Feedback Controlled Software Systems
William B. Dunbar, Eric Klavins, Stephen Waydo
June 7, 2002

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 the machine on which the software is executing hang or crash. What seems to be required to address this fragility is a general means of using feedback to stabilize these systems. In this paper we develop a preliminary dynamical systems model of iterative software processes along with the conceptual framework for “stabilizing” them in the presence of disturbances. To keep the computational burden of the controllers we design low, randomization and approximation are used. We describe initial attempts to apply the model to a faulty list sorter, using feedback to improve its performance. Methods by which software robustness can be enhanced by distributing a task between nodes which are capable of selecting the “best” input to process are also examined, and 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.

1 Introduction

Software systems in general are not robust to disturbances, which may come in the form of bugs, unforeseen input, unexpected interactions with other system components (both hard- and software), and so on. A single disturbance is often sufficient to cause an instability in the system, and the software can hang or crash. The language of dynamical systems is a natural one in which to express the idea of stability, and we explore how software systems can be modeled within this framework. One a software system can be described in this way, control theory can provide a methodology by which feedback can be applied to (provably) ensure robustness to specified types of disturbances. In this paper we describe the tools needed to apply these concepts to a general iterative software system, then apply the model to a possibly faulty list sorter. Feedback is shown to improve the faulty sorter’s performance. Further gains in robustness are realized when multiple sorters which use feedback to monitor their progress are networked together and are thus able to exchange their states.

In Section 2 we explore a preliminary model of the time/space evolution of a generic 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. These modeling concepts are then investigated in Section 3 for the case of sorting algorithms. In Section 3.1 we define an appropriate metric on the group of permutations of \( n \) elements as well as a less descriptive but easier to handle quality measure for the purposes of control. Section 3.2 provides an analysis of the open-loop dynamics of a faulty list sorter using a Markov chain. In Section 3.3 we develop a simple example of the application of feedback to stabilize the sorting system of Section 3.2. In Section 4 we describe a distributed array of nodes in which each node consists of a partial sorter and a controller, and simulation results are shown. Conclusions and future directions are given in Section 5.

2 Modeling Software

A piece of software along with the data it operates on (and the hardware upon which it is run) can be modeled as a dynamic system with internal state \( x \), input \( u \), and output \( y \). Incorrect operation may manifest in the form of bugs in the software and unforseen inputs or interactions with other software components. From a robust control theory perspective, these types of uncertainty could be labeled as model (internal) uncertainty and external disturbances, respectively. For the purposes of this discussion, we refer to uncertainty in general as disturbances. A particularly interesting class of programs we investigate are iterative processes which do not run to completion; instead, they provide output after some number of steps \( n \), then use this output as their input for the next set of iterations.

If disturbances such as occasional incorrect iterations exist in this sort of system, there is no longer a guarantee that the process will converge to the correct output, or even that it will converge to any fixed point. Because iterative processes already incorporate feedback in the sense that the output of one iteration is the input to the next, applying some sort of control to this process may be useful and convenient.

2.1 States and Metrics

Evidently, to determine what actions to take to stabilize a system with feedback, we need to at least define the state \( x \in X \) of the system and a metric \( d \) on \( X \) which says when two system states are close to each other. For a software system, \( x \) may simply be a snapshot of the memory used by the software. The metric \( d \), describing the “distance” between, for example, the current state and a goal state, is also required for control, as it is a means by which the control system determines whether or not the software system is performing its assigned task. However, \( X \) is rarely a metric space (or even a reasonable topological space) in software systems, and thus some surrogate for \( d \) will be required. Thus, we also use a quality measure \( Q : X \rightarrow \mathbb{R} \) of the system state with the set \( \{ x \mid Q(x) = 0 \} \) defining the goal state and the assumption that
$Q$ does not increase in normal operation. Thus, in place of $d(x_1, x_2)$ we have $|Q(x_1) - Q(x_2)|$. In Section 3.1 we define both of these notations for lists of numbers to be used in our algorithms for sorting with faulty sorters.

