Fault Tolerance and Security for Power Transmission System Configuration with FACTS Devices

B. McMillin
Department of Computer Science

M. L. Crow
Department of Electrical and Computer Engineering

University of Missouri-Rolla
Rolla, MO 65409

Abstract

One of the most promising decentralized network controllers is the family of power electronics-based controllers, known as "flexible AC transmission system" (FACTS) devices. These devices locally modify the topology of the system by rapid switching. By the choice of switching patterns, these devices can achieve a variety of power system objectives, such as voltage support, oscillation damping, and stability improvement. This ability can be used for not only reliability objectives such as increasing the maximum throughput power, but can also be used to adjust power flow for economic reasons. In the power system restructured environment, it is foreseeable that power flows will be adjusted throughout the system to maximize economic objectives. These devices, however, are relatively new and few are currently beyond the prototype stage, therefore their wide-spread impact on the transmission network has not yet been thoroughly analyzed. While these devices offer increased network power flow controllability, the decentralized nature of their actions may cause deleterious interactions between them. In this paper, we propose to utilize flexible topology FACTS devices in developing distributed control strategies to i) *detect and mitigate intentional or unintentional cascading failures,* ii) *develop operating strategies that can automatically adjust to changing economic and physical environments, and* iii) *develop interaction policies to mitigate counterproductive actions.*

1 Introduction

The bulk power system grid is arguably the largest man-made interconnected network in existence. The sheer size of the power network makes control of the grid an extremely difficult task. Compounding this, the network is comprised of both continuous and discrete controls which typically operate in a disassociated fashion. Cascading failures are the most severe form of contingency that can occur in a power system. A cascading failure occurs when one contingency on the system induces a second contingency. If not immediately mitigated, the second contingency may lead to a third contingency, fourth contingency, and so on until load must be shed to stabilize the system. A typical cascading failure involves transmission line overloads. In a system where there is a large directional power flow from one region to another, the loss of a single transmission line can instigate a cascading failure. If the first transmission line is lost, the power it carried must be shunted across the remaining parallel transmission lines. This may cause one or more of these transmission lines to overload, thereby directing still more power across fewer and fewer lines. At some point, it is no longer possible to serve the load with the existing network topology. If load is not shed in a timely manner, generators may trip as a result of under-frequency protective relays. Distributed FACTS controllers in the network can help alleviate the overload problem by directing extraneous power flow away from highly loaded lines.

Grid control has historically been decentralized due to geographic and regulatory constraints. However, with the expansion of communication technologies, including optical and wireless communication, it is rapidly becoming possible to incorporate real-time decision-making processes over a widespread area using a variety of information. Within this framework, controllers will be able to broadcast and receive operating status information from other entities throughout the system. While this ability creates considerable potential for improved operation, it also poses considerable difficulties in determining communication protocols, selection of necessary information exchange, and coordination of actions, among other difficulties. The family of FACTS devices holds considerable promise as network-embedded controllers. The problem then becomes one of identifying the sizing and placement of these controllers, and how to coordinate their actions.

Taken together, the computational elements of FACTS devices and generators in the power system network comprise a distributed system [13]. In a distributed system, processes cooperate in the solution of a distributed algorithm. The processes run on autonomous physical processors and exchange messages over the communication network. The difficulty in creating an effective distributed algorithm is the lack of a consistent, global state of the system. Computational decisions must be made locally, at each process, using (possibly inconsistent) state information obtained by messages from other processes. Processors and/or interprocessor connections may fail or be compromised during the operation of the system. In this paper, the processors will execute a graph-theory-based max-flow distributed algorithm to identify critical transmission corridors and adjust power flow to avoid cascading failures.

The remainder of this paper is organized as follows. In the Section 2, the flow in a power grid is modeled as maximum-flow in a digraph. In Section 3, decentralized configuration control using this model is discussed. Section 4 presents our decentralized control algorithm that functions in the presence of faulty (malicious) component behavior.

2 Power Flow as a Maximum Flow Problem

The power flow in a power grid is modeled as a directed network flow problem with a directed graph $G(N, A)$ modeling the power network. The set of nodes, N, correspond to the bus bars (or simply buses) of the power network. Power flow between nodes $n_i, n_j \in N$ is represented by an arc $a_{ij} \in A$. Each arc is assigned a weight, u_{ij}, denoting the current (or maximum allowed) power flow over that arc in the network. For each node, $n \in N$, due to Kirchoff's Current Laws (KCL)s, power in must equal power out. An example of a power network modeled in this way is shown in Figure 1.

