Self-Stabilization
Beyond the token ring circulation

Augusto Ciuffoletti
Dipartimento di Informatica - Università degli Studi di Pisa
mailto://augusto@di.unipi.it
http://www.di.unipi.it/∼augusto

April 13, 2001

Abstract

Condensing the definition of self-stabilization in a few temporal logic predicates highlights strengths and limits of this concept: we show that it enforces structural properties in the design of an algorithm, and prove that the original definition is inherently limited to token passing in a ring.

We complete our discussion introducing an extended definition motivated by the formal analysis.
Keywords: design of algorithms, distributed computing, fault tolerance, self-stabilization.

1 Introduction

The concept of self-stabilization [12] is extremely deep in pointing out the features of a fault-tolerant distributed system; however, there is not one reference definition, but diverse definitions are introduced, promoting some aspects and demoting others.

The purpose of this paper is too explain this fact: we argue that the concepts introduced by Dijkstra are valuable, but that the model introduced in the seminal paper is too restrictive. We use formal tools to establish a solid ground for our discussion, and find the limits of Dijkstra's model. Next we show that the model can be extended, without compromising the important concepts: we give an example for this, defining an extended model and a problem that could not be solved in the original model.

2 Understanding self-stabilization

The self-stabilization concept has been introduced in 1974 by E.W.Dijkstra: the original two pages long paper contains the model of a distributed computation,

and three examples. The model is in fact the distillation of a number of important concepts: taken as a whole, it describes a system that spontaneously recovers from transient failures, without apparent error detection and with an extremely reduced control. Many problems that characterize distributed fault-tolerant systems apparently melt down when the algorithm conforms to the self-stabilization model. Self-stabilization raises the interest of the theorician, that looks at the elegance of the algorithm, and of the practitioner, that finds a simple and robust solution.

Thus Dijkstra's model can be considered as a set of requirements: an system that exactly meets such *requirements* enjoys the *structural properties* of self-stabilizion: elegance and robustness. Let us summarize these requirements, as they are found in the original paper.

The *system layout* is a directed graph. Each *unit* operates according to a set of *guarded assignments*: the left term in an assignment corresponds to a local variable, while the guard, as well as the right term in an assignment, refer to the state of the unit and of its neighbors. The assignments whose guards evaluate to true are *enabled*.

The set of the *legitimate states* must satisfy four requirements (we add a label for future reference):

Legitimate Deadlock Freedom: in each legitimate state, one or more commands are enabled.

Stability: in each legitimate state each possible move will bring the system into a legitimate state.

Minimal Guards: each guard must be enabled in at least one legitimate state.

Minimal States: for any pair of legitimate states there exists a sequence of moves transferring the system from the one into the other.

In [24] the authors prove that the *Legitimate Deadlock Freedom* requirement is redundant, since it is a consequence of *Minimal States*.

A *self-stabilizing algorithm* must exhibit the following properties:

Finite Convergence: regardless of the initial state, the system is guaranteed to find itself in a legitimate state after a finite number of steps.

Deadlock Freedom: regardless of the initial state, at least one command will always be enabled.

As informally noted in [24], the last requirement is unnecessary: it is a consequence of *Legitimate Deadlock Freedom* and *Finite Convergence*. [1]

The *computational model* must satisfy the following requirement:

[1] If the system is bound to enter a legitimate state regardless of the initial state by the Finite Convergence, then none of the non-legitimate states can deadlock. Since Legitimate Deadlock Freedom requires that none of the legitimate states can deadlock, we conclude that Legitimate Deadlock Freedom and Finite Convergence imply that none of the states can deadlock, which coincides with the Deadlock Freedom statement.

Restricted Scheduling: at any time, any of the enabled assignements can be selected for the next move.

The scheduler "knows" which of the predicates is currently true, and arbitrarily selects one of them: the choice is therfore restricted to enabled statements.

The model is therefore made of three, tightly bound parts: the requirements on the set of legitimate states, the requirements on the algorithm, and the requirements on the execution model. We stress that the requirements encompass not only the algorithm, but the statement of the problem itself: limiting the way legitimate states are described limits the set of problems that can be modeled using self-stabilization. A formal proof of this fact is one of the contributions this paper.

This explain why, except in a few cases (as in [4, 6, 11, 14]), authors change the definition of self-stabilizing system, in order to fit the problem they want to solve: to cope with permanent failures [7], to implement a system clock [15, 19], to implement mutual exclusion in trees [23], or to color graphs [17]. However, in these cases there is no mention that the model is modified, and it is not clear whether the change is really needed, and what share of the *structural properties* survives.

