i of each node $v_i \in V$ is defined by the set of local variables. The tuple $(s_1, \ldots, s_n)$ forms the *configuration* of the network and defines the global state. A self-stabilizing algorithm consists of rules in the form of guarded statements:

$$guard_i \rightarrow statement_i$$

Guards are Boolean expressions based on the state s_i of the node v_i and the states of all nodes $v_k \in N(v_i)$ in the neighborhood of v_i. The node degree is given by $d_i = |N(v_i)|$. If a guard of a node evaluates to true, it is called *enabled*. An enabled node performs a *move* by executing the statement of a rule. Statements can change the local state only. An *execution* is a sequence of configurations $c_0, c_1, \ldots$ such that the transition from c_i to c_{i+1} is caused by moves of nodes that are enabled in configuration c_i.

Informally self-stabilization means that if a system is in a fault-free state, it remains in this state as long as no transient error occurs. After a transient error has occurred, the system reaches a fault-free state in finite time. More formally let $\mathcal{P}$ be a predicate over the configuration of the network that defines the fault-free state of the system. A configuration is *legitimate* relative to $\mathcal{P}$ if it satisfies $\mathcal{P}$. A system is *self-stabilizing* with respect to $\mathcal{P}$ if the following two properties hold: (1) Closure: A transition always moves a legitimate configuration into a legitimate one. (2) Convergence: Starting from an arbitrary configuration a legitimate one is reached within a finite number of transitions. This definition holds under the assumption that faults may corrupt the configuration of the network, but not the behavior of the network, i.e., rules are stored in fault resilient memory.

To model the different degrees of concurrency in distributed systems the concept of *daemons* was introduced. A daemon defines the execution model of a self-stabilizing system. It chooses a subset of enabled nodes that perform their moves concurrently. Three different daemons are defined: *Central daemon* – only one node is selected from the set of enabled nodes, *synchronous daemon* – all enabled nodes are selected, and *distributed daemon* – a nonempty subset is selected. The communication model describes the underlying assumptions about the communication paradigms that are used to exchange the node state information among the neighborhood.

Many self-stabilizing algorithms are defined for an abstract computational model: central daemon, shared memory, coarse-grained atomicity. This model simplifies the design and verification of self-stabilizing algorithms. But, the assumptions of this model are against the spirit of distributed systems since it does not allow concurrency (i.e., central daemon and atomicity) and are not feasible in wireless networks (i.e., shared memory). Algorithms developed for a central daemon often do not stabilize under a distributed or synchronous daemon due the concurrent execution within the neighborhood. The MIS algorithm depicted in Fig. 1 is an example of such an algorithm. To solve this so called *transformations* have been proposed [6, 3, 2, 10]. They convert algorithms designed for such abstract models into semantically equivalent algorithms that stabilize under weaker assumptions.

2.2. Transformations for WSNs

In general transformations are defined for adopting the communication or the execution model. To overcome the shortcomings of the shared memory model, Herman [6] introduced a concept for exchanging node states that uses the broadcast characteristic of the underlying wireless channel. The *cached sensornet transformation* uses periodic broadcasts of the own state and a cache to store the states received from the neighbors. Rule evaluation is performed by using the cache. For ensuring atomicity during the evaluation of the rules, a concept of *rounds* must be introduced [10]. At the beginning of each round the nodes evaluate and execute the rules and broadcast the new state during the rest of the round. To prevent collisions a random backoff timer is used.

Transformations of the execution model ensure the exclusive execution within each neighborhood under the distributed or synchronous daemon. The idea behind these transformations is to break the symmetry by using unique identifiers or randomization. A *strict transformation* converts the algorithms in such a way that the execution of the resulting algorithms is equivalent to an execution under the central daemon. An algorithm $\mathcal{A}$ is transformed into $\mathcal{A}'$ such that only one node in each neighborhood performs a move of $\mathcal{A}$ concurrently. Examples for strict transformer are the deterministic conflict manager (CMD) [3] that uses unique node identifiers and BitToss [2]. The latter elects a neighbor by a Bernoulli trial until solely a single one node is enabled. The main drawback of these strict transformer is the limited concurrent activity, exactly one node within each neighborhood executes its statement. Often this limitation is too restrictive and a higher degree of concurrency is needed. Algorithms converted by a *weak transformation* produce an execution that is not possible under a central daemon. The reason for this is the fact that nodes may perform a move within a neighborhood concurrently. The idea is that potential deceptive statement executions are resolved after some time, but with the advantage of a faster convergence. Examples for weak transformations are the randomized conflict manager (CMR) [3] and the randomized transformation introduced by Turau and Weyer [10], which both lead to a probabilistic convergence. The latter reference also proposed a transformation that is even self-stabilizing in the case of occasional message losses.