### 2.2 Stability, Performance, and Robustness

Once a suitable metric has been defined, the notions of stability and performance can be formalized. Suppose that the state of the system at step $k$ is $x_k$ and put $Q(k) = Q(x_k)$. The following condition defines stability:

$$\exists \epsilon, Q_{fp}, k_{fp} : |Q(k) - Q_{fp}| < \epsilon \ \forall k > k_{fp}.$$  

This condition states that as the number of iterations (of the software system) $k$ becomes greater than some (possibly large) $k_{fp}$ which will likely depend on initial conditions, $Q(k)$ will approach a fixed-point quality $Q_{fp}$ and remain in a region of radius $\epsilon$ around it. A stronger stability condition is that of asymptotic stability:

$$\lim_{k \to \infty} |Q(k) - Q_{fp}| = 0,$$

which says that $Q(k)$ will (asymptotically) approach $Q_{fp}$ and stay there.

The performance of a system characterized by $Q$ is measured in two ways: how closely $Q_{fp}$ matches the goal state, and how quickly the process approaches $Q_{fp}$:

$$|Q_{fp}| < \delta$$

$$k_{fp} < k_{des}$$

where $\delta$ is small and $k_{des}$ is some desired convergence time.

The objective of the controllers discussed in this paper is to ensure robustness, which translates to two goals: robust stability and robust performance, which are the properties of maintaining the above conditions in the presence of classes of disturbances. An example disturbance class might include bounds on the frequency with which the software makes errors.

### 2.3 Sensors and Estimators

The state $x$ of a software process may be enormously complicated. In fact, determining the value of $Q(x)$ may be equivalent to executing the software process in question to completion. Clearly, some method of generating an approximation of $x$ or at least of $Q(x)$, will be necessary. The role of sensing and estimation in the realm of control can be related to this issue. Typically in control, theoretic sensors provide continuous information about some portion or all of the state, however it may be defined. In practice, sensors provide a discretized (approximate) version of this information. In the context of sensing software, we already have a truly discrete process, but require approximation by spatial discretization of the information available for computational efficiency. In both
contexts, the more time the sensors are given to collect information, the more reliable the control decision can be, at the price of delay. A postulated role for sensing in feedback-stabilizing a software process is detailed in Section 2.5. In this context, a generic control algorithm would incorporate a sensor which sub-samples output data according to some randomized algorithm.

In control theory, an estimator [4] uses a model of the process dynamics along with the system inputs and outputs to generate an estimate of the system state in the presence of noisy and/or incomplete data. This concept can be extended to software systems where and estimator takes the sensed information along with knowledge of the software’s inputs and nominal operation to approximate the evolving state of the system. The example investigated in Section 3.3 helps to solidify these abstractions.

2.4 Controllers

Once an estimator has been designed for a system, a controller can be designed which takes as input an estimate of \( x \) (or at least an estimate of \( Q(x) \)) and decides the next action the software should take. In controlling an iterative process, several possibilities for control action are available. First, the controller may judge that the new output is “further” from the goal state than the previous output (i.e. the quality has decreased). Assuming a nondeterministic process, the controller might instruct the software simply to perform the previous set of iterations again. Secondly, the controller may adjust the number of iterations the software performs before generating another output to be checked.

Given the probability that any single iteration is correct, and given that some processing resources are used during a state estimate and quality comparison, there will be an optimal number of iterations to perform between such comparisons. If a piece of software is very reliable, all processing resources should be devoted to running the software. As reliability goes down, checks should become more frequent to maximize the amount of progress made, or a decision to abort may be appropriate. This reliability figure, however, will not in general be known a priori, and may in fact change in time as more knowledge is gained about the performance of the system, so active control should be used to continually ensure that the system is sitting near an optimal point.

Another implementation of active control may be in tuning the fidelity of the estimators and sensors. While general estimators are often limited in some way by the amount of noise in a system or the quality of its sensors, software estimators can be made arbitrarily accurate as more processing resources are devoted to them. The estimated accuracy of the current state estimate may be used to tune the fidelity of future estimates, perhaps in the form of increasing or decreasing sub-sampling density in the sensor.