The power system active power loads are shown as arrows from each bus, and the directed active power flows are shown for each transmission line. This power system may be equivalently represented as a directed graph as shown in Figure 2. Note that all power flows from the source (s) to the termination (t). The termination may be considered the network "ground" node and the source is a "virtual source" node connecting all the power generators. The nodes s and t, together with G, form a graph $G'(N', A')$.

This model is useful in the case when a line is lost due to a failure, and the resulting power flow stresses the network. Too much load is drawn over lines of inadequate capacity and one-by-one the lines overload and trip offline. A method is needed to rapidly re-balance the power by either shedding some loads to maintain power flow to the remain-

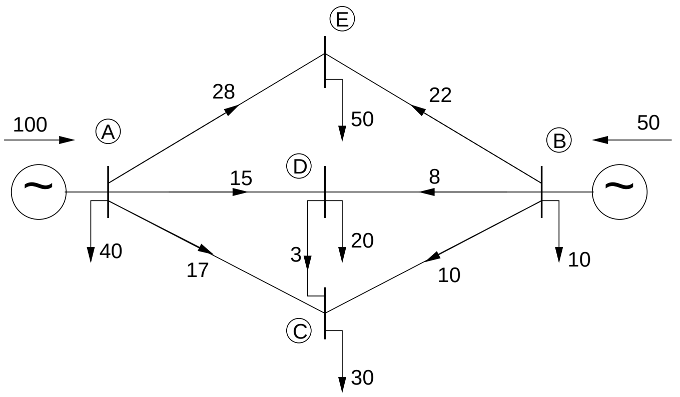

Figure 1. Example power system network

ing loads or directing the power flow across transmission corridors with greater capacity. The solution to this problem can be approached by modeling it as determining the maximum flow (max-flow) in a digraph [5]. Algorithms for performing this task may be found in the literature of operations research (for example, see [1]). For simplicity, we use the algorithm of [4]. The algorithm works by successively assigning flows $f(a_{ij})$ to arcs along a directed path from s to t until no more flow can be added. The algorithm is shown in Figure 3.

Applying the max-flow algorithm to determine the contribution of the sources to each line flow and/or load for the network of Figure 2 yields the process shown in Figure 4. Each path from source to termination is enumerated with the maximum allowable power flow. Note that at every node, KCL (the power equivalent that power in must equal power out) is satisfied. The maximum power flow across any node-sequence path (denoted by Δ) is the minimum of all of the allowable power flows. By tracing a path from source to termination, it is possible to determine the contribution of each source to each line flow and load in the network.

When an arc (line) is lost, perhaps due to a transmission line outage on the arc $a_{ij} \in A$ between nodes $n_i, n_j \in N'$, the arc weight u_{ij} becomes 0. Then the flow in the network from s to t is re-balanced using the same max-flow algorithm on the network. Since flow can be bidirectional in a power grid, each arc (except for the connections to s and t) in Figure 2 has an anti-parallel arc with the same capacity, but with zero initial flow, and in the reverse direction. In reconfiguration, flow may reverse direction and use some of these anti-parallel arcs as well as use the forward arcs, but to a lesser capacity.

In the running example, if the line $B - D$ is lost, the transmission network is no longer able to service the load. If load is not immediately shed, the system will suffer from a collapse of voltage and ultimately cascading blackouts.

