An Exercise in Proving Convergence through Transfer Functions

Oliver Theel and Felix C. Gärtner*
Darmstadt University of Technology
Department of Computer Science
D-64283 Darmstadt, Germany
Email: {theel | felix}@informatik.tu-darmstadt.de

Abstract

Self-stabilizing algorithms must fulfill two requirements generally called closure *and* convergence. *We are interested in the convergence property and discuss a new method on proving it. Usually, proving the convergence of self-stabilizing algorithms requires a well-foundedness argument: briefly spoken, it involves exhibiting a* convergence function *which is shown to decrease with every transition of the algorithm starting in an illegal state. Devising such a convergence function can be a difficult task, since it must bear in itself the essence of stabilization which lies within the algorithm. In this paper, we explore how to utilize results from* control theory *to proving the stability of self-stabilizing algorithms. We define a simple stabilization task and adapt stability criteria for linear control circuits to construct a self-stabilizing algorithm which solves the task. In contrast to the usual procedure in which finding a convergence function is an afterthought of algorithm design, our approach can be seen as starting with a convergence function which is implicitly given through a so-called* transfer function. *Then, we construct an algorithm around it. It turns out that this methodology seems to adapt well to those settings which are quite difficult to handle by the traditional methodologies of self-stabilization.*

Keywords: Self-Stabilization, Convergence, Control Theory, Transfer Function

1. Introduction

The design and verification of self-stabilizing algorithms is a difficult task. This is mostly due to the convergence property of the self-stabilization paradigm: starting from *any* state, the algorithm must reach a predefined set of legal states in finite time. Usually, a proof of convergence consists of a well-foundedness argument: a total function from the program state into a well-founded set must be given, which decreases with every program transition when starting in an illegal state.

Building on this idea, research in the past years has provided algorithm designers with a set of methods which help design and verify the convergence property more easily. Almost all of them are based on a notion of composition: Let A_1 and A_2 be two algorithms, A_1 self-stabilizes to a predicate P_1 and A_2 self-stabilizes to a predicate P_2 once P_1 holds. It can now be shown, that the parallel execution of A_1 and A_2 self-stabilizes to P_2 provided that A_2 does not interfere with the convergence of A_1 [1, 2]. The idea behind this fact is that the function which exhibits the convergence of the composed algorithm can be easily constructed by composing the convergence functions of A_1 and A_2 in the following way: let f_1 and f_2 be the convergence functions of A_1 and A_2 respectively, then (f_1, f_2) is a convergence function of the composed algorithm (using the lexicographic ordering as well-founded ordering relation). The composition of self-stabilizing algorithms has proven to be a powerful design paradigm, leading to very elegant solutions to complex problems (see for example the distributed reset algorithm by Arora and Gouda [3]).

Refined versions of the composition of self-stabilizing algorithms have been presented: among them are Gouda and Multari's idea of a *convergence stair* [4] and the notion of a *dependency graph* by Arora et al. [5]. All of them rely on the fact, that the convergence functions which exhibit the convergence of the composed algorithms are already given. However, finding these functions for given base algorithms is probably the most difficult task in proving the self-stabilization property.

Usually, algorithm designers construct an algorithm and then try to prove it correct. If the first proof attempts reveal flaws in the algorithm, it might go through several refinement cycles until the designer is sure, that it works. But finding a working convergence function at this stage poses

*This author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.

problems since such a function must bear the very essence of the stabilization property of the algorithm.

In this paper we explore some different paths in designing and verifying self-stabilizing algorithms: instead of constructing an algorithm and searching for a proof, our approach can be seen as starting with a convergence-like function and then constructing an algorithm around it. By doing this, we continue our recent work on adapting results from the area of *control theory* to prove the stability of distributed algorithms [6].

At the moment, the benefits of our approach are obvious in some restricted application scenarios. But it turns out that these are in fact the settings which are quite difficult to handle by the traditional methodologies of the self-stabilization domain.

The paper is structured as follows: In Section 2 we present and precisely define the stabilization task at hand. In Section 3 we analyze the problem and adapt results from control theory to first devise a transfer function and then derive an algorithm from it. Section 4 discusses some of the issues involved including the merits and drawbacks of our approach. Finally, Section 5 concludes the paper.