2.5 A Feedback Controlled Software Model

Figure 1 is a block diagram depicting a feedback controlled software system. At the top of the diagram is the software itself, which we consider to be the dynamic
system to be controlled. The software has a state which is some function of its internal variables and which evolves in some manner particular to the task the software performs. The number of iterations $n$ the software performs before producing output is an input coming from the controller. There is also an input port on the software which allows the controller to replace the state of the software with another state, for example to roll back a set of incorrect iterations or to incorporate new information coming in from other nodes on a network. After performing $n$ iterations, the software outputs its state $x$. For the reasons discussed above, it is likely that operating on this full state will be too computationally intensive for active control, so it is passed to a “sensor” which performs a sub-sampling operation to produce a less information rich but more useful output set $y$. The sensor takes as a control input $d$, the sub-sampling density; this density can be actively tuned by the sensor controller to suit the current needs of the system. The reduced information set $y$ forms the input to an “estimator,” which approximately evaluates the quality $Q$ of the software state, outputting its estimate $\hat{Q}$ and error bound $e$. This information, possibly along with $Q$, $e$, and $x$ from some number of agents elsewhere within a network, forms the input to the software controller. This controller can then adjust the number of iterations the software does between checks, $n$, based on some function of how the software has been affecting the quality of the state. For example, the software controller could output an $n$ that is akin to a PID function on $\hat{Q}$. In the context of distributed software systems, another function of the controller could be to compare the outputs of some number of other nodes on the network to the output from the node in question and provide the “best” data set to its software process as input for the next set of iterations.

Figure 1: A block diagram of a feedback controlled software system.
3 Application to a Single Sorter

3.1 Metrics and Measures for List Sorting

We assume that all lists generated by partial sorting are equal when viewed as sets. Therefore, a faulty sorter or disturbances may unsort the list, but the assumption requires that the list may not change as a set. A list \( a = a[1], ..., a[n] \) drawn from a universe \( \{0, 1, ..., m\} \) is a sequence of \( n \) ordered and distinct elements. Our results generally do not require the elements to be distinct, but we assume this for simplicity. 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 \( i < j \). A metric for sortedness quantifies the distance between any two lists in a given group. Quality measures, in the context of sorting, refer to functions from \( S_n \) to \( \{1, ..., n\} \) that rank lists by sortedness. For example, a (trivial) measure might output 0 if the list is sorted and 1 otherwise. Measures can be used to prove the correctness of a particular sorting algorithm, e.g. Bubblesort [3]. From the control analysis perspective, a metric is likely to prove useful in verifying things about the closed-loop behavior of a sorter/controller agent. The function of the controller as described above requires a measure of sortedness for any given list. We now give a few example measures and metrics for list sortedness.

**Definition 3.1 (Total Inversion Measure).** The total inversion measure \( m_{TI} \) of a list \( L \) is:

\[
m_{TI}(b) = \sum_{i=1}^{n} \sum_{j=i}^{n} (L[i] - L[j])
\]

where

\[
\langle x \rangle = \begin{cases} 
1, & x > 0 \\
0, & \text{otherwise}
\end{cases}
\]

In words, \( m_{TI} \) gives the total number of pairs that are out of order, with a maximum of \( \binom{n}{2} = n(n-1)/2 \). Determining \( m_{TI} \) is \( \mathcal{O}(n^2) \). A simpler measure is defined next:

**Definition 3.2 (Adjacent Inversion Measure).** The adjacent inversion measure \( m_{AI} \) of a list \( L \) is

\[
m_{AI}(b) = \sum_{i=2}^{n} (L[i] - L[i-1]).
\]

This measure gives the total number of adjacent pairs out of order, with a maximum of \( n - 1 \). Determining \( m_{AI}(b) \) is \( \mathcal{O}(n) \).