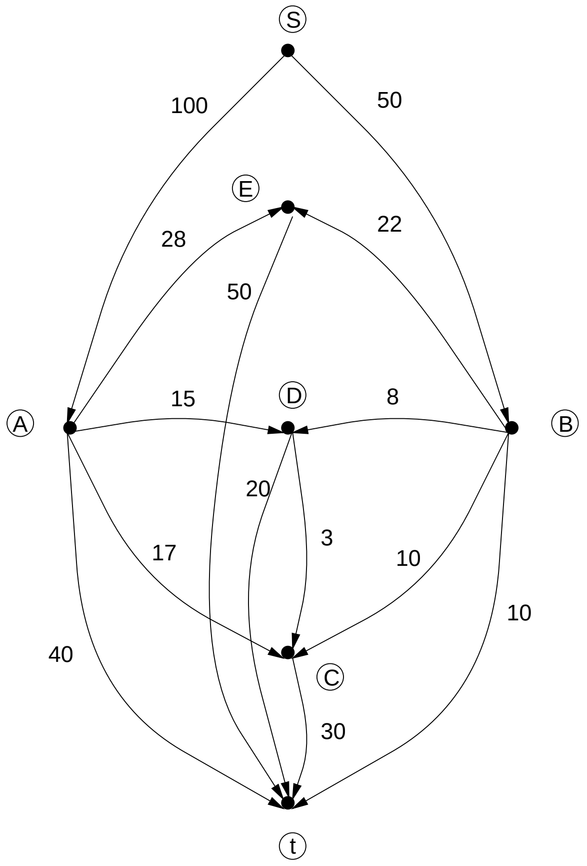

Figure 2. Power network directed flow graph

Definitions:

- Forward Labeling of a_{ij}:
  If n_i is labeled and n_j is not and $u_{ij} > f(a_{ij})$, n_j gets labeled and $\Delta = u_{ij} - f(a_{ij})$
- Backward Labeling of a_{ij}:
  If n_i is labeled and n_j is not and $f(a_{ij}) > 0$, n_j gets labeled and $\Delta_{ij} = f(a_{ij})$

Algorithm:

1. Assign an initial flow ($f(a_{ij}) = 0$ for all arcs in A)
2. Mark s labeled and all other nodes unlabeled.
3. Search for a node that can be labeled by either a forward or backward edge. If none found, flow is maximum, stop. If $n_j = t$ go to step 4, otherwise, repeat this step.
4. Backtrack the path computing the minimum Δ_{ij} used. If a_{ij} used a forward labeling, $f(a_{ij}) \leftarrow f(a_{ij}) + \Delta_{ij}$. If a_{ij} used a backward labeling, $f(a_{ij}) \leftarrow f(a_{ij}) - \Delta_{ij}$. Go to step 2.

Figure 3. Ford and Fulkerson Algorithm for Max-Flow

However, if alternate paths with sufficient power flow capacity are available, then it is possible to continue to service the load if the network can be reconfigured to bypass the lost transmission capability. This reconfiguration is not intended to be the addition or deletion of transmission lines, but is rather the change of the line characteristics such that critical lines can carry more (or less) power flow as required. This impedance change can be accomplished using high power electronic switching devices that can impact the impedance of a transmission line by injecting an adjustable voltage source in series with the line. By adjusting the magnitude and phase angle of the series voltage source, the apparent impedance of the transmission line may be varied. This change in impedance may by translated into a similar change in maximum power flow capacity across the line.

The max-flow algorithm may be implemented in real-time by taking the line capacity as the scheduled power flow (satisfying a load flow solution) for that line. In the event of a contingency, these capacities are the initial capacities for the max-flow algorithm. If the load cannot be served under the current operating scenario, then the max-flow algorithm is solved to determined the minimum number of line capacity changes (implemented by the FACTS devices) that are required to continue to satisfy the load.

For the loss of line $B-D$, the resulting graph and flow is shown in Figure 5. Loads off bus D must be reduced from 20 to 15 and 3 to 0, and the load off bus C must be reduced to 27 to avoid exceeding the power carrying capacities of the other lines in the graph.

3 Decentralized Configuration Control

3.1 Related Work

Much of the previous work in reconfiguring power flow has concentrated on determining *a priori* how power is distributed in the network from each source. The work of [3] traces the flow of power to load by solving nodal flow equations. This idea is further explored in [15] by selectively removing power sources to identify flow in the network. Both these efforts are targeted towards identifying flows for economic reasons, but the same principles can be extended to configuration management. Our work differs in that we compute, in a distributed real-time manner, the actual flow balance needed to prevent cascading failures. The configuration control is executed by embedded control devices