2. The Stabilization Task

Consider the simple system in Figure 1. It consists of an automaton which reads its input from a communication channel *in* and writes output to a channel *out*. While nothing is known about the internals of the system, the input/output behavior is well-defined, known, and of the following type: the system encapsulates a form of load which can be read via the *out* channel. It is given as a number. Once the system receives a positive signal over the input line, it begins to produce load items and thus increment the local load. On the other hand, if a negative signal is received over *in*, the system begins to destroy load items and thus decrements the internal load. More precisely, the automaton can be described as shown in Figure 2. The upper graph shows a temporal behavior of the input channel to the automaton, indicated by $in(t)$. Setting *in* to 1 results in an immediate receipt of a single load unit and also in triggering the continuous sending and reception of load units. Assume that a load unit is sent and received after every fixed real time interval T_0, then after $k \cdot T_0$ time, *out* will read the value of $1 + k$, k being a natural number. In general, setting *in* to some integer value u for $k \cdot T_0$ time will result in $u \cdot (1 + k)$ additional load units after $k \cdot T_0$ time (note that u may also be negative). Changing *in* can thus be seen as controlling the speed of load unit collection.

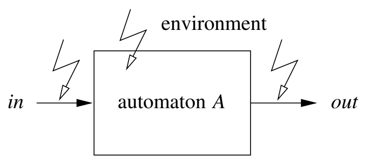

Figure 1. A simple system to be stabilized.

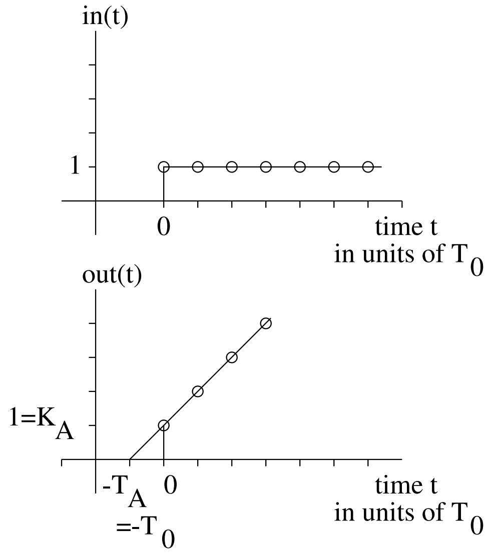

Figure 2. Behavior of automaton A.

Automaton A can be influenced by the environment resulting in a spontaneous change of the local load, indicated by the flashes in Figure 1. Sources of such changes can be internal transient faults, corrupted messages received via *in* or a maliciously altered output *out* or any other external change of the local load.

An automaton like the one depicted in Figures 1 and 2 is not untypical for real systems: the given characteristics of A can be due to the fact that the component is implemented in hardware or given as a binary. Also, the simple behavior of the system closely resembles the possible actions of small components of mobile or embedded systems, which usually trade complex behavior for low power consumption and minimal space requirements.

The actual behavior can be interpreted in different ways. For example, the automaton could be a reseller of tickets for which a necessary final ingredient for ticket production must be received via *in*. The value of *out* would simply be the amount of tickets currently available. Another example is that the automaton could be the controller of an electric motor with the value of *in* being the intended change to the accelerator setting and *out* being an indication of the current speed.

The stabilization task at hand is to design and verify a self-stabilizing algorithm around A which, given a fixed target value w, guarantees that the local load within A always stabilizes to w once the spontaneous changes of A's local load caused by the environment have stopped. Formally this can be re-formulated using the usual requirements of self-stabilization [7]:

- (closure) If the local load of A has reached w, then it will not change in the future.
- (convergence) Starting from an arbitrary state in which the local load is not w, the system will reach a state in which the local load is w in finite time.

3. Constructing a Solution

We use a simple sensor/actuator approach to tackle the problem. By doing this, a *sensor* is added to the *out* channel of A. This component is used to "observe" the local load value by listening to *out*. The observed value is subsequently passed to an *actuator*, which resides at the *in* channel of A. It is used to influence the local load by passing information to A via *in*. The configuration is shown in Figure 3. In our scenario, the sensor is very simple: the only thing it does is to pass the information received via *out* directly to the actuator. By comparing the target value w and the information given by the sensor, the actuator now must calculate an input to A which will eventually lead to a stable load count equal to w.