Another measure is \( n \) minus the length of the longest increasing sub-sequence. Discussion of these measures and other examples are given in [3]. A given value from a measure does not define a unique list except for the zero value. Similarly, given any list, a nontrivial distance value from a metric does not correspond to a unique list given the other list. This is of course true for any metric space;
there are a circle of points in $\mathbb{R}^2$ any given distance $\rho$ from the origin. Note that
over $S_n$, there are fewer elements that have any given value for $m_{TI}$ than the
number of elements that have the same value of $m_{AI}$. Thus, $m_{TI}$ gives a finer
rank resolution over $S_n$ than $m_{AI}$.

Two metrics, the Kendall distance $K$ and Spearman’s footrule distance $F$
are given in [1] when $m = n$ (which we have assumed to be the case). The
complexity of determining $K$ and $F$ are $O(n^2)$ and $O(n)$, respectively. A metric
discovered by the authors identifies a unique total inversion vector $q$ for any
element of $S_n$, i.e. any permutation of a given list $a$ (and also includes the case
where elements of the list are not distinct or when $m > n$):

**Definition 3.3 (Total Inversion Vector).** The total inversion vector $q : S_n \to \{1, \ldots, n - 1\}$ has $n - 1$ components $[q_1(a), \ldots, q_{n-1}(a)]$, where the $k^{th}$
component is defined by:

$$q_k(a) = \sum_{j=k}^{n} a(k) - a(j).$$

Component $k$ in this vector corresponds to the $i = k$ summation term in
the expression for $m_{TI}$. We leave off the $n^{th}$ term in the summation since it is
always zero. Our metric, which is similar to the Kendall distance in that they
are both based on the total number of inversions, is then defined by:

$$d(q(a_1), q(a_2)) \triangleq ||q(a_1) - q(a_2)||,$$

where $|| \cdot ||$ can be taken as any norm on $\{1, \ldots, n - 1\}^{n-1}$. It follows that
$m_{TI}(b) = d(q(b), 0)$ given some $0 \in \{1, \ldots, n - 1\}$, using the 1-norm. Proving
that $d$ is a metric includes the uniqueness of the total inversion vector for every
permutation of a given list.

### 3.2 Open-loop Behavior

In order to explore the issues involved in stabilizing and improving the per-
formance 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 quality at time $k$ is taken to be the total inversion measure
of the list:

$$Q(k) \triangleq m_{TI}(L(k)),$$

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

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

Denoting the current state by $q(k) = m_{T1}(L(k))$, a state transition matrix of dimension $m + 1 \times m + 1$, where $m = Q_{\text{max}}$, is obtained. Note that swapping an adjacent pair (with distinct values) will always increment or decrement $m_{T1}$ by 1. The state transition probabilities are:

$$
\begin{align*}
T_{q,q} &= (1 - d) \\
T_{q,q-1} &= pd, \ 1 \leq q < m \\
T_{q,q+1} &= (1 - p)d, \ 1 \leq q < m \\
T_{0,1} &= d \\
T_{m,m-1} &= d \\
T_{q,q+\delta} &= 0, \ \forall \delta > 1
\end{align*}
$$

The stable left eigenvector of this matrix describes the long-term distribution of the list quality $q$. Following the method of [2], we have the following proposition:

**Proposition 3.1.** The stable left eigenvector of the state transition matrix $T$ is given by:

$$v = \left[ 1, \frac{1 - p}{p}, \ldots, \frac{(1 - p)^{i-1}}{p^i}, \ldots, \frac{(1 - p)^{m-2}}{p^{m-1}}, \frac{(1 - p)^{m-1}}{p^{m-1}} \right]$$

or:

$$
\begin{align*}
v_0 &= 1 \\
v_i &= \frac{(1 - p)^{i-1}}{p^i}, \ 1 \leq i < m \\
v_m &= \frac{(1 - p)^{m-1}}{p^{m-1}}
\end{align*}
$$