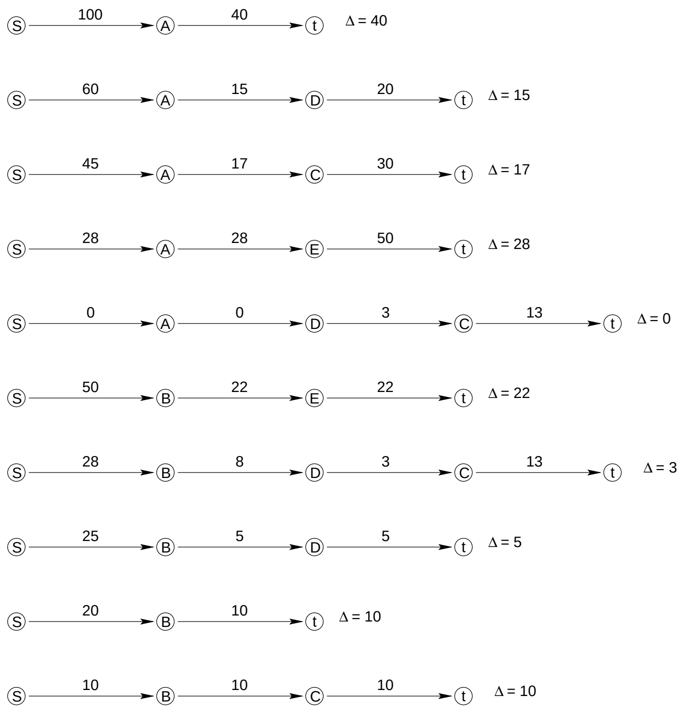

Figure 4. Max-flow process for the example network

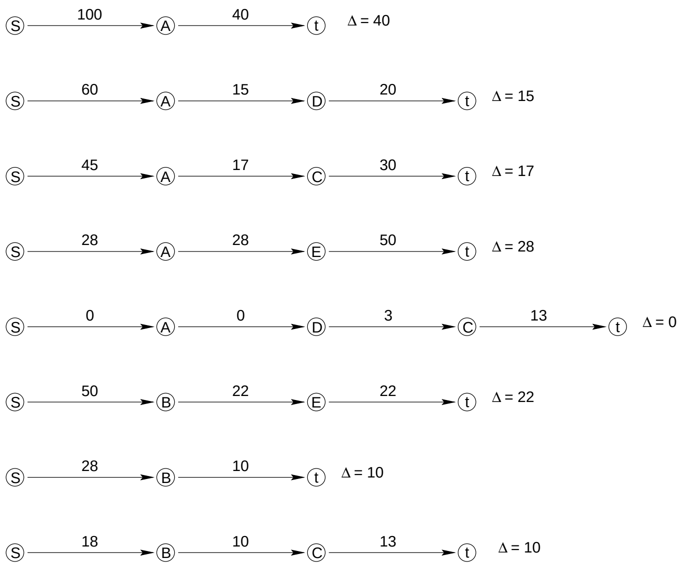

Figure 5. Reconfigured Flow graph

rather than a static allocation or switching.

In this paper, we present a basic distributed algorithm and the proposed method of providing the required fault tolerance of the system.

3.2 Distributed Implementation

While the Maximum-Flow algorithm can be run from a central site, there are several problems with this centralized approach. The first is that communication with the central site may be compromised due to hardware failures or intrusions into the communications network. The central site may also be subject to failure. Finally, the computational cost of a max-flow algorithm written this way is $O(\mid N \mid^2 \mid A \mid)$. However, more recent algorithms can reduce this to nearly $O(\mid N \mid\mid A \mid)$ [1].

Decentralized, or parallel, algorithms for the solution of max-flow have been proposed by many researchers. In [1], a parallelization using a shared-memory multiprocessor results in nearly linear speedup (performance improves linearly as the number of processors increases). A distributed algorithm is presented by [10] with a time complexity of $O(\mid N \mid^2 \sqrt{\mid A \mid})$ using a processor at each node.

In this application, it is unlikely that a processor would exist at each node. However, processors exist at the FACTS devices and at the power generating sites. An easy modification to the Ford and Fulkerson algorithm is to explore augmenting flow paths, simultaneously, from each source by passing probe and response messages over the communications network. Winning paths are selected for the resulting power flow on a "first discovered" basis. We present a simplified version of this algorithm in Figure 6 that requires a FACTS controller at every node, although this can be easily extended to have a single FACTS device/processor control multiple lines and flows.