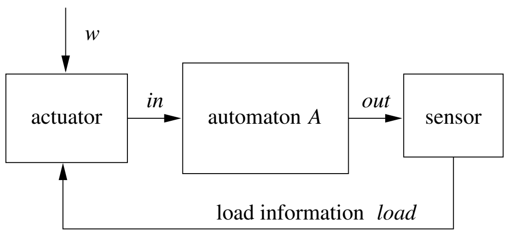

Figure 3. Placement of actuator and sensor around A.

We will now use results from control theory to derive such an implementation of the actuator. For this, we need some information of the output-to-input behavior of A. With respect to time, A's output-to-input behavior can be denoted as

$$G_A(t) = \frac{out(t)}{in(t)} \tag{1}$$

This is called the *(time) transfer function* of A. Key to deriving $G_A(t)$ is the following observation: Assume that the load count of the automaton was 0 at some initial time $t_0 = 0$ and that the changes of *in* over time are given by some function $in(t)$. Assuming a continuous flow of time, the behavior of A is fully specified by the following equation:

$$out(t) = in(t) + \int_0^t in(t)\,dt \tag{2}$$

Equation (2) could either be solved directly (which may require some effort, in particular in a general case), or solved by using *Laplace transforms* [8].

In the scope of this paper, we are not concerned with deriving $G_A(t)$ due to the following two reasons: First, computer systems are discrete event systems where time also is discrete. The description of modules must address this fact. Therefore, we assume a sampling time interval of T_0, i.e., the various components are at most active at points $k \cdot T_0$ in time with k being a natural number. Second, we can use a different description of A's output-to-input behavior which takes discrete time into account and additionally allows for an easier handling of A when interacting with other modules like the sensor and the actuator. This description of A can be derived from Equation (2) leading to a so-called *z-transform* of A's output-to-input behavior:

$$G_A(z) := \frac{out(z)}{in(z)} = \frac{1}{1 - z^{-1}} \tag{3}$$

Automaton A interacts with two other modules: actuator and sensor. Their internal structures — specified by components as used in control theory — are shown in Figure 4. The left part of the figure shows the internals of the actuator. It has two input channels *w* and *load* and one output channel *out*. Usually, *w* will remain fixed to a specific value, but in the general case it may change over time. The target number *w* first serves as input to a submodule P which simply multiplies any value of *w* by some constant K_P. The incoming *load* signal is then subtracted from the resulting product $K_P \cdot w$ prior to being passed on to a submodule R. The submodule R encapsulates the essence of the behavior of the actuator. It can be seen as a function which maps the target value w (subtracted by the current *load*) to an input

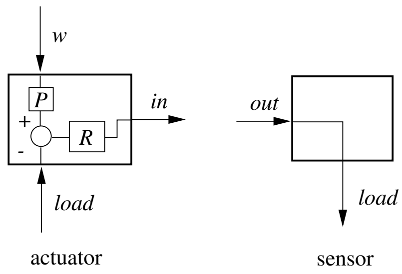

Figure 4. Implementation of actuator and sensor

value for *in*. The stabilization task is thereby reduced to deriving a "good" transfer function $G_R(z)$ of R which renders the entire system stable.

The right part of Figure 4 shows the implementation of the sensor. As already indicated, the implementation of the sensor is quite simple: the sensor passes the input *in* directly to the output *load*. This leads to

$$G_S(t) = \frac{\mathit{load}(t)}{\mathit{out}(t)} = 1 \qquad (4)$$

Due to the properties of the z-transform, the z-transform of the sensor's transfer function $G_S(z)$ is also equal to 1.

In a last step prior to constructing a suitable transfer function for R, it must be stated how the input w affects the output *out* of the entire system. This is done through the following transfer function

$$G(t) = \frac{\mathit{out}(t)}{w(t)} \qquad (5)$$

which is defined by $G_A(t)$, $G_R(t)$, K_P, and the way these components interact with each other. For the same reasons as already stated when discussing $G_A(t)$, we subsequently use the z-transformed transfer function of $G(t)$, namely $G(z)$. By analyzing the flow of information from w to *out*, as given in Figures 3 and 4, $G(z)$ calculates to

$$G(z) = \frac{\mathit{out}(z)}{w(z)} = K_P \cdot \frac{G_R(z) \cdot G_A(z)}{1 + G_R(z) \cdot G_A(z) \cdot G_S(z)} \qquad (6)$$

Finally, in oder to fully implement the actuator, a particular implementation for submodule R must be given. As indicated above, R must be chosen such that once a particular value for w is given, the system stabilizes to some value. This will guarantee that the entire arrangement, i.e., together with a suitable setting of K_P, stabilizes to $w = \mathit{out}$.