**Proof:** It is sufficient to show that $vT = v$: 

\[\]
\[(vT)_i = \sum_{j=1}^{m} v_j T_{ji}\]

\[(vT)_0 = 1(1 - d) + \frac{1}{p}pd = 1 = v_0\]

\[(vT)_1 = 1d + \frac{1}{p}(1 - d) + \frac{1 - p}{p^2}pd = \frac{1}{p} = v_1\]

\[(vT)_i = \frac{(1 - p)^{i-2}}{p^{i-1}} (1 - p)d + \frac{(1 - p)^{i-1}}{p^i}(1 - d) + \frac{(1 - p)^i}{p^{i+1}}pd = \frac{(1 - p)^{i-1}}{p^i} = v_i, 2 \leq i < m - 1\]

\[(vT)_{m-1} = \frac{(1 - p)^{m-3}}{p^{m-2}} (1 - p)d + \frac{(1 - p)^{m-2}}{p^{m-1}}(1 - d) + \frac{(1 - p)^{m-1}}{p^{m-1}}d = \frac{(1 - p)^{m-2}}{p^{m-1}} = v_{m-1}\]

\[(vT)_m = \frac{(1 - p)^{m-2}}{p^{m-1}} (1 - p)d + \frac{(1 - p)^{m-1}}{p^{m-1}}(1 - d) = \frac{(1 - p)^{m-1}}{p^{m-1}} = v_m\]

so:

\[vT = v\]

and \(v\) is a stable left eigenvector of \(T\).

Normalizing \(v\) to have a 1-norm of unity we obtain the long-term probability distribution of \(Q\):

\[\eta = \sum_{i=0}^{m} v_i,\]

\[v' = \frac{v}{\eta}\]

The weighted sum of the entries of \(v'\) is the asymptotic expected value of \(Q\), which we call the fixed point quality \(Q_{fp}\):

\[Q_{fp} = \lim_{k \to \infty} E[Q(k)] = \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]}.

Figure 2 depicts the dependence of \(Q^*\) on \(p\) for a list of length 10. Note that the curve drops sharply around \(p = 0.5\), and that relatively favorable values of \(Q_{fp}\) are reached even for fairly low \(p\), in the range of 0.6 and greater.

Figure 3 is a plot of the Markov chain-predicted time history, the predicted \(Q_{fp}\), 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.

### 3.3 Closing the Loop

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. After each sorting iteration $k$, the checker picks \( l \) random pairs and calculates $Q_{\text{pre}}(k)$, the number of subsampled pairs that are out of order. The checker then rejects the sorting step if $Q_{\text{pre}}(k) \geq Q_{\text{pre}}(k-1)$, i.e. $L(k+1) = L(k)$. The accuracy of the checker is derived as follows: the sample space of the checker consists of \( \binom{n}{2} \) pairs. Let a given pair have value 1 if it is out of order, and 0 if it is in order (or if the values are the same). Again use:

$$Q(k) \triangleq m_{\text{TR}}(L(k)).$$

If $Q(k) = b$, the sample space of the checker consists of $b$ out of order pairs and \( \binom{n}{2} - b \) in order pairs. In order for $Q_{\text{pre}}$ 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 the number of ways it can pick $c$ of the $b$ out of order pairs times the number of ways it can pick $l - c$ of the \( \binom{n}{2} - b \) in order pairs divided by the number of ways it can pick $l$ of the \( \binom{n}{2} \) total pairs:

$$P[Q_{\text{pre}} = c \mid Q = b] = \frac{\binom{b}{c} \binom{n}{2} - b}{\binom{l}{c}}.$$
Two probabilities are used to characterize the checker - the probability $r_1$ that it recognizes a good step as improving $Q$ and the probability $r_2$ that it recognizes a bad step as making $Q$ worse. The probability that the checker recognizes a correct step is the probability that $Q_{pre}$ improves given that $Q$ improves, and is a function of $b$ and $l$:

$$r_1(b) = P[Q_{pre}(k) < Q_{pre}(k-1) \mid Q(k-1) = b, Q(k) = b-1].$$

Because $Q_{pre}(k)$ and $Q_{pre}(k-1)$ are separate measurements, they are independent random variables, and:

$$P[Q_{pre}(k-1) = c_1, Q_{pre}(k) = c_2 \mid Q(k-1) = b, Q(k) = b-1] = P[Q_{pre}(k) = c_1 \mid Q(k-1) = b]P[Q_{pre}(k) = c_2 \mid Q(k) = b-1]$$

