Toward Feedback Stabilization of Faulty Software Systems: A Case Study

Stephen Waydo William B. Dunbar
Control and Dynamical Systems
California Institute of Technology
Pasadena, CA 91125

Eric Klavins
Electrical Engineering Department
University of Washington
Seattle, WA 98115

***Abstract*— Software systems generally suffer from a certain fragility in the face of "disturbances" such as bugs, unforeseen user input, unmodeled interactions with other software components, and so on. A single such disturbance can make an entire system hang or crash. We postulate that what is required to address this fragility is a general means of using feedback to robustly stabilize these systems. In this paper we develop a model of an iterative software process, specifically a nondeterministic, faulty list sorter. Feedback is introduced into the process to achieve robust stability with respect to incorrect sorting operations. To keep the computational requirements of the controllers low, randomization and approximation are used. Methods by which software robustness can be enhanced by distributing a task between nodes, each of which are capable of selecting the "best" input to process, are also explored. The particular case of a sorting system consisting of a network of partial sorters, some of which may be buggy or even malicious, is examined.**

I. INTRODUCTION

Software systems are very often not robust to disturbances, which may come in the form of bugs, unforeseen input, unexpected interactions with other system components, and so on. The language of dynamical systems and control theory is a natural one in which to express the idea of stability, and in this paper we explore how an iterative software process could be modeled within this framework. In [6] we explore a preliminary model of the time/space evolution of a generic iterative software system and suggest analogs to the traditional control-theoretical notions of estimators and controllers that may be used to feedback-stabilize, and thereby improve, the performance of the system. In the work here, we explore the example of a faulty list sorter, where sort operations may or may not (nondeterministically) be correct. The model is numerically validated and feedback is shown to improve the faulty sorter's performance. We then examine a system wherein multiple sorters that use feedback to monitor their progress are networked together. Overlap of partial sort operations, although minimal in some cases, and the use of feedback together result in the convergence of fault-free sorters in the presence of a faulty sorter.

The research we report on in this paper is related to the theory of self-checking programs as described in, for example, [11], [14], [3] except that we are concerned with stability and disturbance rejection rather than error correcting *per se*. The networked sorters example in Section IV resembles in some respects N-Version Programming [2]. Also similar is the idea of self-stabilizing protocols [5], [12] wherein a network of processors executing a self-stabilizing protocol can be shown to recover from disturbances and arbitrary initial conditions into a set of *legal* states. In fact, the same analogy that we employ here of a *pseudo-energy* function, or Lyapunov function, can be used to show the protocols are robust and stable [9], [13]. We believe the idea of self-stabilization exactly corresponds to robust control design and hope to make this and related notions formal in future work. A more complete list of related references is given in [6]. What has not been investigated, to the best of our knowledge, is the use of feedback to control, rather than merely terminate, a faulty process. The addition of a control input to a program may in fact require fundamental reworking of the program/observer paradigm. We hope that the present paper suggests a possible avenue for such an effort.

Relevant definitions and discussion on the need for approximation are first given in Section II. In Section III we investigate a modeling and control approach for stabilization of nondeterministic sorting algorithms. In Section III-A we define an appropriate metric on the group of permutations of n elements and a *pseudo-energy function* measuring the sortedness of a list for the purposes of control. Section III-B provides an analysis of the open-loop dynamics of a faulty list sorter using a Markov chain model. In Section III-C we use a simple feedback controller to stabilize the sorting system in Section III-B. In Section IV we describe a distributed array of nodes in which each node consists of a partial sorter and a controller. We investigate the behavior of the system through simulation. Conclusions and future directions are given in Section V.

II. PROBLEM DESCRIPTION

A particularly interesting class of programs we investigate are iterative processes that do not run to completion but instead provide output after some number of steps and then use this output as their input for the next set of iterations. As such programs already incorporate feedback in this sense, applying control via manipulation of the iterated information may be a useful step toward correcting aberrant behaviors.