A result from control theory [8] gives us sufficient conditions how to devise $G(z)$ (and therefore $G_R(z)$) so that *out* will converge to w, thereby solving the stabilization task: If all poles of the z-transformed transfer function $G(z)$ lie within a circle of radius 1 around the origin of the z-plane (the y-axis is imaginary, the x-axis is real), then the entire system will stabilize for any value of w

In general, the behavior of an automaton like A can be specified using different parameters for the initial value K_A ($K_A = 1$ in the example) and the slope K_A/T_A of the "load curve" in Figure 2. Also, the sampling interval T_0 must be specified ($K_A/T_A = \frac{1}{2}$ and $T_A = T_0 = 2$ in the example). If concrete values for these parameters are not given, it is possible to construct general solutions which depend on them. In such general cases, $G_R(z)$ and some adequate settings for T_0, T_A, and K_A must be found such that the before-mentioned stability condition holds. Some values can also be pre-defined by the implementation of the automaton A (like in our example T_A, T_0, and K_A). In such a case, the designer of $G_R(z)$ is confronted with a harder problem. Nevertheless, in most cases, multiple solutions exist. They differ in complexity, convergence time and robustness against parameter degradation.

Although solutions can be constructed by making an ansatz for $G_R(t)$, for example, a linear differential equation of order 2, the more convenient method is to simply take advantage of standard components which are known to bear the potential of stabilizing certain other components, like, e.g. automaton A. One of those standard component is a so-called PT_1 *regulator*. It combines *proportional* and *time differential* properties of order 1. The linear differential equation of a PT_1 regulator is given by Equation (7):

$$T_1 \dot{y}(t) + y(t) = K_R x(t) \qquad (7)$$

Here, $x(t)$ is the input signal and $y(t)$ represents the output signal of the component. Assuming a sampling interval T_0, the z-transform of its transfer function calculates to

$$G_R(z) = \frac{y(z)}{x(z)} = K_R \cdot \frac{\left(1 - \exp(-\frac{T_0}{T_1}\right) z^{-1}}{1 - \exp(-\frac{T_0}{T_1}) z^{-1}} \qquad (8)$$

One solution, for example, is obtained by the following setting.

$$\begin{array}{ll} T_0 = 2 \text{ time units}, & K_R = 1 \\ T_1 = 2 \text{ time units}, & K_A = 1 \\ T_A = 2 \text{ time unit}, & K_P = 1 \end{array}$$

In this case, the resulting transfer function is

$$G(z) = \left(1 - \frac{1}{e}\right) \cdot \frac{z}{z^2 - \frac{2}{e}z + \frac{1}{e}} \qquad (9)$$

with $e = \exp(1)$. By analyzing $G(z)$, we find that it has two poles $z_{1,2} = -1/e \pm \sqrt{1-e}/e = -0.36 \pm 0.48i$. Since both poles lie within distance 1 from the origin of the z-plane, the entire arrangement exhibits a stable behavior.

Figure 5 shows the stabilization of the *out* signal to the value of the *in* signal over time.[1] In the interval $[0T_0, 50T_0)$ *in* was set to 5 and at time $50T_0$ it was changed to 3.

By transforming $\mathit{out}(z) = G(z) \cdot w(z)$ into the (discrete) time domain, the input/output behavior of the entire system is given by

$$\mathit{out}(nT_0) = \sum_{k=0}^{\infty} g((n-k)T_0) \cdot w(kT_0) \qquad (10)$$

with $g(kT_0)$ being

$$g(kT_0) = \frac{\sqrt{1-e}}{2} \left\{ \left(\frac{1}{\sqrt{1-e}+1}\right)^{kT_0} - \left(-\frac{1}{\sqrt{1-e}-1}\right)^{kT_0} \right\}$$

[1] The graph shows the results obtained by a MATLAB simulation. Due to the avoidance of algebraic loops, A's output is delayed by T_0 time.

