Self-stabilizing Automata[*]

Torben Weis and Arno Wacker

Abstract Biological systems are known to be probabilistically self-stabilizing, i.e. with a high probability they can reach a stable state from any initial state. This property is very important to computer-based systems, too. However, building self-stabilizing systems is still very difficult. Proving that any given implementation is in fact self-stabilizing is even harder. Nature has a big advantage: Any living being must eventually die and limited energy limits the harm that an error can have on the system. This greatly simplifies the realization of self-stabilization. To transfer this concept to computer-based systems, we propose to modify the computational model on which software is currently being built. We introduce energy-awareness in Turing Machines (TMs). This will guarantee that any TM program that is correct in the absence of errors is at the same time self-stabilizing in the presence of errors.

1 Introduction

Today's software often assumes that errors do not occur. Better software designers define at least an error model. For example, they assume network errors but no memory errors. If errors occur which do not fit in the error model, the error is neither detected nor corrected. If the error fits in the error model, it can be detected and by default the application is stopped. In the best case, the error is detected and corrected.

Biological systems are different. They do not feature any error detection and they don't throw exceptions. In the physical world all possible states are allowed

Torben Weis
University of Duisburg-Essen, Duisburg, Germany e-mail: torben.weis@uni-duisburg-essen.de

Arno Wacker
University of Duisburg-Essen, Duisburg, Germany e-mail: arno.wacker@uni-duisburg-essen.de

[*] This work has been supported by the DFG SPP Organic Computing

and biological systems have the tendency of developing from most initial states to a set of preferred states. In these preferred states, living beings exist and reproduce themselves.

In computer science we divide all possible states in valid and invalid states. Furthermore, computer programs assume a precisely defined initial state. Any state that can be reached without errors from this initial state is then called a valid state. A system is called self-stabilizing if it transits from any invalid state to a valid state in a constant amount of time.

In the related work section we discuss algorithms which are known to be self-stabilizing. However, the development of such algorithms and the proof of their self-stabilization property require much time and expertise. In most commercial applications this is simply too expensive. Furthermore, real-life systems are very complex which renders theoretical proofs next to impossible.

Obviously, there seems to be a major difference between software development and biological evolution. In biology, evolution is constantly changing the genes and up to now every known genetic program either terminated (i.e. the species died) or it could reach a stable state where it continuously reproduces itself. Furthermore, no species has been accidentally created that happened to eat up the universe or bring it to a halt. Unfortunately, this is what we have to expect if we apply arbitrary mutations on software programs. We will most likely end up with a program that neither terminates nor reaches a valid state. Even worse, it will consume all CPU time, disable all interrupts and lock the computer.

We claim that the problem is the automaton which executes the programs. Biological systems are being executed on a physics engine which follows a set of fundamental laws. Computer systems are being executed on machines which are a Turing Machine or some equivalent. If we want to develop computer software inspired by biology, we must first fix the computational model, i.e. the machine on which the software is executed. In this paper we present a modified Turing Machine which takes fundamental laws of physics into consideration. The result is that applications running on this machine are automatically self-stabilizing.

2 Application Scenario

Biologically inspired software is not necessarily the best approach for all application scenarios. In some application domains errors are not part of the normal operation. Hence, if an error is detected, the system is stopped and the administrator must fix it. For example, office and enterprise applications belong to this category.

Applications dealing with sensors and actuators are different. Temporary sensing errors or temporarily broken actuators belong to the normal mode of operation and there is no system administrator available for fixing every possible problem. Thus, self-stabilizing systems are preferable, because they can autonomously recover from a wide range of errors without any intervention by a system administrator.

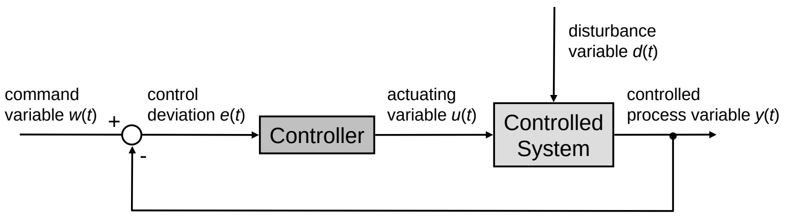

Figure 1 Control Loop

The application scenario for our research consists of a wide range of control applications. The structure of control applications is shown in Figure 1. The controlled system consists of actuators and sensors and interacts with the physical world. The controller is implemented in software and communicates with the controlled system by exchanging messages. Based on the sensor input and its internal state the controller sends commands to the actuators. The behaviour of the actuators in turn influences the sensors. Thus, the system is a software controlled feedback loop. Errors or disturbances can be introduced on the controlled system. If the controller is self-stabilizing, it can recover from any temporary sensing error and from any temporarily broken actuator in constant time. Thus, for every system exists a constant time t such that the entire system reaches a consistent state in no more than t seconds after all temporary errors are gone.