2.3. Convergence Time

Convergence time is a central performance measure of self-stabilizing algorithms. It represents the *responsiveness* of an algorithm with respect to the occurrence of transient faults. By means of the convergence time the suitability of a given algorithm for the use in real WSNs can therefore be determined. A high convergence time inhibits the use of an algorithm in a real sensor network because it reduces the *fault rate* that an algorithm can tolerate. If the mean time between the occurrence of two faults is shorter than the convergence time, the algorithm will hardly stabilize.

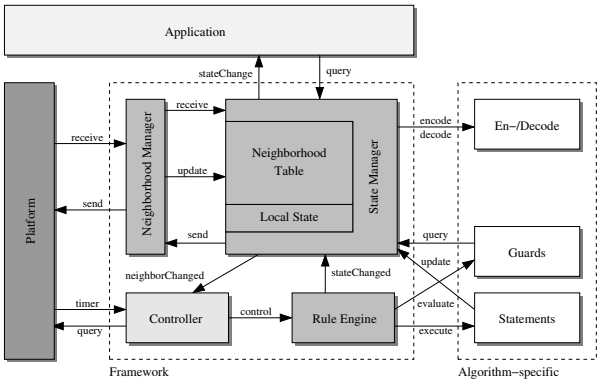


Figure 2. The SelfWISE framework

The convergence rate of a given algorithm strongly depends on the execution model that is applied. Traditionally most self-stabilizing algorithms have been written for the central daemon model. Since this model boils down to a sequential execution model convergence under a central daemon clearly is an upper bound [8, 7]. Whenever the execution model allows distributed execution the convergence time is likely to be better than that.

Section 2.2 introduced some transformations that have been proposed. To our knowledge no analysis of the transformation's influence on the convergence time has been carried out. Naturally, deriving measures of the convergence time analytically is done by proving upper bounds. Whereas deriving analytically the average convergence time of an algorithm is so laborious due to the size of the value space, that it is practically impossible. The only way to estimate the average convergence time for an algorithm under a certain model is simulation. To our knowledge there has been one publication that evaluated the convergence time by simulation for a limited subset of transformations for the execution model [8]. In this paper a detailed analysis of the convergence time of three algorithms commonly regarded as central building blocks for WSN applications is presented.

3. SelfWISE

SelfWISE [13] is a programming abstraction designed for applying self-stabilizing algorithm in WSNs. It consists of the SelfWISE framework (see Fig. 2) that is the runtime environment for executing self-stabilizing algorithm and a language to express those algorithms (see Fig. 1, ref-fig:color, and 4). The SelfWISE language is similar to the formal definitions of self-stabilizing algorithms found in lit-

```
algorithm VertexColoring;
    public map int Neighborhood.numOfNeighbors 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));
```

Figure 3. Vertex Coloring [5]

erature to simplify the usage for algorithm developers and to hide the complexity of wireless sensor networks. A compiler converts algorithms into C (or C++) code that is linked against the SelfWISE framework. The SelfWISE framework is running on top of the underlying operating system on each sensor node. At the time of writing there exist two implementations of the framework. One for TinyOS and one for REFLEX [12], an event driven operating system for deeply embedded systems. An optional application can use the available interfaces to operate on the outcome of the self-stabilizing algorithm.

The SelfWISE framework, as depicted in Fig. 2, consists of the following components. The easily exchangeable controller implements the execution model. Based on the implemented transformation the controller evaluates the guards and executes enabled rules via the rule engine. The rule engine hides the details of each algorithm from the controller. The local state of each node and the cached states of each neighbor are stored by the state manager. Node states are broadcast via the neighborhood protocol that is also responsible for establishing neighbor relations between nodes. The compiler creates additional de- and encoding routines for sending and receiving node states.