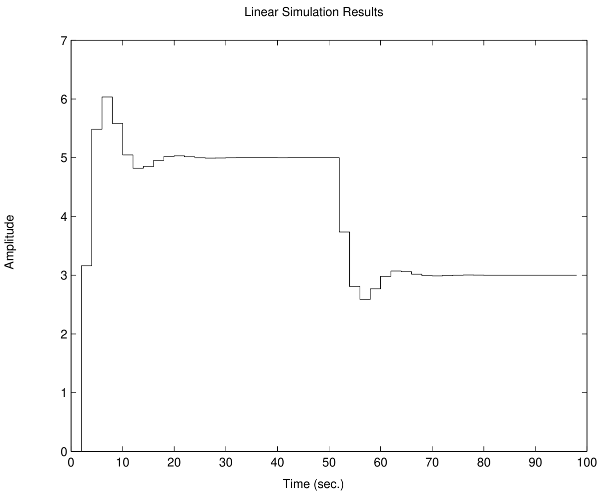

Figure 5. Asymptotically stable behavior of the scenario

Equation (10) can in fact be regarded as a convergence function of the system, since for any input function

$$w(nT_0) := \begin{cases} \text{arbitrary} & nT_0 < n_0T_0 \\ c & nT_0 \geq n_0T_0 \end{cases} \quad (11)$$

$out(nT_0)$ asymptotically stabilizes to c. Note that by using this methodology, stabilization is guaranteed not only for a certain value but for a value of c which can be freely chosen.

We note in passing that environmental influences can be viewed as transient faults and the automaton A alone can be regarded as a component which does not tolerate these faults. This is because the local load may deviate from w forever if no correcting action is taken. Thus, the stabilization task described in the previous section can be seen as a fault tolerance problem. In this view, our solution has many similarities to solutions constructed using Arora and Kulkarni's methodology of *detectors* and *correctors* [9].

4. Discussion

There are many objections that can be brought forward when discussing the relevance of our approach. The first one is that the problem becomes seemingly trivial to solve if it is viewed as the question whether a simple "while" loop terminates or not. This is depicted in Figure 6. From this perspective, the integrating features of the automaton A result in a simple addition of its input value i to the current load n. The stabilization task in this setting can be formulated: "design a function g for which the loop provably terminates." An *ad hoc* solution would design $g(n, w) = w - n$. With this version of g it can be shown that

variables:
  n integer $\{*$ load count, can be arbitrary $*\}$
  w integer $\{*$ target value, arbitrary but fixed $*\}$
  i integer $\{*$ can be arbitrary $*\}$

while $n \neq w$ **do**
  $i := g(n, w);$ $\{*$ "actuator" $*\}$
  $n := f(n, i);$ $\{*$ here: $f(n, i) = n + i$ $*\}$
end

Figure 6. Reformulation of the stabilization task in terms of a while loop.

the loop terminates after a single cycle. While being correct and simple, there are still some problems with this solution. For example, the solution is not directly transferable to other possibly more complex versions of f. For instance, if the system restricts the values which may be input to f by limiting the value of i to at most 2, then the *ad hoc* solution fails.

On the other hand, it is interesting that the above solution can be rephrased in terms of control theory by applying a regulator whose transfer function contains a "spike" of height $w - n$ at any point in time where the sensor reads $w \neq n$. The z-transform of such a regulator $G'_R(z)$ is

$$G'_R(z) = 1 - z^{-1} \quad (12)$$

Interestingly, using regulator $G'_R(z)$ in the sample system of Section 3 leads to a transfer function of $G(z) = \frac{1}{2}$. Changing K_P from 1 to 2 finally results in a transfer function of $G(z) = 1$. This can be interpreted as a system which immediately adjusts the output signal *out* according to the input signal w. Figure 7 shows such an input/output behavior.[2] It should be noted that regulators like $G'_R(z)$ are not met in practice since physical systems usually are subject to inertia and do not respond well to such inputs.

This leads to a second possible objection: Although some application scenarios were given in Section 2, the automaton A and the whole stabilization task is rather unrealistic. We admit that there is a grain of truth in this objection: components like the automaton A which suffer from undesirable inertia and are so difficult to control are not as commonplace in distributed systems as it might have been implied. However, this does not alleviate the fact that such components actually can be constructed. And if they are constructed, engineers must find ways to control them in a stabilizing way.