The best case complexity of the algorithm is when the number of power sources is the same as the number of nodes in the system. Since each flow augmentation takes, at most, one step in this case, the algorithm trivially terminates in one parallel step. In the case of a single power source, the complexity degrades to that of Ford and Fulkerson($O(\mid N \mid^2 \mid A \mid)$. Further analysis is required to obtain the complexity of an "average" case.

4 Fault Tolerance

The paper has concentrated on the coordination requirements of the FACTS devices in configuration control. Even more critical than the basic algorithms are the software/hardware embedded systems aspects of safety, liveness, fault tolerance, or security addressed by this paper. Experience with the TCAS II Air Traffic Collision Avoidance System [9] has shown that the engineering of software for such systems is an extremely involved and error-prone process and that formal specification and formal methods alleviate many of the ambiguities that lead to software errors.

```
Assign an initial flow ($f(a_{ij}) = 0$ for all arcs in $A$)
for each power source $s_i$, $a_{ss_i} \in A'$, do in parallel

1. Mark $s$ labeled.

2. Label node $s_i$ as a forward edge along $a_{ss_i}$ with $\Delta$. If
   this cannot be done, stop.

3. Explore each outgoing arc with a probe($\Delta$) labeling by
   a forward or backward edge.

4. Wait for messages to return. If the return message is

      • response($s_i, \Delta$), set $f(a_{ss_i}) \leftarrow f(s_{ss_i}) + \Delta$ If
        $\Delta > 0$ Go to step 2, else, stop.
      • blocked($s_i$), all paths to $t$ are blocked by other
        searches, go to step 2.

for each FACTS device, $n_j$,

1. wait for a probe($s_i, \Delta_i$)

2. If the arc is already labeled, return blocked($s_i$) to the
   sender of probe.

3. Explore each outgoing arc as a forward or backward
   edge sending a probe of $\Delta$=min($\Delta, \Delta_i$).

4. Wait for messages to return. If the return message from
   node $n_k$ is

      • response($s_i, \Delta_i$), set $\Delta$=min($\Delta, \Delta_i$)
          – If $a_{jk}$ used a forward labeling, $f(a_{jk}) \leftarrow$
            $f(a_{jk}) + \Delta$.
          – If $a_{jk}$ used a backward labeling, $f(a_{jk}) \leftarrow$
            $f(a_{jk}) - \Delta_i$.
        clear label and send response($s_i, \Delta_i$) to sender of
        probe.
      • blocked($s_i$), all paths to $t$ are blocked by other
        searches, clear label and send blocked($s_i$) to
        sender of probe.
```

Figure 6. Distributed Algorithm Using Multiple Sources for Max-Flow

Previous work has developed a run-time system for ensuring these formally specified properties hold in the presence of failure, incorrectness, and security intrusions for several problem domains. Analysis of these types of errors within the context of power grid configuration control yields:

- Safety Violations - detect violations of safe states of the system (states which will lead to cascading failures)

- Liveness Violations - detect violations of timing requirements such as message delivery and eventuality of finishing reconfiguration.

- Fault Tolerance - detect system safety and liveness violations in the presence of failed or degraded computation hardware and/or communications.

- Security Violations - detect malicious intent such as spurious initiations of messages, false identities, denial of service attacks that lead to safety and liveness violations.

For each of the above types of errors, a corresponding specification of correctness exists. For a system to be correct and free of these errors, it must perform within the bounds imposed by each type of specification. Power systems are particularly suited to this treatment; correct operation is succinctly specified by using formal methods of mathematics such as KCL and differential equation models of generators. In this paper, we will create fault-tolerant and secure distributed algorithms by ensuring the formal specification of power systems operation is satisfied during system operation.

These specifications are not substantially different from each other [12], as they can all be treated as aspects of correctness that need to be enforced over the system's behavior. The idea of treating system properties uniformly, using similar concepts and tools for each type of property, appears in different forms in the literature (see for example Jacob [7] and more recently Rushby [11]), but it is usually directed at methods for proving a system's design or implementation is correct, in a formal treatment, rather than the run-time focus of the proposed work.

4.1 Traditional Approaches

The traditional approach to hardware fault tolerance is to explicitly replicate a calculation at a number of computing sites and vote on the result. This introduces explicit hardware redundancy overhead and, in general, requires rigid synchronization of processes not present in power grid control. Software fault tolerance typically consists of executing multiple versions of the software (written from the same specification) to catch residual design flaws. Work has shown, though, that little benefit is gained from this approach once the system is installed.