To develop a systems theory perspective of software systems, we consider defining a state $x \in X$ of the system and a metric d on X, which quantifies the "closeness" of two system states. For a software system, x may simply be a snapshot of the memory used by the software. A metric d, describing the "distance" between two states, is a means by

which we may determine how well the software system is performing its assigned task. In software systems X is rarely a metric space (or even a reasonable topological space), and thus some we may need to resort to a surrogate for d. In the context of sorting, however, we explore the symmetric group $X = S_n$, the group of all permutations of $\{1, ..., n\}$, for which we can a define a metric (Section III-A).

For analysis and feedback controller design, it is useful to define a *Pseudo-Energy Function* $V : X \to \mathbb{N}$ of the system state, with the set $\{x \mid V(x) = 0\}$ defining the goal states. In this paper, V is an increasing function of the distance of the current state to a unique goal state (the sorted list) defined by $V(x) = 0$. The pseudo-energy function can be thought of, roughly, as a *Lyapunov Function* [8] for the system. Several pseudo-energy functions are defined in Section III-A.

The state x of a software process may be enormously complicated. In fact, the computational cost of determining $V(x)$ may be equivalent to the cost of executing the software process to completion. For this reason, approximations of the pseudo-energy functions are required.

III. Sorting Lists

In this section we describe the set of lists and the metrics and pseudo-energy functions that can be defined on them. A model of the evolution of a pseudo-energy function is given and numerically validated. To monitor a piece of software on-line, it is critical that the control tools require minimal (relative to the software) computational complexity. As such, we comment on the complexity of the functions defined and use approximations in computing the pseudo-energy function used for stabilization of the model. In our simulations, we observe an improvement in closed-loop behavior as more computations are allotted to the controller.

A. Metrics and pseudo-energy functions for List Sorting

We make the simplifying assumption that all lists generated by partial sorting are equal when viewed as sets. A faulty sorter or disturbance may unsort the list, but the assumption requires that the list may not change as a set. A list $L = [L[1], ..., L[n]]$ drawn from the set $\{1, 2, ..., N\}$ is a sequence of n ordered and distinct elements. We further assume that $M = n$: the set of all lists of length n is then the symmetric group S_n of all permutations of $\{1, ..., n\}$. A list L is sorted if $L[i] < L[j]$, for all $i < j$, and we denote the sorted list as $L^* = [1, 2, ..., n]$. A metric for sortedness quantifies the distance between any two lists in a given group. Pseudo-energy functions, in the context of sorting, refer to functions from $S_n \to \mathbb{N}$ that rank lists by sortedness. For example, a (trivial) pseudo-energy function might output 0 if the list is sorted and 1 otherwise. Pseudo-energy functions can be used to prove the correctness of a particular sorting algorithm, e.g. Bubblesort [10]. From the control analysis perspective, a metric is likely to prove useful in verifying properties of the closed-loop behavior of a sorter/controller agent. The function of the controller as described above requires the pseudo-energy for any given list. We now give an example pseudo-energy function and metric for list sortedness.

Definition 3.1 (Total Inversion Function): The *total inversion function* V_{TI} of a list L is

$$V_{TI}(L) \triangleq \sum_{i=1}^{n} \sum_{j=i}^{n} \langle L[i] - L[j] \rangle$$

where

$$\langle x \rangle \triangleq \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise.} \end{cases}$$

In words, V_{TI} gives the total number of pairs that are out of order, counting 1 for each pair out of order, with a maximum value of $\binom{n}{2}$. Determining V_{TI} is $O(n^2)$ and $V_{TI}(L) = 0$ iff $L = L^*$.

We now define two vectors that will be used in the metric we give below.