The third objection concerns the relevance of the problem scenario to the area of self-stabilization and also

[2] Due to algebraic loops the system's output is delayed by T_0 time in the presented MATLAB simulation.

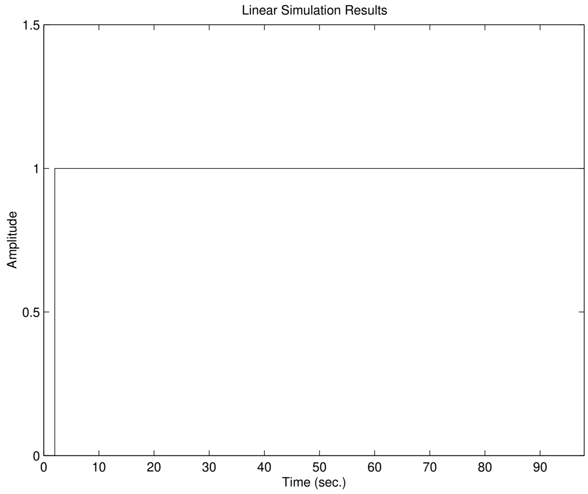

Figure 7. Asymptotically stable behavior of the scenario using regulator $G'_R(z)$

touches the main point we want to make: The stabilization problem described in Section 2 using e.g. a PT_1 regulator as in Section 3 is clearly one which falls into the self-stabilization domain. Yet, all approaches we have explored using standard methodologies from the self-stabilization community [10, 11, 12, 13, 14, 15] have failed miserably [6]. We conjecture that finding direct solutions to the problem is so difficult because of the necessary cyclic dependencies within the compositional structure of possible solutions. For example, an implementation of a similar algorithm in a different setting contains a dependency graph between program variables of length three [6]. While coming close to being able to handle such cyclic interrelations [5], we know of no "traditional" approach that can handle our problem scenario effectively.

Moreover, the control theory approach offers the desirable feature to be able to estimate the rate of convergence. We find that this is an interesting property because the area of stabilization complexity has — according to Gouda — remained "largely uncharted" [16].

5. Conclusions and Future Work

In this paper we have explored a method to prove the convergence of self-stabilizing algorithms which is different from the traditional self-stabilization methodologies. Our method borrows ideas from the area of control theory to prove that a given algorithm converges to a predefined state in finite time. We have shown that this method is feasible and have discussed its merits and shortcomings. By doing this, we have attempted to help bridging the gap between control theory and self-stabilization research. However, our paper is also an indication that this gap is still very wide; much work needs to be done to understand better how both research domains can benefit from each other.

Convergence properties are instances of general *liveness* properties of distributed systems [17]. Since *termination* is such a liveness property as well, one could hope that investigating the standard literature on (automated) termination proofs [18] could reveal other alternatives to the usual convergence function arguments. For example, the method of *induction* is frequently used to prove the termination of loops or entire algorithms. Applying this method to self-stabilization, one could try to prove a statement like "for any state s, the system will eventually reach a legal state again in finite time". Let us abbreviate this statement by $P(s)$. The base case of this proof would be to choose s as a *legal* state, which truthifies the statement trivially. The induction hypothesis would then assume that $P(s)$ holds and then try to prove $P(s')$, where s' is some state which is reachable from s via a single state transition. However, not all neighboring states of s may be considered as s'; the induction method requires that the set of states S must be a well-founded set. This means that $s \prec s'$ must hold for some well-founded ordering relation $\prec$ on S. Finding this ordering relation resembles finding "convergence paths" into the set of legal states and is thus as difficult as exhibiting a convergence function (see Fig. 8). It is interesting to note that the transfer function, although exploited for its termination-like properties, is not a termination function in the usual sense. Reviewing Figure 5, the non-monotonic behavior of the transfer function $G(t)$ is clearly visible. Future work will comprise to further examine the relationship between transfer functions and ordinary termination functions. Furthermore, we would like to compare our approach with techniques derived for dynamic discrete event systems [19, 20].