Only a few papers formalize the alternate definition: it turns out to be either an important relaxation (considering primarily stability [18], or weakening the fairness requirements in favor of a probabilistic attitude [22]), or the reduction to a degenerate case (like [17], where legitimate states are the fixed points of a finite computation). However, even when the alternate definition is explicitely introduced, the authors do not explain if the new definition enjoys the same *structural properties* of the old one.

A limited number of papers introduce fault-tolerance paradigms related with self-stabilization, without addressing a specific problem. As a general rule, they have an attitude centered on the their original contribution, and do not take into account how much of the old concept is kept, and how much is traded off. This is the case of [5], that formalizes and compares the concepts of *stabilization* and *pseudo-stabilization*; both concepts differ from the original concept of self-stabilization, and are not explicitely compared with it. In [2] the authors mention the relationship between the paradigm of *closure and convergence* and self-stabilization, but do not compare further the two approaches.

Summarizing, authors often modify the *requirements* for self-stabilization, but they fail to say why, and how much of the *structural properties* associated to the original concept are preserved, and we suspect that extensions often lend to a substantial loss of structural properties. Using well known facts as a metaphore, let us consider a computer language without repetitive statements: its programs are very clean and understandable, but it has a limited applicability. We may extend it either adding labels and `goto` statements, or introducing a `while` construct. In both cases we obtain a more powerful language, but in the former we would compromise the structure of the programs (at least of badly written ones). Our case is similar: Dijkstra's model is sharp, but too restrictive, and

we want to extend it without losing sharpness.

We focus on this point, and we obtain two basic results:

- classify the systems that conform to Dijkstra's requirements, and
- find a wider class sharing similar structural properties.

To reach the first result, we need to formally define the requirements, and obtain a description of the class of systems that correspond to these specifications. We conclude that the set of systems that comply with Dijkstra's definition is very limited.

To obtain the second result, we need to introduce an extension and to evaluate its structural properties. The first step in this direction is to understand which are the structural properties we are interested in. This is an informal step, and the reference paper is [24]. Next we formalize and evaluate an extension, which is in fact a common factor of many extensions that can be found in the literature (although never stated formally). We conclude arguing that it preserves the structural properties of the original idea, but we still need to prove that the extension really extends the original definition. To this purpose, we introduce a simple system: it witnesses that there are systems that do not satisfy Dijkstra's requirements for self-stabilization, but are such in the extended definition.

3 Formalizing the concept of self-stabilization

We restate self-stabilization as follows:

- translate the syntactical framework given by Dijkstra into Unity statements;
- prove that the execution model induced by Dijkstra's daemon is equivalent to the model induced by Unity's scheduling rule;
- map the informal requirements of a self-stabilizing system into a Unity definition of a self-stabilizing program.

3.1 Mapping the syntax

The **initially** section of the Unity program that describes a self-stabilizing system is necessarily empty, since the definition requires stabilization not to be limited by the initial state: therefore we cannot assume an initial value for the variables.

The guarded commands described in [12] can be immediately translated into corresponding Unity conditional statements and quantified statements containing conditional statements.

In order to make significant some of the indications in Dijkstra's paper, we require that conditional statements are not nested: a conditional statement is

composed of a predicate, the *guard*, and of a sequence of statement that does not contain conditional statements, the *guarded assignement*. In this sense we say that we restrict to "flat" Unity programs. This requirement is not restrictive of the power of the language: any nested conditional statement can be rewritten as a (longer) non-nested conditional statement.

The binding between a unit and the set of commands affecting that unit is described as a *mapping* from the program to the architecture. This step implements the allocation of the variables to the *units*. We recall that in order to respect the distributed nature of the algorithms, a unit has read-only access to the variables of adjacent units. This requirement limits the range of admissible mappings for a given program.

3.2 Mapping the execution model

The execution model introduced in Unity associates a set of execution sequences with a program. An *execution sequence* consists of a possibly infinite list of pairs (x_i, s_i) where x_i is the index of a statement, and s_i is a state. Each pair is obtained from the state of the previous pair, by applying statement x_i to the state s_{i-1}. The initial state s_0 is chosen from those that satisfy the **initially** section.

We can give a similar execution model which adheres to the one informally defined by Dijkstra, and this model is fair if, following the indication in the first part of Dijkstra's paper, we introduce a daemon as an entity which "models the undefined speed ratios of the various machines". If speed ratios are considered finite (although undefined), a form of fairness is implicitly introduced: a statement cannot remain enabled forever without being executed.