Definition 3.2 (Total Inversion Vectors): The *total inversion vector* $\boldsymbol{q} : S_n \to \{0, ..., n-1\}^n$ has n components $[q_1(L), ..., q_n(L)]^T$, where the kth component is defined by

$$q_k(L) = \sum_{j=k}^{n} \langle L[k] - L[j] \rangle.$$

The *ordered total inversion vector* $\boldsymbol{q}^o : S_n \to \{0, ..., n-1\}^n$ has n components $[q_1^o(L), ..., q_n^o(L)]^T$, where the $L[k]^{th}$ component is defined by

$$q_{L[k]}^o(L) = \sum_{j=k}^{n} \langle k - L[j] \rangle.$$

In words, q_i^o is the number of elements less than i, located in $\{L[1], ..., L[n]\}$, to the right of i. The definition of $\boldsymbol{q}^o$ is based on [4] and references therein, which discuss the construction of a (total) inversion list from a given permutation. The reason for the use of the word "ordered" in defining $\boldsymbol{q}^o$ is that its construction depends upon the location of each $L[k]$ relative to its value; consequently, the definition does not generalize to operating on lists. On the other hand, component k in the $\boldsymbol{q}$ vector corresponds to the $i = k$ summation term in the expression for V_{TI}. As such, $\boldsymbol{q}$ is already well-defined for operating on lists, rather than being restricted to permutations. Note that the nth component in $\boldsymbol{q}$ and the 1st component in $\boldsymbol{q}^o$ are always zero. We now define a metric based on the ordered total inversion vector.

Lemma 3.1: Given the function $d : S_n \times S_n \to \mathbb{R}$ defined by

$$d(L_1, L_2) \triangleq ||\boldsymbol{q}^o(L_1) - \boldsymbol{q}^o(L_2)||,$$

where $|| \cdot ||$ is any norm on $\mathbb{R}^n$, (S_n, d) is a metric space.

A proof is given in [6]. In this case, it follows from the definitions that $V_{TI}(L) = d(L, L^*)$. Other metrics are possible, such as the Kendall distance K and Spearman's footrule distance F, defined in [1] for $N = n$. Comparisons to d are made in [6].

B. Open-loop Behavior

To explore the issues involved in stabilizing and improving the performance of a sorting system, we consider a model of the simplest imaginable (buggy) sorting system. The sorter is a dynamic system whose state at step k is the list $L(k)$. The pseudo-energy at time k is taken to be the value of the total inversion function of the list

$$V(k) \triangleq V_{TI}(L(k)),$$

which for a list of length n can vary from 0 (no pairs are out of order) to $V_{max} = \binom{n}{2}$ (all pairs are out of order). At each time step, the sorter picks an adjacent pair of list entries. We suppose that this is a "correct" operation (i.e. the chosen pair is out of order) with probability p. The sorter then swaps the pair with probability w. If the list is already completely sorted or unsorted ($V = 0$ or V_{max}), the sorter simply swaps some adjacent pair with probability d. $L(k)$ is thus a random variable, and $V(k)$ is a random variable that is a function of $L(k)$. The probability distribution of $V(k+1)$ is dependent only on the distribution of $V(k)$, and so it can be modeled using a Markov chain. Define the state transition matrix T with its $(i,j)^{th}$ element given by

$$T_{i,j} \triangleq P[\, V(k+1) = j \mid V(k) = i \,].$$

Denoting a possible value for $V(k)$ by q, a state transition matrix of dimension $(m+1)\times(m+1)$, where $m = V_{max}$, is obtained. Note that swapping an adjacent pair (with distinct values) will always increment or decrement V_{TI} by 1. The state transition probabilities are then

$$\begin{aligned} T_{q,q} &= (1-w) \\ T_{q,q-1} &= pw, \;\; 1 \le q < m \\ T_{q,q+1} &= (1-p)w, \;\; 1 \le q < m \\ T_{0,1} &= w \\ T_{m,m-1} &= w \\ T_{q,q\pm\delta} &= 0, \;\; \forall\, \delta > 1. \end{aligned}$$

The left eigenvector v of this matrix corresponding to eigenvalue 1 ($vT = v$) is called the *(neutrally) stable left eigenvector*. It describes the long-term distribution of the pseudo-energy value $V(k)$. Following the method of [7], we have the following proposition.