$$= \frac{b}{c_1} \left(\frac{m-b}{t-c_1}\right) \left(\frac{b-1}{c_2}\right) \left(\frac{m-(b-1)}{t-c_2}\right) \left(\frac{m}{t}\right)^2.$$
where again $m = \binom{n}{2}$. Summing the above expression over all $c_1, c_2$ with $c_2 < c_1$ gives the needed probability:

$$r_1(b, l) = P[Q_{pre}(k) < Q_{pre}(k-1) \mid Q(k-1) = b, Q(k) = b-1]$$

$$= \binom{m}{l}^{-2} \sum_{c_1=1}^{m} \binom{b}{c_1} \binom{m-b}{l-c_1} \sum_{c_2=0}^{c_1-1} \binom{b-1}{c_2} \binom{m-(b-1)}{l-c_2}.$$

Figure 4 is a plot of $r_1$ as $l$ ranges from 1 to $\binom{n}{2}$ for a list of length 10 with $Q = 23 \approx \binom{n}{2}/2$.

Similar reasoning leads to the probability that the checker recognizes a bad step as making $Q$ worse (or at least failing to improve $Q$):

$$r_2(b, l) = P[Q_{pre}(k) \geq Q_{pre}(k-1) \mid Q(k-1) = b, Q(k) = b+1]$$

$$= \binom{m}{l}^{-2} \sum_{c_2=0}^{m} \binom{b+1}{c_2} \binom{m-(b+1)}{l-c_2} \sum_{c_1=0}^{c_2} \binom{b}{c_1} \binom{m-b}{l-c_1}.$$
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 above we have:

$$
T_{q,q} = (1 - d) + pd(1 - r_1) + (1 - p)dr_2, \quad 1 \leq q < m
$$
$$
T_{q,q-1} = pdr_1, \quad 1 \leq q < m
$$
$$
T_{q,q+1} = (1 - p)d(1 - r_2), \quad 1 \leq q < m
$$
$$
T_{0,0} = (1 - d) + dr_2
$$
$$
T_{0,1} = d(1 - r_2)
$$
$$
T_{m,m} = (1 - d) + d(1 - r_1)
$$
$$
T_{m,m-1} = dr_1
$$
$$
T_{q,\delta} = 0, \quad \forall \delta > 1
$$

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 $Q_{fp}$.

Figure 5 is a plot of the Markov chain-predicted time history, the predicted $Q_{fp}$, 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 $Q_{fp}$ drops quickly once $l$ becomes larger than $\binom{n}{l}/2 \approx 23$; further increasing $l$ increases the rate at which $q$ approaches $Q_{fp}$. The actual sorter performance again closely matches that predicted by the Markov chain analysis.

4 A Network of Sorters

Let $G = (V, E)$ be a graph where $V = \{1, ..., n\}$ and $E \subseteq V \times V$. The set of vertices $V$ are intended to refer to the indices in a network of sorters, the edges in $E$ to the connections between them. Let $\mathcal{L}$ denote the set of all lists over a suitable domain. The state of sorter $i$ at time $k \in \mathbb{N}$ is a list $L_i(k) \in \mathcal{L}$. The state of the network is then an $n$ dimensional vector of lists. The operation of each sorter is to attempt to partially sort one of the lists incoming from its neighbors. In particular, it chooses to partially sort the incoming list it believes is already the best sorted, arriving at this belief by applying an approximate quality measure to each list and taking the apparently best one.

To each sorter $i$ we associate a sorting function $\text{sort}_i$ and a “picking” function $\text{pick}_i$ where

$$
\text{sort}_i : \mathbb{N} \times \mathcal{L} \rightarrow \mathcal{L}
$$
$$
\text{pick}_i : \mathbb{N} \times \mathcal{P}(\mathcal{L}) \rightarrow \mathcal{L}.
$$

Here, $\mathcal{P}(\{X\})$ refers to the power set of the set $X$. 

13
The functions \textit{sort}$_i$ and \textit{pick}$_i$ are supposed to make random choices. To model this we suppose each is equipped with a pseudo random number generator that takes as input a natural number \(k\) and returns the \(k\)th pseudo random number, or a pair of numbers, etc. as required. Thus, each function associated with a sorter takes as its first argument the current time step.