Figure 3 shows a self-stabilizing algorithm that creates a unique vertex coloring within 1-hop as introduced in [5]. Each node has a variable `c` that indicates the color of the node. The given algorithm tries to minimize the number of colors used. The maximal color index of each node is equal to the number of neighbors plus one. The algorithm is based on the predicates `B1` and `B2`. `B1` evaluates to true if the color is already used within the neighborhood or the color of the node is larger than the maximum available. The predicate `B2` is true if all color indices from 1 to number of neighbors are used in the neighborhood. The algorithm assigns a new color to a node if its color is already used or greater than allowed. The `choose` operator selects nondeterministically an unused color. Under this condition the algorithm stabilizes under the central scheduler only. If the available color space is increased to twice the number of neighbors or higher, as suggested in [8], the algorithm stabilizes under distributed control without any transformation.

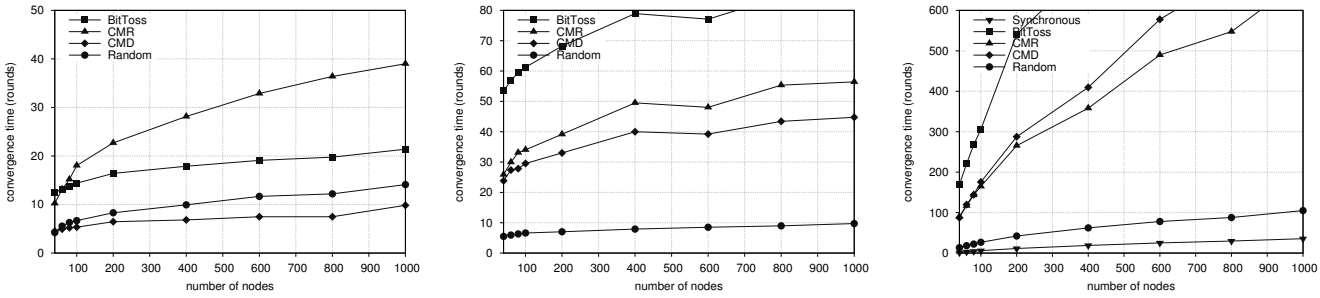
```
algorithm SpanningTree;
    public map NodeID Platform.ID as ID;
    public int dist;
    public NodeID parent;
    declare int minD := min(v.dist | Neighbors v);
rule R1:
    ID = 0 and !(parent = null and dist = 0) –>
        parent := null;
        dist := 0;
rule R2:
    ID != 0 and !(parent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

Figure 4. Spanning tree algorithm [1]

The last example of a self-stabilizing algorithm, as shown in Fig. 4, creates a spanning tree rooted at the node with the id zero. The algorithm is a non-uniform algorithm since the root node and all other nodes execute different rules [1]. The node identifier must be provided by the underlying platform and is mapped into the variable `ID`, which can be used within the algorithm. Each node has two variables. The hop distance to the root node is stored in `dist` and the parent node in the tree is stored in `parent`. Due to the non-uniform character of the algorithm the tree is constructed in a wave-like manner starting at the root node. Each node selects the node with the lowest distance as its parent and assigns itself a distance incremented by one. The root has an empty parent and a distance of zero. Due to the wave-like execution the algorithm stabilizes under distributed control without any transformation.

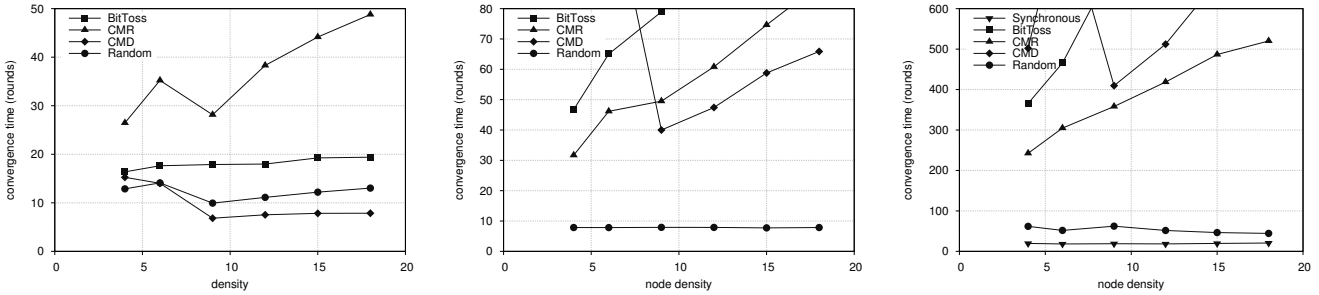
4. Evaluation

For the two available implementations of the SelfWISE framework several simulations are performed. TOSSIM the integrated TinyOS simulator and OMNeT++[11], which has been enhanced to simulate sensor nodes running REFLEX, are used for evaluating the different transformations introduced in Section 2.2. Both simulators yield similar results, so here the TOSSIM results are presented only. Each transformation and algorithm combination is simulated on a set of topologies. The number of nodes are varied (40, 60, 80, 100, 200, 400, 600, 800, and 1000) with different node densities (4, 6, 9, 12, 15, and 18). For each combination of network size and node density, different topologies are created. Each topology is simulated 10 times with different random seeds. That results in 22.000 simulations for each combination of transformer and algorithm leading to over 300.000 simulations in total for each simulator. To simulate the convergence time of the transformations no message losses due to noise or link variations over time are modelled. Collisions can occur if two nodes are transmitting concurrently.



(a) Maximal Independent Set (b) Vertex Coloring (c) Spanning Tree

Figure 5. Simulations with node density 9



(a) Maximal Independent Set (b) Vertex Coloring (c) Spanning Tree

Figure 6. Simulations with 400 nodes

To minimize the probability of collisions the round length is scaled appropriately.