3.3 Formalizing the concept of self-stabilization

Now that we have justified the use of Unity in order to describe self-stabilization, we can use it to describe the properties informally introduced in Section 2.

The property of being self-stabilizing pertains to a pair (F, l) where F is a program and l is a predicate that characterizes legitimate states. Thereafter the infix notation 'F self-stabilizes to l' is used, and the set of predicates $\Pi = p_1, \ldots, p_v$ stands for the set of the guards in the program.

We will now define the properties that characterize the set of legitimate states, with reference to the statements highlighted in the previous section, and explain their part in the self-stabilization concept.

Definition 1 (Stability) *l is stable*

We recall that '*stable*' indicates a property of a predicate that, once established, is satisfied forever ([8]). This is a characterizing property: once the state is correct, it remains such.

The role played by the two properties that follow is less apparent, but characterize the self-stabilization as a *software engineering* principle. Although they

reflect the fairness of a token circulation algorithm, they also say that the design does not contain exceptional items.

Definition 2 (Minimal Guards) $\forall p \in \Pi, \{l \wedge p\} \neq \emptyset$

This requirement asserts that the program does not contain guards that are satisfied only if the system is in a non-legitimate state. These predicates would roughly correspond to *detectors* in the terminology introduced in [3].

One of the exceptional features of the self-stabilization concept is that it introduces at the same time functional and structural properties of the algorithm; the *Minimal Guards* property reflects this, and entails some important structural consequences:

no error detection: the absence of recovery routines means that there is no need of local error detection, which would trigger the execution of such routines. The implementation of error detection is especially difficult in distributed system, where the knowledge of the global state, which is needed to detect failures, is impossible, or extremely expensive [9];

no cost fault tolerance: when the state of the system is legitimate, no share of the activity of the system applies to the implementation of the fault tolerance;

no "by case" design: error recovery (especially forward error recovery) often requires the enumeration of different exceptional patterns, and related recovery actions. The resulting design is poorly structured, its coverage is difficult to test, and hides bugs that perform only when an emergency raises. The *Minimal Guards* requirement explicitly excludes this.

Definition 3 (Minimal States)

$$\forall s_1, s_2 \in \{l\}, s_1 \mapsto s_2$$

The above definition should be equivalent to the statement: "for any pair of legitimate states there exists a sequence of moves from the one to the other". Indeed, the informal statement can be interpreted in two ways: the transition from s_1 to s_2 is performed if 1) the daemon enables the units in an appropriate order, or 2) independently from the behavior of the daemon. The first interpretation does not guarantee a sort of fairness: a legitimate state might be left unvisited forever, if the daemon never follows the appropriate strategy. This consequence is serious if, as in the examples given by Dijkstra, each legitimate state must be necessarily visited, since each legitimate state is relevant for the behavior of the system. Therefore we follow the second interpretation, and formalize this statement using the "eventually" concept embedded in the $\mapsto$ relation.

This interpretation introduces another structural property of the self-stabilization concept: *none of the legitimate states warns about a fault that might cause a failure*. Consequently the daemon cannot anticipate failures by driving the system into a fault related legitimate state, since this one would be an instance of

a state that may or may not be reached, depending on the occurrence of the fault; this fact is excluded by the *Minimal States*.

This implies that the daemon (i.e., the underlying abstraction levels) cannot seamlessly drive the system in a *fault recovery* state: in fact, such daemon should have some fault detection strategy, and this is what the self-stabilization concept wants to exclude. Like the *Minimal Guards*, but at a different abstraction level, this property addresses completely decentralized designs where global error detection mechanisms are inapplicable.

Definition 4 (Finite Convergence) $true \mapsto l$

Together with *stability*, this is a strongly characterizing property: the system always tends to a *legitimate state*, independently from the current state.

We are now able to give a definition of a self-stabilizing system, which is the literal translation of the original definition into a formal language:

Definition 5 (Self-stabilizing System) *Given a (flat) Unity program P_U with guards Π and a predicate l, P_U is self-stabilizing to l if and only if:*

- *the* **initially** *section of P_U is empty and*
- *l is stable and*
- $\forall p \in \Pi, \{l \wedge p\} \neq \emptyset$ *and*
- $\forall x, y \in \{l\}, x \mapsto y$ *and*
- $true \mapsto l$

The following theorem states that a very limited set of problems may be solved by a self-stabilizing algorithm:

Theorem 1 *A predicate l satisfies* Stability *and* Minimal States *if and only if*

1. *only one state satisfies l and the execution of any statement leaves that state unaltered or*
2. *there are $n > 1$ states that satisfy l and an indexing $\{l\} = \{s_0, \ldots, s_{n-1}\}$ exists such that s_i ensures $s_{(i+1) \bmod n}$.*

The proof is available in [10].