By contrast, compliance to formal specifications can be approached either at development time, through program verification, or at run time, through distributed run-time assertion checking, using one or both.

Formally, a distributed program is verified by considering all its possible executions (events or traces) and showing that the specifications hold on each possible trace, or that there is no trace for which any specification is violated. In fact, verification shows that the program conforms to a specification if the underlying system executes correctly, but says nothing about abnormal situations in the system. Thus, the idea is not relevant to providing fault tolerance, directly (although the fault tolerance techniques, themselves, can be formally verified).

Distributed run-time assertion checking, by contrast, focuses only on the unique execution in progress when the program is being checked, disregarding any other traces or interleavings that might happen at other times. Focusing only on the execution in progress guarantees that the current execution meets its specification, without regard to underlying hardware or system confidence. Unlike testing, though, in run-time assertion checking these checks become part of the running system. The advantage of run-time checking is that any hardware, software, or security errors are checked at run-time, under all circumstances. Its relation to verification is solely as a source of formally-specified assertions.

The general programming model we adopted for this type of environment is CSP[6], in which programs for the whole system are composed of a set of communicating sequential processes, with each single process having a local state, and the global program being seen as the parallel composition of the independent processes.

We have used the ideas of CCSP[2], a previously developed in-house tool that offers CSP-like syntax on top of a C environment and provides for distributed run-time assertion checking. In CCSP, concurrent processes can be created to handle different tasks in the system. Communication mechanisms are available for processes to exchange messages and program states. Further, assertions embedded in the code can be evaluated when the code is executed and action can be taken depending on their truth values. The mathematics behind the power flow application also provide a specification of a correct solution. Consider, as one of these constraints, the requirements of KCL and balanced flow at each node[1].

$$\mathrm{sum} = \sum_{\mathrm{j}} \mathrm{f(a_i j)} = 0, \text{ for all } \mathrm{n_i} \in \mathrm{A} \tag{1}$$

[1]This is a simplified form of the actual assertion that does not take into account encountering an in-progress flow augmentation

Thus, any flow augmentation, upon completion, must satisfy (1). Using CCSP to carry out the send and receive operations, and embedding the following check in each of the FACTS and power station's code, yields an assurance that the algorithm maintains balanced flow:

```
global $f(a_{ij}$;
for each $n_i \in A$
        assert(sum = $\sum_j value(f(a_{ij})) = 0$);
```

The "global" declaration, declares global state information that each process may use to reason about the correctness of other processes in the system. Since this is a distributed system no actual global variables exist so CCSP diffused copies of the global variables throughout the system by piggybacking them on existing message traffic, in the presence of faulty processes and processors. Each process executes "assert" statements on this diffused state information and local state information to be able to state that an assertion must be true at the point in the program that the `assert` statement is located. During run-time, it will be checked to see if the assertion in the "assert" statement can be evaluated to true.

Since the above "assert," checks not only the node at which the code is executing, but also all other nodes, each $f(a_i j)$ must be declared as a CCSP `global`. This means that whenever a message is sent from a node n_j to a node n_i, all known values of global variables are also sent along. In this way, the values of the global variables diffuse through the system and all processors check the computations of all other processors with a minimum of overhead.

As an example of the message diffusion, consider the simple network and its flow augmentation of Figure 1. Let nodes B and C be faulty. Moreover, these two nodes are in collusion to overload the line between them by attempting to push flow of 20 over the line B-C. The flow augmentation paths are shown in Figure 7.

Since there are two sources, both send probes simultaneously. The probe through A labels arcs s-A, A-C, and C-t. Then the probe from B arrives. Finding C-t already labeled, it augments the arc B-t. Both probes return to the sources and two new probes are initiated. Since B is faulty, it attempts to overload the arc B-C with a flow of 20. The node C, in this scenario, is also faulty and attempts to hide this overload by augmenting the flow to t by 13. Node C then returns a response to B. Before B can respond to the source, the probe through D and A arrives. Finding the path to t full (capacity is 30), it returns a blocked message to D. The blocked message, however, carries with it C's sum of 7. D checks this using its `assert` statement and signals an error has occurred.