Proposition 3.1: The neutrally stable left eigenvector v of the state transition matrix T is given by $v = [v_0 \dots v_m]^T$, where $v_0 = 1$,

$$v_i = \frac{(1-p)^{i-1}}{p^i}, \; 1 \le i < m, \quad v_m = \frac{(1-p)^{m-1}}{p^{m-1}}.$$

The proof is straightforward and is given in [6]. Normalizing v we obtain the long-term probability distribution v' of V as

$$v' = \frac{v}{\eta}, \quad \eta \triangleq \sum_{i=0}^{m} v_i.$$

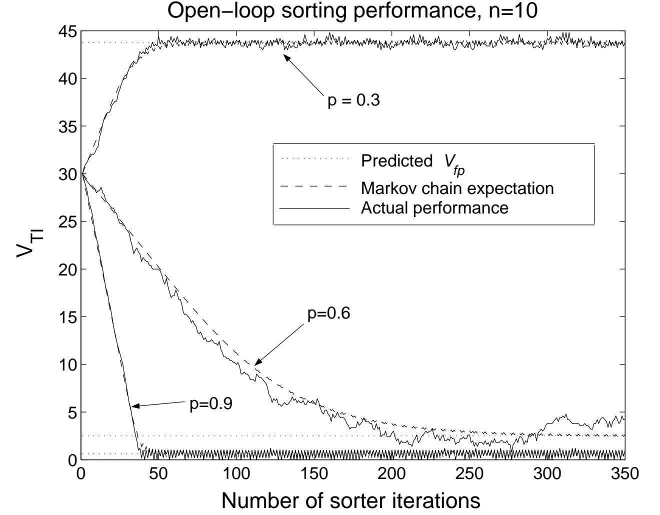

Fig. 1. Comparison of actual sorter performance to model.

The weighted sum of the entries of v' is the asymptotic expected value of V, which we call the *fixed point pseudo-energy* $V_{fp} \triangleq \lim_{k\to\infty} E[V(k)]$. We have

$$V_{fp} = \sum_{i=0}^{m}(i)v'(i) = \frac{p^m - (1-p)^m[1 + 2m(2p-1)]}{2(2p-1)[p^m - (1-p)^m]}.$$

Naturally, with higher probability of correct sort operations, the more likely the list will be sorted (V approaches 0). Figure 1 is a plot of the Markov chain-predicted time history, the predicted V_{fp}, and a time average of 10 actual sorting runs for a list of length 10. The actual sorter performance closely matches that predicted by the Markov chain analysis.

C. Closing the Loop

Underlying the following approach is the assumption that we are allowed to control the number of iterations the software performs and when the iterations start. The above Markov chain model can be extended to show the benefit of including a simple controller. We now model the same sorter along with an approximate checker that computes an approximation $\hat{V}$ of V. After each iteration k, the checker picks l random pairs and calculates $\hat{V}(k)$, the number of subsampled pairs that are out of order. The checker then rejects the sorting step if $\hat{V}(k) \ge \hat{V}(k-1)$. Although checking is here more expensive than a single sort iteration (O(l^2) versus O(1)), we investigate cases for small and large l to observe the tradeoff. Also, the restriction to checks between single sort iterations permits tractable analysis, as shown. In the network sorting example in the next section, we also explore cases where checking operations are cheaper than the sort operations permitted between checks.