The aim of our research is to create a software development process and tools for building self-stabilizing controller software. In addition, we believe that the results of our work can be applied to other application scenarios, for example in the area of pervasive and ubiquitous computing.

3 Physics versus Turing Machines

The Turing-Church-Thesis claims that every effectively computable function can be regarded as computable under the definition of the Turing Machine. It does by no way claim the converse. Not every function that can be computed by a Turing Machine can be computed by a physical machine. The typical argument is that a TM has an infinite tape whereas all physical machines are limited. However, the difference between physics and Turing Machines is not only a matter of tape length.

Our argumentation is that Turing Machines do not obey the second law of thermodynamics, which states that ”the total entropy of any isolated thermodynamic system tends to increase over time, approaching a maximum value.” In contrast, a Turing Machine can work until eternity on a program that continuously increases the entropy of the tape. Even if we could build a physical computer with infinite amount of memory, it would still not be Turing equivalent because it must obey the second law of thermodynamics.

As a consequence of this observation, we modified the Turing Machine. The first law of thermodynamics states that ”in a closed system energy can neither be created nor can it disappear. It can only be transformed in other kinds of energy” (e.g. thermal energy or work force). Thus, we had to introduce a concept that is comparable to thermal energy and work force and a transformation between both of them. The second law of thermodynamics limits the transformation between thermal energy and work. It implies that it is impossible to construct a process that translates thermal energy lossless to work force. This is often expressed as: ”Perpetuum Mobili cannot exist”. Thus, our machine must have a way of transforming thermal energy to work force in a non-lossless way only.

We do not want to overstress the parallels to physics. However, the laws of thermodynamics have been the starting point of our approach and inspired our machine. Furthermore, these laws describe very well that there are some major differences between the computational model used by biology (the laws of physics) and the computation model of computer systems (Turing Machines).

4 Energy-aware Turing Machines

In our approach we assume that the read/write head of the TM has a certain thermal energy. The tape has the thermal energy 0 and no symbols exist on the tape initially, i.e. the head is hot and the tape is cold. The thermal energy of the read/write head can be transformed into work force and it can be transferred to the tape and its symbols. A read/write head can perform three kinds of work. It can move, read, or write. Performing any of these actions affects the thermal energies. We assume that the tape is huge, i.e. we can transfer much thermal energy to the tape without changing its temperature significantly. The symbols are in contrast tiny. Little energy transfer is required to heat them up. The head is supposed to be much larger than the symbols but small compared to the tape. Size matters because it determines how much energy is required to change the temperature of an entity.

In our machine the head is moving upwards and downwards. Moving the head upwards transforms thermal energy of the head into potential energy. Moving the head downwards transforms potential energy back into thermal energy. Because of the second law of thermodynamics this transformation process must not be lossless. Therefore, during each movement the read/write head heats up the tape, i.e. transfers the thermal energy $\Delta E > 0$ from the head to the tape (see Figure 2). As a result no energy is ever lost or created and a perpetuum mobile (i.e. a head that is moving forever) is impossible because more and more energy is transferred to the tape.

Over time the symbols exchange thermal energy with the tape. Thus, the symbols cool down. Eventually the tape and all symbols will have the same temperature and can no longer exchange thermal energy. A symbol which has the same temperature as the tape is no longer readable and disappears. Thus, the head must always write symbols which are at least as hot as the tape. If this is no longer possible, no symbols can be written any more.

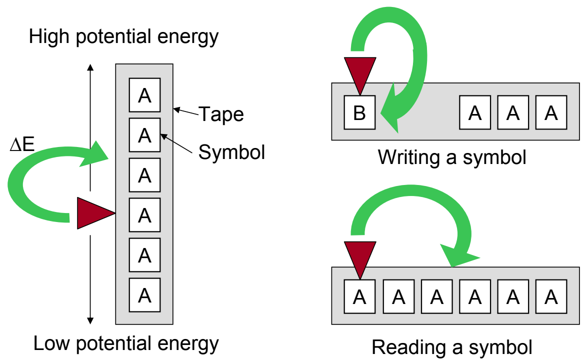

Figure 2 Operations of an energy-aware Turing Machine

Writing a symbol to the tape transfers thermal energy to the symbol until head and symbol have the same temperature. This implies that hot heads write hot symbols and cooler heads write cooler symbols. This is in line with the second law of thermodynamics because the thermal energy flows from a warm entity (the head) to a cooler entity (the symbol) until both having the same temperature.