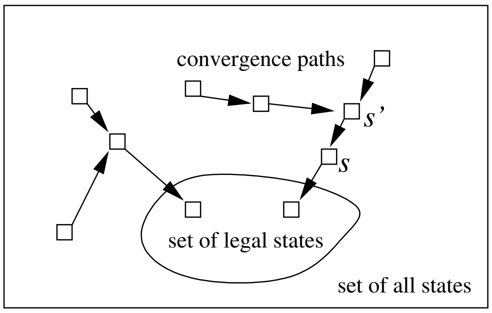

Figure 8. Convergence paths in stability proofs by induction

Additionally, we plan analyze the induction method as well as to investigate the essence of stability of *non-linear* control circuits. Based on these results, we plan to approach the solution of systems exhibiting distributed control, like distributed mutual exclusion and distributed election. We believe that the adoption of techniques from control systems will in particular lead to easy stability proofs for distributed, self-stabilizing algorithms.

Acknowledgments

We wish to thank Matthias Bormann for interesting discussions on termination proofs in sequential environments. We would also like to thank the anonymous referees for their valuable comments.

References

[1] Shlomi Dolev, Amos Israeli, and Shlomo Moran, "Self-stabilization of dynamic systems assuming only read/write atomicity", *Distributed Computing*, vol. 7, pp. 3–16, 1993.

[2] Ted Herman, *Adaptivity through distributed convergence*, PhD thesis, Department of Computer Science, University of Texas at Austin, 1991.

[3] Anish Arora and Mohamed G. Gouda, "Distributed reset", *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 1026–1038, Sept. 1994.

[4] Mohamed G. Gouda and Nicholas J. Multari, "Stabilizing communication protocols", *IEEE Transactions on Computers*, vol. 40, no. 4, pp. 448–458, Apr. 1991.

[5] Anish Arora, Paul Attie, Michael Evangelist, and Mohamed Gouda, "Convergence of iteration systems", *Distributed Computing*, vol. 7, no. 1, pp. 43–53, 1993.

[6] Oliver Theel and Felix C. Gärtner, "On proving the stability of distributed algorithms: self-stabilization vs. control theory", in *Proceedings of the International Systems, Signals, Control, Computers Conference (SSCC'98), Durban, South Africa*, Vladimir B. Bajic, Ed., Sept. 1998, vol. III, pp. 58–66.

[7] Anish Arora and Mohamed Gouda, "Closure and convergence: a foundation of fault-tolerant computing", *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

[8] R. Isermann, *Digitale Regelsysteme, Band I (in German)*, Springer-Verlag, New York, 1988, ISBN 0-387-16596-7.

[9] Anish Arora and Sandeep Kulkarni, "Detectors and correctors: A theory of fault-tolerance components", in *Proceedings of the 18th International Conference on Distributed Computing Systems*, 1998.

[10] Anish Arora, Mohamed G. Gouda, and George Varghese, "Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems", *Journal of High Speed Networks*, vol. 5, no. 3, pp. 293–306, 1996.

[11] Gerard Tel, *Introduction to Distributed Algorithms*, Cambridge University Press, 1994.

[12] George Varghese, *Self-stabilization by local checking and correction*, PhD thesis, MIT, 1993, Published as Technical Report MIT/LCS/TR-583.

[13] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev, "Self-stabilizing by local checking and global reset", in *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857*, 1994, pp. 326–339.

[14] Shmuel Katz and Kenneth J. Perry, "Self-stabilizing extensions for message-passing systems", *Distributed Computing*, vol. 7, pp. 17–26, 1993.

[15] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese, "Self-stabilization by local checking and correction", in *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, 1991, pp. 268–277.

[16] Mohamed G. Gouda, "The triumph and tribulation of system stabilization", in *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972*, 1995, pp. 1–18.

[17] Bowen Alpern and Fred B. Schneider, "Defining liveness", *Information Processing Letters*, vol. 21, pp. 181–185, 1985.

[18] Christoph Walther, "On Proving the Termination of Algorithms by Machine", *Artifical Intelligence*, vol. 7, pp. 101–157, 1994.

[19] Stanley D. Young and Vijay K. Garg, "On Self-Stabilizing Systems: An Approach to the Specification and Design of Fault Tolerant Systems", in *Proc. of the 32nd Conference on Decision and Control, San Antonio, TX, U.S.A*, 1993, pp. 1200–1205.

[20] Cüneyt M. Özveren and Alan S. Willsky, "Stability and Stabilizability of Discrete Event Systems", *Journal of the Association for Computer Machinery*, vol. 38, no. 3, pp. 730–752, July 1991.