The accuracy of the checker, or its ability to predict the current pseudo-energy, is here derived. The sample space of the checker consists of $\binom{n}{2}$ pairs. Again, we use $V(k) \triangleq$

$V_{TI}(L(k))$. If $V(k) = b$, the sample space of the checker consists of b out of order pairs and $\binom{n}{2} - b$ in order pairs. For $\hat{V}$ to be equal to some value c, the checker must pick c out of order pairs and $l - c$ in order pairs. The probability that the checker does so is

$$P[\hat{V} = c \mid V = b] = \left[\binom{b}{c}\binom{\binom{n}{2} - b}{l - c}\right] \Big/ \left[\binom{\binom{n}{2}}{l}\right].$$

In the following, we denote $V(k)$ (or $\hat{V}(k)$) as V_k ($\hat{V}_k$). Two probabilities are used to characterize the checker – the probability r_1 that $\hat{V}$ decreases when V does and the probability r_2 that $\hat{V}$ does not decreases when V increases. The value of r_1 is a function of b and l as

$$r_1(b, l) = P\left[\hat{V}_k < \hat{V}_{k-1} \,\middle|\, V_{k-1} = b, V_k = b - 1\right].$$

Because $\hat{V}_k$ and $\hat{V}_{k-1}$ are separate computations, they are independent random variables, and

$$\begin{aligned} &P[\hat{V}_{k-1} = c_1, \hat{V}_k = c_2 \mid V_{k-1} = b,\ V_k = b - 1] \\ &= P[\hat{V}_{k-1} = c_1 \mid V_{k-1} = b] \cdot P[\hat{V}_k = c_2 \mid V_k = b - 1] \\ &= \left[\binom{b}{c_1}\binom{m-b}{l-c_1}\binom{b-1}{c_2}\binom{m-b+1}{l-c_2}\right] \Big/ \left[\binom{m}{l}^2\right], \end{aligned}$$

where $m = \binom{n}{2}$, $c_2 < c_1$. Summing the above expression over all c_1, c_2 gives the needed probability

$$\begin{aligned} r_1 = \binom{m}{l}^{-2} \sum_{c_1=1}^{m} \binom{b}{c_1}\binom{m-b}{l-c_1} * \\ \sum_{c_2=0}^{c_1-1} \binom{b-1}{c_2}\binom{m-b+1}{l-c_2}. \end{aligned}$$

Figure 2 is a plot of r_1 as l ranges from 1 to $\binom{n}{2}$ for a list of length 10 with $V = 23 \approx \binom{n}{2}/2$. Similar reasoning leads to the probability r_2 as

$$\begin{aligned} r_2(b, l) &= P[\hat{V}_k \geq \hat{V}_{k-1} \mid V_{k-1} = b, V_k = b + 1] \\ &= \binom{m}{l}^{-2} \sum_{c_2=0}^{m} \binom{b+1}{c_2}\binom{m-(b+1)}{l-c_2} * \\ &\quad \sum_{c_1=0}^{c_2} \binom{b}{c_1}\binom{m-b}{l-c_1}. \end{aligned}$$

The state transition probabilities are very similar to the open-loop case, with the addition that the system may now reject sorter steps (correctly or incorrectly) according to the above probabilities. Using the same definition of the state transition matrix T as before we have