If the head is reading a symbol, it cools down until the head and symbol have the same temperature. The thermal energy lost by the head in this process is transferred to the tape. Thus, the cooler the symbol that is being read, the cooler the head becomes while reading it. This implies that the head cannot read a cold symbol and write a hot symbol afterwards. The energy of a freshly written symbol is always higher than the energy of any symbol written afterwards.

The head cannot move until eternity because it constantly looses thermal energy to the tape. Eventually the only possible movement is downwards because this is the position with the lowest potential energy and the head cannot spend more thermal energy on moving upwards. Eventually, the machine will fall back to its initial state. The head falls down to the lowest position and all symbols disappear once they reach the same temperature as the tape.

The amount of energy that is lost (i.e. transfered between head or symbols and tape) determines the stabilization time. This energy loss must always be higher than 0 to avoid a perpetuum mobile. The higher the loss, the faster will the head return to its initial position and the faster will erroneous symbols disappear. On the other hand, high energy losses mean that the system will forget fast, i.e. its view on past sensor values is very narrow, because old values disappear very quickly.

The machine presented so far is in line with the rules of thermodynamics. However, it is not very useful yet. The machine has simply a limited time for execution. After this time all parts of the machine have the same temperature and the machine resets to its initial state. In the next chapter we will therefore extend our machine to read sensor values and control actuators.

5 Sensors and Actuators

The rules of thermodynamics cited so far apply to closed systems. However, if we allow our machine to receive sensor data from outside the machine and to control actuators outside the machine, the system is no longer closed. We assume that the machine increases its thermal energy when it receives data (from a sensor) and emits thermal energy when it sends data (to actuators). As long as sensors continuously send more data, the machine does not necessarily cool down to the point where it falls back to its initial state (head at the lowest position and no symbols on the tape).

Our machine is in fact a three tape Turing Machine as shown in Figure 3. The middle tape is the working tape. The first one is the input tape and the last one the output tape. A sensor is sending input data as a finite sequence of symbols. If the machine is in a receiving state (which is the case when the head is at the initial position) these input symbols are copied on the input tape. The head moves along all received symbols and they transfer thermal energy to the head. Thus, the energy level of the machine is increased and it can read and process the data.

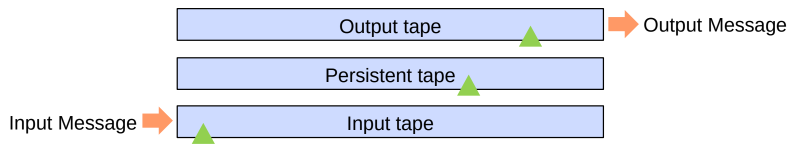

Figure 3 3-Tape Turing Machine

The machine can write symbols on the output tape. When the machine falls back to the initial position, it can receive new input and it sends its output. The symbols on the output tape are sent to actuators. By doing so these symbols disappear from the output tape because their thermal energy is sent to the actuators. The machine looses energy by sending.

The energy limits the number of symbols the machine can send. This shows already one great advantage of this machine. The damage its output can cause is limited by the input it receives. As long as sensors are sending only at a limited frequency and with limited thermal energy, the machine cannot run berserk by flooding the system with bogus symbols. For example, accidental distributed denial of service attacks are impossible because of energy restrictions. This argument holds for all programs executed by this machine.

6 Self-stabilization

The inability of our machine to store data forever (symbols loose energy to the tape) and its inability to amplify the energy of stored data (cannot read a cold symbol and write a hot one) is the key to its self-stabilization property. We assume that the machine is executing a program that is correct if no errors occur. Possible errors are:

- Tape symbols are accidentally added, removed, or modified
- The head position is accidentally changed
- Sensor data is lost, duplicated, modified or its ordering is changed
- Too much energy is sent to the machine
- The machine has been loosing to much energy

If anything is wrong with the current state of the machine, then a certain thermal energy is attached to this wrong information. It could be a wrong symbol or a wrong head position. After a constant time the head must move back to its initial position and after a constant time all symbols have cooled down and disappear. All information disappears after a constant time and it cannot be refreshed, i.e. its energy cannot be amplified. Furthermore, the energy of derived information cannot be higher than the energy of the source information, because the head cannot write a symbol that is hotter than any other symbol read or written before. After a constant time it does not matter whether some information was wrong or not because the information itself and all information derived from it disappears. What remains has necessarily a high energy level and is therefore fresh information that is in no way dependent on the wrong information and therefore correct. The only source for fresh information is new sensor readings. Thus, after a constant time all erroneous information is gone and only fresh and correct information remains in the machine.