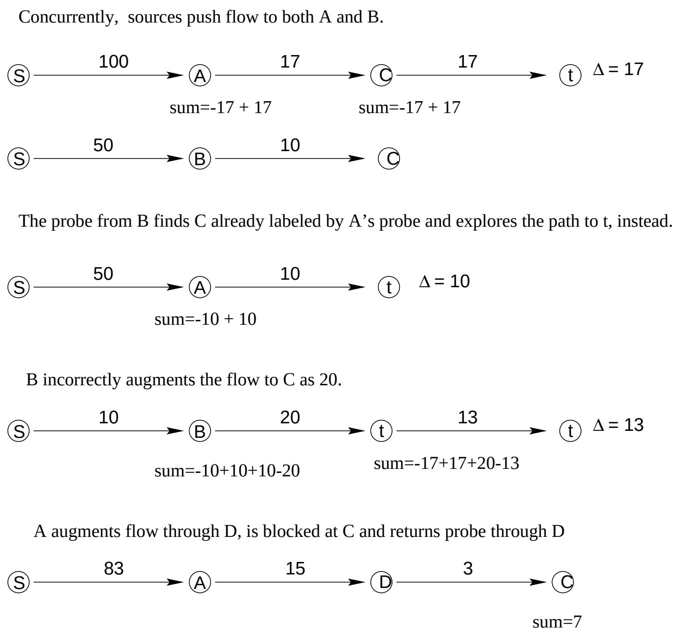
Concurrently, sources push flow to both A and B.
S
100
A
17
C
17
t
Δ = 17
sum=-17 + 17
sum=-17 + 17
S
50
B
10
C
The probe from B finds C already labeled by A's probe and explores the path to t, instead.
S
50
A
10
t
Δ = 10
sum=-10 + 10
B incorrectly augments the flow to C as 20.
S
10
B
20
t
13
t
Δ = 13
sum=-10+10+10-20
sum=-17+17+20-13
A augments flow through D, is blocked at C and returns probe through D
S
83
A
15
D
3
C
sum=7

Figure 7. Distributed Flow Augmentation with Two Faulty Processes, B and C

4.2 Other forms of Assertions

Our prior work provided temporal extensions for CCSP to evaluate Interval Temporal Logic Formulae [14]; each process maintains its own discrete clock and a vector clock, allowing events occurring in different, independent processes to be partially ordered using causality. Temporal extensions allow reasoning about events in time on complete, or partially complete, received event sequences to ensure liveness. Formal security policies can also be executed just as safety and liveness properties are [12] and CCSP has been extended to handle run-time evaluation of security calculus expressions.

The system specification, in terms of both functional (safety, such as the KCL assert of the previous example, and liveness) and security requirements, is expressed formally in order to automate any type of checking about the specifications. In this form, the policy can be refined into code and embedded into the executable computer programs. While each program is executing its current tasks, it also performs an operational evaluation, checking that the specifications hold at run time, even in the presence of failures and intruders.

Since an event history is a sequence of events occurring in a system, it represents a process's observation of all the processes during execution. This history can be utilized to do evaluation of assertions at run-time. The evaluation is

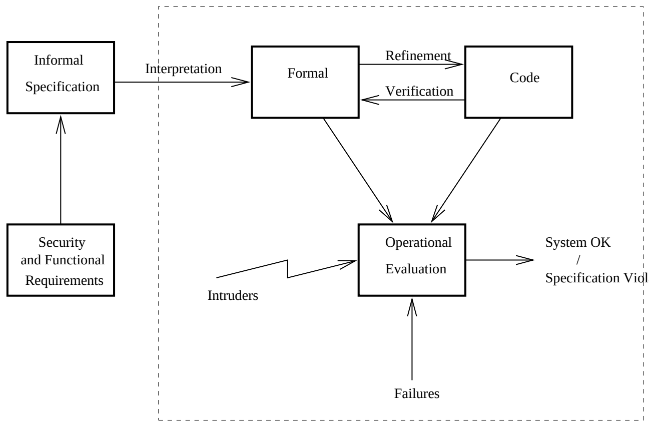

Figure 8. The System's Framework

a simple matter, then, of breaking down the assertions into predicate calculus expressions quantified over this history sequence. If the run-time behavior violates its specification, then appropriate actions should be taken such as halting the system (as is done in the railroad switching industry) or in reconfiguration and recovery, (as is done for systems such as power grid reconfiguration, which cannot be stopped). The complete process that will form the basis of the application in this paper is presented in Figure 8.