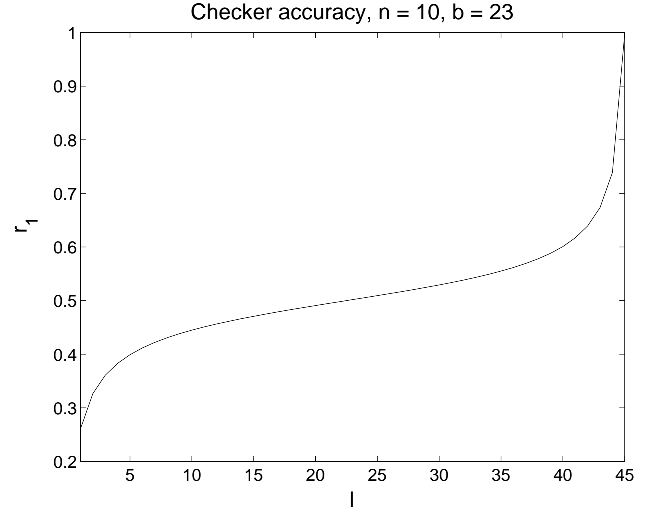

Fig. 2. Theoretical checker accuracy

$$\begin{aligned} T_{q,q} &= (1 - w) + pw(1 - r_1) + (1 - p)wr_2, \ \ 1 \leq q < m \\ T_{q,q-1} &= pdr_1, \ \ 1 \leq q < m \\ T_{q,q+1} &= (1 - p)w(1 - r_2), \ \ 1 \leq q < m \\ T_{0,0} &= (1 - w) + wr_2 \\ T_{0,1} &= w(1 - r_2) \\ T_{m,m} &= (1 - w) + w(1 - r_1) \\ T_{m,m-1} &= wr_1 \\ T_{q,q\pm\delta} &= 0, \ \ \forall\, \delta > 1. \end{aligned}$$

Work is currently in progress to develop a closed-form solution for or approximation to the stable eigenvector of the closed-loop transition matrix following the methods used above. Until such a solution is found, numerical methods can be used to predict v' and V_{fp}.

Figure 3 is a plot of the Markov chain-predicted time history, the predicted V_{fp}, and a time average of 10 actual sorting runs for a list of length 10 and a sorter with $p = 0.4$. The open-loop performance is shown along with that of checkers with l equal to 20, 30, and 40. Note that V_{fp} drops quickly once l becomes larger than $\binom{n}{2}/2 \approx 23$; further increasing l increases the rate at which q approaches V_{fp}. The actual sorter performance again closely matches that predicted by the Markov chain analysis. Note that in all closed-loop cases $V_{fp} = 0$, but lower values for l resulted in much slower convergence rates.

In the above discussion, only *sorter* iterations were taken into account when judging convergence rates, in which case larger values of l will clearly always improve convergence time. If one plots the closed-loop performance in Figure 3 as a function of the total number of iterations, convergence rates of $\hat{V}$ still improve with increasing l. The reason appears

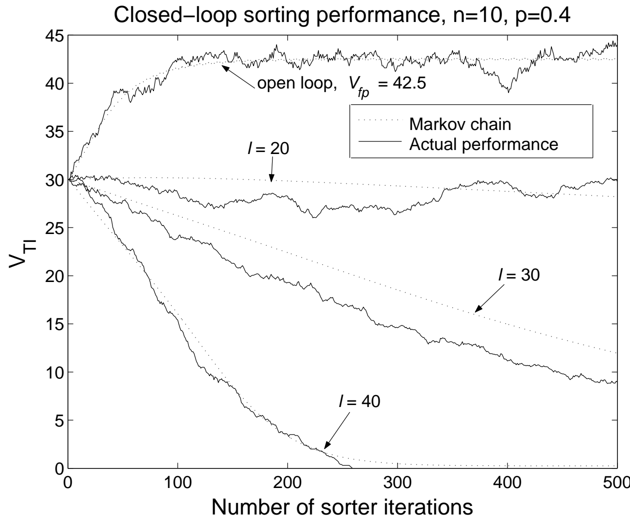

Fig. 3. Comparison of actual sorter performance to model.

to be that this sorter makes more bad decisions than good ones, on average. The Markov chain-predicted closed-loop time history in terms of sort and total iterations when good decisions are more frequent ($p = 0.6$) is explored numerically in [6]. In this case, $l = 10$ resulted in fewer total steps than $l = 30$ or 40 to converge. It may be that there is some optimal l (as a function of p) that will result in the fastest total convergence time. This is an issue we would like to explore in considerably more detail, especially in the case where the sorting accuracy is not know a priori and l must be adaptively tuned in some way.