Our machine greatly simplifies the development of self-stabilizing algorithms. The developer does not have to prove that his algorithm can recover from every possible error. He must prove that the algorithm executes correctly on our machine in the absence of errors. If this is the case, the self-stabilization property is guaranteed. This is a great advantage over current coding techniques where the self-stabilization property requires a manual proof.

However, the proof of correctness is now a bit more complex than with normal Turing Machines. The proof of correctness must take the energy transfer into account. In addition, no sane programmer will develop an algorithm for execution on a Turing Machine. Therefore, we are working on a model-driven development approach [1]. The developer describes his program on a very high-level programming language [2]. This program is then automatically translated into a program for our machine. This program can now be tested on a simulated machine. Testing is of course no proof of correctness, but in practice testing is easier to do than correctness proofs.

A disadvantage of our machine is that it is even less efficient than normal Turing Machines. The CPU must calculate the energies whenever the head moves, reads, or writes. Thus, it is no platform on which you would execute a word processor or enterprise applications. However, a machine that is inherently forgetful is not useable for this kind of applications anyway.

For control applications, however, our machine is well suited. A constant performance factor is often tolerable. Furthermore, the forgetfulness of our machine is no problem here. When a control application receives input, it calculates its output

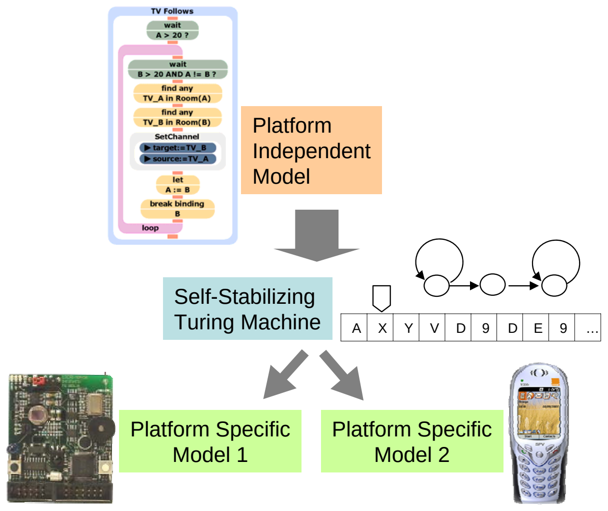

Figure 4 Development process

based on the new input and a fixed amount of previous inputs. Very old inputs are not required, which is good, because our machine has already forgotten these old inputs and every data derived from them.

7 Implementation and Simplification

So far the energy-aware Turing Machine is a theoretical concept. To turn it into something useful, we must execute the energy-aware Turing machine programs on a real machine. Although it would be an interesting challenge to build a thermodynamic machine which adheres to our formal specification, this is of course not a practical thing to do. Instead, we see two possible options: A software solution and a silicon hardware solution.

In the first case, the programs are executed in a simulated energy-aware Turing Machine. This is the easiest thing to do and ensures that the program is self-stabilizing as long as the underlying software is stable. However, if the simulator is not working correctly or if the operating system crashes then the entire system will not be self-stabilizing. The more radical approach is to build hardware in silicon which behaves like an energy-aware Turing Machine. In this case there is no software layer (simulator or operating system) that could fail. Such a system would really be self-stabilizing. Since our expertise is not in chip design, we are working with the simulator approach currently.

In both cases, the energy-aware Turing Machine is hard to implement because it needs much computation to calculate the energies as floating point numbers. Therefore, we simplified the energy-aware Turing Machine without sacrificing its self-stabilization properties. First of all, energy is quantized to represent energy levels as integers. Furthermore, the potential energy of the read/write head can be ignored. It has only been introduced to make sure that the head falls back to the initial position once it lost too much thermal energy. In an implementation, we simply check after each step whether the head energy level is too low and move it back to the initial position.

The symbols are constantly transferring energy to the cooler tape until they disappear. We implement this by storing for each symbol the step in which it has been written and its energy level at this time. If the head later on reads the symbol, we subtract one from the energy level for every step that happened in between. If the resulting energy level is 0 or below, the symbol is erased.

It could happen that a memory error changes the step or energy level stored for some symbol. This does not harm the self-stabilization property as long as all of these values are expressed as numbers with a fixed amount of bits. Thus, if a memory error accidentally increases the energy level of a symbol, the additional energy is limited by the amount of bits. After a constant time this energy has been transferred to the tape, the symbol disappears, and the system can stabilize again. The fewer bits we use, the shorter is the self-stabilization time. However, the forgetfulness of the machine increases when the number of bits decreases. The best number of bits to use is therefore a trade-off between stabilization time and forgetfulness.