Figures 5 and 6 show the results of the simulations. The used transformations are: BitToss [2], deterministic conflict manager (CMD) [3], randomized conflict manager (CMR) [3], randomized transformation (with a fixed probability $p = 0.5$) [10], and the synchronous daemon. The communication model utilizes CST as transformation.

CMD creates the fastest transformed maximal independent set algorithm (see Fig. 5(a)). If node identifiers are randomly distributed with a high probability in each neighborhood exactly one node concurrently performs a move. Since the variable `in` is initialized to false for all nodes, the maximal independent set is created after the first step of CMD. This advantage inverts for algorithms where a higher degree of parallelism does not lead to an increasing number of conflicts (see Fig. 5(b) and 5(c)). The real drawback of CMD is the usage of node identifiers. If a topology contains longer chains of increasing node identifiers, the convergence time dramatically increases (see Fig. 6). The probability of such chains increases with decreasing node density. This effect can be seen for topologies with density 4 and 6.

BitToss is not influenced by the distribution of the node identifiers since it is based on a Bernoulli trial. But, the trial increases convergence time since the election of the executing node can be postponed. This effect increases when more nodes need to perform a move currently within a neighborhood.

CMR and randomized transformation are selecting nodes based on randomization. This leads to more concurrent executions but also to deceptive executions. The main drawback of CMR is that the random process is scaled directly by the node degree. So the convergence time under CMR always depends on the node density (see Fig. 6). This leads to an increased convergence time in higher densities. The simulations clearly indicate, that for algorithms with a possible higher degree of concurrency (e.g., spanning tree) CMR is faster than CMD. The randomized transformation outperforms all presented transformations, except for MIS (see above). With adverse node identifier distribution, CMD's gain vanishes (see Fig. 6(a)) even here.

The upper bound for maximal independent set is given as $2n$ rounds [4] under the central daemon. Under distributed control the number of rounds is reduced due to the concurrent execution of moves among non-neighboring nodes. The convergence time of vertex coloring is shown to be $O(n)$ [5]. Average convergence time is clearly linear with a very small constant. The simulation results suggest that it might be even better. More data is needed to substantiate such a claim. Here, it can be seen plainly that the

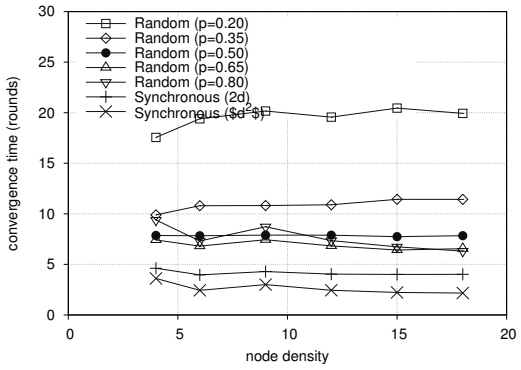


Figure 7. Vertex Coloring (400 nodes)

randomized transformation depends on n only and not on node density. The upper bound for the spanning tree algorithm, which stabilizes under the synchronous daemon, is given as $O(\Delta)$ rounds [1]. The randomized transformation does not introduce much overhead here either. Even for this strongly concurrent algorithm it remains the only transformation (apart from synchronous execution), where convergence time depends on Δ only. Note that in fixed size networks Δ decreases with increasing node density.

The vertex coloring algorithm stabilizes under a synchronous daemon when the color space is enlarged (see Section 3). That variant was simulated with a color space of $2d$ and d^2 under the synchronous daemon. In Fig. 7 the results are compared with coloring under the randomized transformation with different probabilities. The convergence time of the randomized transformation is comparable to the synchronous daemon with respect to the lower color space used. Increasing the probability slightly decreases the convergence time. If the probability p is decreased the convergence time rapidly increases, since with a low value for p the selection of a node can be postponed.