IV. A Network of Sorters

In this section we consider a fully connected, four node network of sorters, each of which is equipped with a pseudo random number generator thereby making random choices possible. The objective is that the network converge (defined below) on L^*, which is defined to be in increasing order from left to right. The operation of the network is as follows:

- Each node (sorter) is initially given the same list of length n, with elements from the set $\{1, ..., N\}$, $N > n$.
- The *partial sort* operation on each node randomly picks j ordered pairs, with the first element of each pair to the left of the second element, and swaps them if they are out of order with some probability. After all partial sorts, each node transmits its current list to all other sorters it is connected to in the network (all other sorters in this case).
- Once the initial partial sort and transmission has occurred, each node repeatedly performs the *pick* and partial sort operations subsequently. The pick operation computes the approximate pseudo-energy $\hat{V}$ for each incoming list and selects the list with the smallest $\hat{V}$ value to be partially sorted. The approximation of $\hat{V}$ is quantified by the constant $\kappa \in \{1, ..., n\}$, which is the length of an array window randomly extracted from each input list. Note that randomness in the array window extractions means a node that makes the correct sort decisions all of the time could eventually sort the list by using its own list for subsequent operations.
- Define the output of the network at iteration k as the list (or state) of the node with the lowest pseudo-energy at iteration k. A network is said to *converge* if the pseudo-energy of its output converges to a value L_c.

The stability and expected value of L_c, desired to be the sorted version of the list, are the main indicators of the performance of the network. We are particularly interested in the network performance in situations where one or more of the sorters is imperfect and κ is fairly small for computational reasons. Will the output of bad sorters propagate through the network or will the pickers be good enough to weed out bad lists? We investigate the performance here by example.

In the following, $(N, n, j) = (100, 30, 10)$ and the sorter iteration histories shown are actually an average of 50 separate runs, all with a different randomly chosen initial list (picker iterations are not shown). When all sorters make correct decisions all of the time, all four sorters (and thus the network) converge to the sorted list for $\kappa = 5$ and $\kappa = 24$, with faster convergence for larger κ (see figures in [6]). To explore the effect of a faulty sorter, node 4 is given a bad sorter that flips a coin and if it is heads, swaps the randomly chosen ordered pair; otherwise, the pair is kept in the original order. The corresponding pseudo-energy histories are shown in Figure 4 for $\kappa = 5$ and $\kappa = 24$, where again we average over 50 runs, all with different initial lists. The figure shows that for a better approximation of the true pseudo-energy, i.e. for larger κ, the fault-free sorters converge *on average* independently of the faulty sorter. For $\kappa = 5$, convergence suffers as the faulty sorter's list is more likely to be chosen by the good sorters more frequently. We suspect that as the number of iterations increases the chances of the fault-free sorters to converge improves, for the following reason. As the number of iterations increases, the correctly sorted lists from sorters 1-3 becomes substantially more sorted than sorter 4's list, which (on average) maintains its initial unsortedness. As a result, the approximate pseudo-energies of the four lists within pickers 1-3 suggests with increasing frequency that lists 1-3 are more sorted and thus tends to chose among those lists for subsequent operations.

V. Conclusions and Future Work

We have attempted to put the problem of making software robust to certain kinds of disturbances into a dynamical systems and control framework by investigating the example of software that sorts lists. We defined several metrics and pseudo-energy functions for potential use in stabilizing and analyzing software processes that perform sorting. Further, the case of a single sorter operating in open and closed-