Summarizing, if the system has more than one legitimate state, the states can be arranged in a ring, so that the system moves from one state to its successor, in an endless loop.

In each state there is exactly one guard that is satisfied, since more than one guard enabled would mean more than one successor, and no guards enabled would mean deadlock: each statement has the effect of enabling exactly one other statement.

If we map each statement on a different unit, the layout of the system is a ring: in that case self-stabilization is limited to token passing algorithm on that

ring. If we map more than one statement on each unit, we obtain that the token is passed according to a virtual ring: in that case each node is visited as many times as are the statements mapped on that unit.

We conclude here the discussion about the restrictions introduced by Dijkstra's model: this model is bound to describe a token passing on a ring, with minimal variations on that theme. One practical consequence is that self-stabilization is incompatible with path redundancy, which is an important drawback of a scheme addressing fault-tolerance.

4 Unfurling self-stabilization.

The point with Dijkstra's definition of self-stabilization is that it tightly restricts and characterizes both the problem and the solution: not only the solution must respect certain requirements, but the problem itself must exhibit certain features.

The previous section concludes that Dijkstra's description of self-stabilization is exactly shaped on the circulating token problem. Indeed, this conclusion coincides with the intentions of the author, as explained in [13], but presently this feature is an undesirable restriction to a more powerful concept.

Theorem 1 precisely indicates where this restriction is introduced: the *Stability* and *Minimal States* requirements alone induce the limitation to a ring. Since the *stability* requirement strongly characterizes self-stabilization, we focus on the *Minimal States* requirement and explore the possibility to extend it, thus widening the applicability of the self-stabilization concept. We opt to redefine *Minimal States* so that it operates on set of states, instead of single states. A set of *properties* L is given such that each *legitimate state* satisfies one of them, and for any pair of *properties* in L there exists a sequence of moves transferring the system from one to the other.

This simple modification alone eliminates the strong limit expressed by Theorem 1. The new *Minimal States* is:

Definition 6 (Minimal States (extended)) *A class* $L = \{l_1, ..., l_k\}$*, with* $\bigcup_L = l$ *exists such that:*

$$\forall (l_a, l_b) \in L^2,\ l_a \mapsto l_b$$

The following section is dedicated to a simple algorithm admitting redundant paths that is self-stabilizing according to the extended *minimal states* introduced above, but is not self-stabilizing according with the original definition.

5 Example: the "tick-tack" algorithm

The constructive proof that the class of problem-solutions is in fact widened is obtained finding a pair that cannot be described as a pure self-stabilization instance, but complies with the extension mentioned above.

The introduction of a new algorithm is not strictly needed: algorithms have already been presented ([23, 1, 16, 19, 20, 21]) that would indeed fit our purpose, since they implicitly use the extension we propose. But we prefer to introduce a simpler algorithm, exactly shaped to our needs.

We want to design an algorithm that controls a number of units so that they move in lockstep: this means that none of them can engage in the successive step until all other units have committed the previous. Each move is therefore identified by an integer number in the interval $[0..N]$, and is divided in two steps: an operation and its commitment.

To support this requirements, we introduce an internal *clock* register in each of the units records the present state as an integer: the operation of the i-th step corresponds to state $2 * i$, its commitment to state $2 * i + 1$ (here and in the following the increment is intended modulo the number of states). In addition, two other units are introduced: in figure 1 we have depicted a scenery with three controlled units: B_1, B_2, and B_3. Units A and C are used to enforce the lockstep progress of B units.

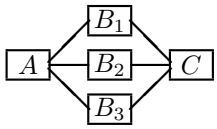

Figure 1: Layout of the system

The predicate that characterizes the legitimate state (i.e. lockstep operation) is the following:

- the difference between the clocks of A and C is ± 1;
- the clocks of the units $B_k, k \in [1..3]$ have a value chosen among those of A and C.

The mutual states of the units A and C are easy to enumerate, since they must move in turn: $A = C - 1$ alternates with $A = C + 1$, with A always odd and cycling in the admitted range of values: since this alternation recalls the move of a pendulum, we say that a state with $A = C + 1$ is a *tick* state, and the others are a *tack* states. The definition of our problem implies that any couple of values for A and C such that A is odd, and C is one of the even numbers adjacent to A must be eventually reached. But the state of the system contains also the state of the B's.