The sorting function operates by randomly choosing an pair of elements from its second argument. If the elements are out of order, it flips them with some probability \(p_i\) (which we model with the pseudo random generator, although the intent is really to model unforseen bugs in the sorter). If the elements are in order, it flips them with probability \(1 - p_i\). Thus \(p_i = 1\) implies a perfectly good sorter and \(p_i = 0\) implies a perfectly bad sorter.

The picking function evaluates a quality approximation on each list in its second argument and returns the list with the best quality. That is

\[ \text{pick}_i(k, L) = L \]

where

\[ \tilde{Q}(k)(L) > \tilde{Q}(k)(L') \text{ for all } L' \in L - \{L\}. \]
The approximate quality measure $\tilde{Q}(k)$ when applied to $L \in \mathcal{L}$ chooses $\kappa_i$ (a constant) pairs from its argument and returns the number of them that are out of order, thereby approximating $m_{TI}(L)$.

Given an initial state $[L_1(0), ..., L_n(0)]$ the sorting process proceeds according to

$$L_i(k+1) = \text{sort}_i(k+1, \text{pick}_i(k+1, \{L_j(k) \mid (i,j) \in E\})).$$

We usually view the output of a network of sorters at time $k$ as the state of the sorter with the best quality at time $k$. A network is thus said to converge if the quality of its output converges. Equivalently, a network is said to converge if

$$x(k) \triangleq \min\{m_{TI}(L_i(k)) \mid i \in V\}$$

converges. Whether $x(k)$ is stable and what is its expected value of $x(k)$ (when the network is considered to be a true random process) are the main indicators of the performance of the network.

We are particularly interested in the performance of $x(k)$ in situations where one or more of the sorters in the network is imperfect and $\kappa_i$ is fairly low (not computationally intensive). Will the outputs of bad sorters propagate through the network or will the pickers be good enough to weed out bad lists?

Although we have not yet rendered the class of networks we have described amenable to analysis, we have begun to investigate their performance in simulation studies. In the following, we consider a fully connected, four node network ($V = \{1, 2, 3, 4\}$ and $E = V \times V$) with $L_i(0)$ equal to identical, randomly chosen lists of length 30. Figures 6(a) and 6(b) shows the outputs of two runs of the network. Plotted, for each $i \in V$ is $Q_i(k) \triangleq m_{TI}(L_i(k))$ versus $k$. In the first run, $p_i = 1$ for all $i \in V$. In the second $p_1 = 0.5$ and $p_i = 1$ for all $i \in V - \{1\}$. In each run $\kappa_i = 10$ (so that only about 2.3 % of pairs are used in the quality

Figure 6: Two simulations of a network of sorters. In (a), all sorters are perfect. In (b), one of the sorters is bad. In both runs, the quality of the good sorters converges.
approximations). In the first run, the quality of all sorters converge more or less in lock step to 0 (perfectly sorted). In the second plot, the quality of sorter 1 varies wildly as the sorter performs essentially random operations on the list. The other sorters for the most part are unaffected except for short, ever smaller disruptions in their quality. The attenuation of these catastrophes and the converges of the network are of course what we desire and we hope to be able to demonstrate these properties analytically at a future time.

We are also investigating the time it takes to converge as the parameters change. Figure 7 shows the number of iterations required to sort a list of length 30 as a function of $\cdot i = \cdot$. As the approximation used by the picker functions becomes more exact, the number of iterations evidently decreases.

5 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. Analogs of control theoretic sensors, estimators, and controllers within software systems were discussed, and methods for applying feedback to such systems were discussed. The need for metrics or their surrogates for stabilizing software processes has been described. A study of the particular case of sorting systems was initiated, and the case of a single sorter operating in open-loop was thoroughly examined. We plan to further extend the analysis of the closed loop sorter dynamics as well as those of the networked sorters. We plan also to further develop the notion of metric and measure, used on lists above, to more general notions for general software systems.
References