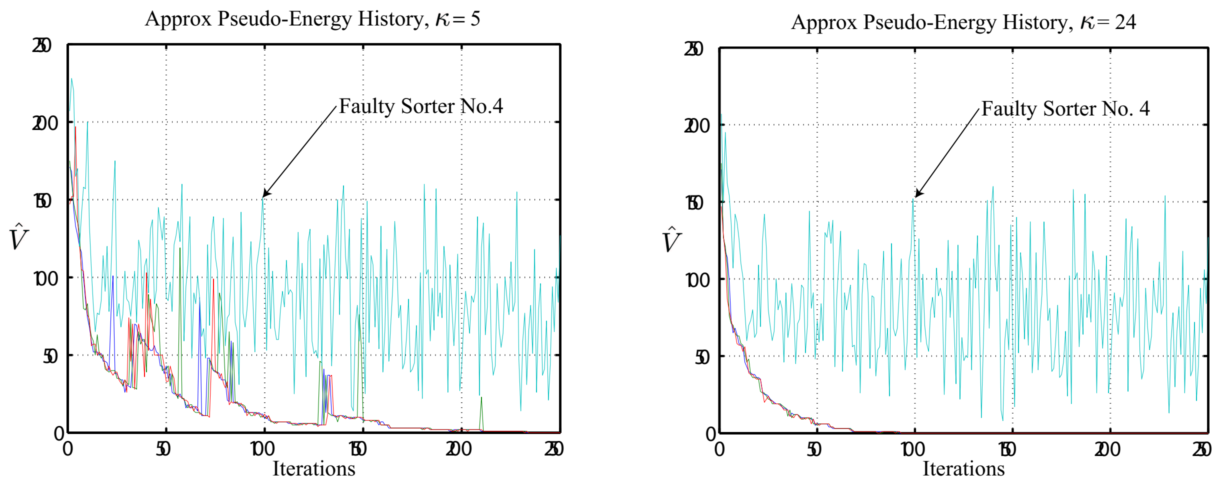

Fig. 4. Approximate pseudo-energy history for four sorters in fully connected network; Sorter 4 is faulty.

loop was thoroughly examined, and closing the loop was shown to dramatically improve the accuracy of a faulty sorter. Simulation for a network of sorters was also presented, where approximation and randomization were important components. We plan to further extend the analysis of the closed loop sorter dynamics as well as those of the networked sorters. The utility and construction of metrics and pseudo-energies as used on lists above for more general software systems will also be explored.

Underlying our approach is the assumption that we are allowed to control the number of iterations the software performs and when the iterations start. It may be of interest to develop approaches that use feedback in software environments where the iterations cannot be so controlled and stabilization must occur in real-time while the state of the software system evolves, as is the case in traditional control problems, e.g. mechanical systems. Such approaches will also be explored in the future.

Acknowledgments

The authors with to thank Richard Murray and Jason Hickey for their suggesting several of the problems we consider in this paper and Natarajan Shankar for his advice and suggestions on relating this work to other, similar, fields. We also thank our reviewers for suggesting further useful references. Partial support for this work was provided by the DARPA SEC program under grant number F33615-98-C-3613 and by AFOSR grant number F49620-01-1-0361. The first author is supported in part by the Fannie and John Hertz Foundation.

VI. REFERENCES

[1] M. Ajtai, T.S. Jayram, R. Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *34th ACM Symposium on Theory of Computing*, Montral, Qubec, Canada, 2002.

[2] A. Avizienis. *The Methodology of N-Version Programming*. John Wiley & Sons, New York, 1995.

[3] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the Association for Computing Machinery*, 42(1):269–291, 1995.

[4] A. Bogomolny. Various ways to define a permutation. Online: `http://www.cut-the-knot.org/do_you_know/Perm.shtml`.

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

[6] W.B. Dunbar, E. Klavins, and S. Waydo. Feedback controlled software systems. CDS technical report 2003-002, California Institute of Technology, 2003. Online: `http://caltechcdstr.library.caltech.edu/`.

[7] W. Feller. *An Introduction to Probability Theory and Its Applications*. J Wiley and Sons, 1957.

[8] H. A. Khalil. *Nonlinear Systems*. Printice Hall, 2nd edition, 1996.

[9] E. Klavins. A formal model of a multi-robot control and communications task. In *Conference on Decision and Control*, Hawaii, 2003.

[10] M. L. Littman. CPS130 course notes - sorting(5), Fall 1997. Online: `http://www.cs.duke.edu/mlittman`.

[11] R. Rubinfeld. *A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs*. PhD thesis, University of California, Berkeley, 1996.

[12] M. Schneider. Self-stabilzation. *ACM Computing Surveys*, 25(1), March 1993.

[13] Oliver Theel. An exercise in proving self-stabilization through Ljapunov functions. In *International Conference on Distributed Computing Systems*, Phoenix, AZ, 2001.

[14] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the Association for Computing Machinery*, 44(6):826–849, November 1997.