8 Related Work

In our approach we are extending a three-tape Turing Machine which is known as Persistent Turing Machine (PTM). PTMs [3, 4] are a minimal extension of Turing Machines (TMs) [5] that express interactive behaviour providing a natural model for sequential interactive computing. A PTM has three tapes: a read-only input tape, a write-only output tape, and a persistent working tape which is preserved among interactions, i.e., among successive computations of the PTM.

Self-stabilizing algorithms have been introduced by Dijkstra in 1974 in his seminal paper on a self-stabilizing token passing algorithm [6]. A self-stabilizing system recovers from any transient fault within a bounded number of steps [7] provided that no further fault occurs until the system is stable again. The maximum number of steps required to bring the system back into a legitimate state is called stabilization time. Self-stabilization is usually proven by showing that the system satisfies convergence (started from an arbitrary state it reaches a legitimate state within a bounded number of steps) and closure (once the system has reached a legitimate state, it stays in the set of legitimate states) if no faults occur.

Many self-stabilizing algorithms utilize soft state, a design pattern which is known from many network protocols [8]. One possible way to implement soft state is leasing. In this case, the state of the system is only leased and has to be periodically refreshed to remain valid. If it is not refreshed in time (i.e., if it expires), it is invalidated and usually deleted. For example, we used subscription leasing to realize self-stabilizing publish/subscribe [9]. A generic implementation of self-stabilization is possible with a precautionary periodic reset [10]. In this case, all state is regularly deleted and rebuilt from an initial configuration. This ensures that corrupted state is eliminated while the correct state is established.

9 Outlook and Conclusions

We presented a modified Turing Machine which features an energy concept which is based on the laws of thermodynamics. Every program that executes correctly on this machine in the absence of errors is guaranteed to be self-stabilizing in the presence of errors. This greatly simplifies the development process since no manual proofs of the self-stabilization property is required.

They key to the self-stabilization property is that derived information has always a lower energy than the information it has been derived from. Together with the inability to amplify the energy of information we get the desired self-stabilization property. However, here we are more restrictive than the laws of thermodynamics would have required. Perhaps other machines exist which have the same self-stabilization property, but are less restrictive in some ways.

It is an open question whether the machine presented in this paper is powerful enough to execute all possible self-stabilizing algorithms. It may be the case that self-stabilizing algorithms exist which cannot execute correctly on our machine. So far we can only state the converse: if it executes correctly, then it is self-stabilizing.

Other open problems are related to the software development process. How can one easily develop applications for this kind of machine? To obtain the self-stabilization property the programs must always be executed under the control of this machine. Today's CPUs would waste much time on this. Perhaps specialized hardware could significantly improve the speed of execution.

In the future we will work on the software development process to ease the development of self-stabilizing control applications. Furthermore, a more formal description of the machine will be subject to our next publication.

References

[1] T. Weis, H. Parzyjegla, M. A. Jaeger, and G. Mühl. Self-organizing and self-stabilizing role assignment in sensor/actuator networks. In *Proceedings of DOA 2006*, volume 4276 of *Lecture Notes in Computer Science*, pages 1807–1824. Springer, 2006.

[2] M. Knoll, T. Weis, A. Ulbrich, and A. Brändle. Scripting your home. In *Proceedings of the 2nd International Workshop on Location- and Context-Awareness (LoCA 2006)*, pages 274–288, Dublin, Ireland, May 2006.

[3] Dina Q. Goldin. Persistent turing machines as a model of interactive computation. In *FoIKS '00: Proceedings of the First International Symposium on Foundations of Information and Knowledge Systems*, volume 1762 of *Lecture Notes In Computer Science (LNCS)*, pages 116–135, London, UK, 2000. Springer-Verlag.

[4] Dina Q. Goldin, Scott A. Smolka, Paul C. Attie, and Elaine L. Sonderegger. Turing machines, transition systems, and interaction. *Inf. Comput.*, 194(2):101–128, 2004.

[5] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

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

[7] Shlomi Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000.

[8] D. Clark. The design philosophy of the darpa internet protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114. ACM, 1988.

[9] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, and A. Ulbrich. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. In *Proceedings of Euro-Par 2005*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 664–674, Lisbon, Portugal, August 2005. Springer.

[10] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Transaction on Computers*, 43(9):1026–1038, September 1994.