More recently, we have explored the JAVA Aglet [8] object model. Aglets form a much more attractive programming model than CCSP and CSP. This model was designed to benefit from the agent characteristics of Java, like platform independence, secure execution, dynamic class loading, multithread programming, object serialization and reflection. In the aglet object model, a mobile agent is a mobile object that has its own thread of control, is event-driven, and communicates via message passing. One problem with CSP is that both the sender and the receiver need to be named. For applications that have dynamic process creation and destruction, such as in mobile computing, this is inadequate. Secondly, our CCSP implementation requires a binding of machine name to process, again, not transparent for distributed systems implementation. While the topology of the power grid application is largely static once the placement of FACTS devices is determined, some dynamic characteristics exist, such as components going offline or coming online.

5 Conclusions

This paper presents a framework for developing fault tolerance approaches for bulk power transmission systems that can be implemented through the hardware and software of distributed FACTS devices. We have laid the foundation for developing fault tolerance assertions based on the well-known max-flow algorithm that allow the run-time identification of system faults, whether intentional or unintentional. Several approaches to conflict resolution have been proposed.

References

[1] R. Ahuja and J. Orlin. A Fast and Simple Algorithm for the Maximum Flow Problem. *Operations Research*, 37(5):748–759, 1989.

[2] E. Arrowsmith and B. McMillin. How to program in CCSP. Technical Report CSC-94-20, University of Missouri - Rolla, Aug. 1994.

[3] J. Bialek. Tracing the flow of electricity. *Proceedings of the IEE Conference on Generation, Transmission, and Distribution*, 143(4):313–320, 1996.

[4] L. Ford and D. Fulkerson. Max Flow Through a Network. *Can. J. Mathematics*, pages 399–404, 1956.

[5] D. Fulkerson and G. Dantzig. Calculations of Maximum Flow in Networks. *Naval Logic Quart.*, 2:277–283, 1955.

[6] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, UK, 1985.

[7] J. Jacob. Basic theorems about security. *Journal of Computer Security*, 1(3–4):385–411, 1992.

[8] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley, 1998.

[9] N. Leveson, H. Heimdahl, and J. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9):684–697, 1994.

[10] J. Marberg and E. Gafni. O(n2m1/2) distributed max-flow algorithm. In *Proceedings of the International Conference on Parallel Processing*, pages 213–216, August 1987.

[11] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[12] C. Serban and B. McMillin. Run-Time Security Evaluation (RTSE) for Distributed Applications. In *Proceedings of the 1996 Symposium on Security and Privacy*, pages 222–232, May 1996.

[13] M. Singhal and N. Shivarati. *Advanced Concepts in Distributed Operating Systems.* McGraw-Hill, New York, 1994.

[14] S. Tsai. *Providing Assurance for Responsive Computing Systems.* PhD thesis, University of Missouri - Rolla, May 1994.

[15] J. Yang and M. Anderson. Tracing the flow of power in transmission networks for use-of-transmission-system charges and congestion management. In *IEEE Engineering Society, Winter Meeting, 1999*, pages 399–405, 1999.

Bruce McMillin received the BS in Electrical and Computer Engineering and the MS in Computer Science from Michigan Technological University, Houghton, Michigan, in 1979 and 1985 respectively and the Ph.D. in Computer Science from Michigan State University, East Lansing, Michigan, in 1988 on Reliable Parallel Processing. Dr. McMillin has worked in both academia and industry. He is currently an Associate Professor of Computer Science and research investigator in the Intelligent Systems Center at the University of Missouri at Rolla. During this time period he also spent a year on sabbatical at SUNY Stony Brook. His research interests include fault tolerance, security, parallel algorithms, software engineering, and distributed systems theory.

M. L. Crow received her BSE degree in electrical engineering from the University of Michigan in 1985, and her MS and Ph.D. degrees in electrical engineering from the University of Illinois in 1986 and 1989 respectively. She is presently a professor of Electrical and Computer Engineering at the University of Missouri-Rolla. She is a member of the IEEE Power Engineering Society System Dynamic Performance Committee. Her area of research interests have concentrated on developing computational methods for dynamic security assessment and the application of power electronics in bulk power systems.