For these units the requirements are fuzzier: their value must be equal to A's or C's, but there is no priority among them, and all possible configuration are in fact equivalent for the informal requirements.

The fuzzy requirement introduced above does not match the original, exhaustive definition of *Minimal States*: we do not want that *every* legitimate

state is eventually visited. Instead, we want that each member set of a partition of legitimate states is eventually visited; all the individual states that are equivalent from the point of view of the requirements are grouped in the same member set of the partition.

The members of the partition are defined by the following predicates:

$$tick_i = \{s \mid A = 2i - 1 \wedge A = C + 1 \wedge (\forall k : B_k = A \vee B_k = C)\}$$
$$tack_i = \{s \mid A = 2i - 1 \wedge A = C - 1 \wedge (\forall k : B_k = A \vee B_k = C)\}$$

Intuitively,

- while the system is in a $tick_i$ state, all the B units have either committed or performed the $(i-1)$-th step,
- while the system is in a $tack_i$ state, some of the B units have committed the $(i-1)$-th step, while the others have already performed the successive step.

Our *Extended Minimal States* property is defined for the class

$$L = \{tick_i, tack_i \mid i \in [0, N]\}$$

where N is the number of possible states of the global clock.

It is possible to prove that the following algorithm is self-stabilizing, and that it satisfies the *Extended Minimal States* property. The proof is available in [10].

```
A:
if (B1==B2) and (B2==B3) and even(B1) A=B1+1;
if (B1==B2) and (B2==B3) and odd(B1) A=B1;

C:
if (B1==B2) and (B2==B3) and even(B1) C=B1;
if (B1==B2) and (B2==B3) and odd(B1) C=B1+1;

Bk:
if A!=(C-1) Bk=A;
else Bk=C;
```

6 Conclusions

The concept of self-stabilization defined by E.W.Dijkstra in [12] has been analyzed, and a motivated extension has been proposed.

We have shown that the original definition can describe only cyclic token passing schemes: under this restriction, there is no room for more adaptive scheduling policies.

We have proposed a weaker definition that extends the applicability of self-stabilization to a wider class of concurrency problems, that do not need strictly cyclic scheduling, but lockstep operation, as indicated by the final sample algorithm.

References

[1] Y. Afek and G.M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1):27–34, 1993.

[2] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993. Special Issue on Software Reliability.

[3] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *IEEE Transactions on Software Engineering*, page to appear, 1999.

[4] G.M. Brown, M.G. Gouda, and C.L. Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[5] J.E. Burns, M.G. Gouda, and R.E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1):35–42, 1993.

[6] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[7] Richard W. Busken and Jr Ronald P. Bianchini. Self-stabilizing mutual exclusion in the presence of faulty nodes. In *Proceedings of the 25th Fault Tolerant Computing Symposium*, pages 144–153, Pasadena, California (USA), 1995.

[8] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, 1989.

[9] K.Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 1985.

[10] Augusto Ciuffoletti. Appendix of this paper.
URL `ftp://ftp.di.unipi.it/pub/Papers/ciuffoletti/ss.ps`.

[11] Adam M. Costello and George Varghese. The FDDI MAC meets self-stabilization. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems – Workshop on Self-Stabilizing Systems*, pages 1–9, Austin – Texas, May 1999.

[12] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[13] Edsger W. Dijkstra. Self-stabilization in spite of distributed control. In David Gries, editor, *Selected writings on Computing*, pages 41–46. Springer Verlag, 1982.

[14] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[15] Shlomi Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, (12):95–107, 1997.

[16] M. Flatebo, A.K. Datta, and A.A. Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8(3):133–142, 1995.

[17] S. Ghosh and M.H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, 1993.

[18] M.G. Gouda and N.J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[19] Mohamed G. Gouda and Ted Herman. Stabilizing unison. *Information Processing Letters*, 35(4):171–175, 1990.

[20] T. Herman. Self-stabilization: randomness to reduce space. *Distributed Computing*, 6(2):95–98, 1992.

[21] S.-T. Huang and N.-S. Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7(1):61–66, 1993.

[22] A. Israeli and M. Jalfon. Token management schemes and random walks yield self stabilizing mutual exclusion. In *Proceedings of the Ninth Annual ACM Symposium on Distributed Computing*, pages 119–129, Quebec City, Quebec, Canada, August 22-24, 1990.

[23] H.S.M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, 1979.

[24] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 26(1):45–67, 1993.