5. Conclusion

In this paper the first evaluation of the average convergence time of self-stabilizing algorithms and appropriate transformations is given. Due to the modular concept of SelfWISE evaluating different transformations is simplified. It provides a useful programming abstraction for self-stabilizing algorithms and computational models.

Our results clearly show that the average convergence time for the considered algorithms is generally much better than the upper bounds given in literature. Comparing results of the different transformations with upper bounds reveals a particularly interesting property of the randomized transformation: It is the only transformation where the convergence time does not depend on other parameters than the convergence time of the original algorithm.

The randomized transformation [10] shows good performance in all given topologies. Additionally, the benefit of solutions based on randomization over those based on identifiers is shown.

The next steps for future work are the usage of the SelfWISE framework on real sensor hardware to validate the gathered results under realistic conditions. It would for instance be interesting to evaluate the *duration* of one round: what is the minimum time for a round under realistic conditions and what factors have an influence on this time?

References

[1] S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, USA, Mar. 2000.

[2] W. Goddard, S. Hedetniemi, D. Jacobs, and P. K. Srimani. Anonymous Daemon Conversion in Self-stabilizing Algorithms by Randomization in Constant Space. In *Proc. 9th Int. Conf. on Distr. Comp. and Networking (ICDCN'08)*, Jan. 2008.

[3] M. Gradinariu and S. Tixeuil. Conflict Managers for Self-Stabilization without Fairness Assumption. In *Proc. 27th Int. Conf. on Distr. Comp. Systems (ICDCS'07)*, June 2007.

[4] S. Hedetniemi, S. Hedetniemi, D. Jacobs, and P. Srimani. Self-Stabilizing Algorithms for Minimal Dominating Sets and Maximal Independent Sets. *Computers and Mathematics with Applications*, 46(5–6):805–811, Sept. 2003.

[5] S. Hedetniemi, D. Jacobs, and P. Srimani. Linear Time Self-Stabilizing Colorings. *Information Processing Letters*, 87(5):251–255, Sept. 2003.

[6] T. Herman. Models of Self-Stabilization and Sensor Networks. In *Proc. 5th Int. WS on Distr. Comp. (IWDC'03)*, Dec. 2003.

[7] H. Kakugawa and T. Masuzawa. Convergence Time Analysis of Self-Stabilizing Algorithms in Wireless Sensor Networks with Unreliable Links. In *Proc. 10th Int. Symposium on Stabilization, Safety, and Security of Distr. Syst. (SSS'08)*, Nov. 2008.

[8] N. Mitton, E. Fleury, I. Lassous, and B. Sericola. Fast Convergence in Self-Stabilizing Wireless Networks. In *Proc. 12th Int. Conf. on Parallel and Distr. Syst. (ICPADS'06)*, July 2006.

[9] M. Schneider. Self-Stabilization. *ACM Computing Surveys*, 25(1):45–67, Mar. 1993.

[10] V. Turau and C. Weyer. Fault Tolerance in Wireless Sensor Networks through Self-Stabilization. *Int. Journal of Communication Networks and Distr. Syst.*, 2(1):78–98, 2009.

[11] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proc. European Simulation Multiconference (ESM'2001)*, June 2001.

[12] K. Walther and J. Nolte. A Flexible Scheduling Framework for Deeply Embedded Systems. In *Proc. 21st Int. Conf. on Advanced Information Networking and Applications WSs (AINAW '07)*, 2007.

[13] C. Weyer and V. Turau. SelfWISE: A Framework for Developing Self-Stabilizing Algorithms. In *Proc. 16th ITG/GI - Fachtg. Komm. in Vert. Syst. (KiVS'09)*, Mar. 2009.

[14] Y. Yamauchi, T. Itou, G. Nishikawa, F. Ooshita, H. Kakugawa, and T. Masuzawa. Clustering Algorithm for Mobile Ad-Hoc Networks to Improve the Stability of Clusters. In *Proc. IASTED Int. Conf. on Sensor Networks (SN'08)*, Sept. 2008.