Software Health Management
An Introduction

Gabor Karsai

Vanderbilt University/ISIS

Tutorial at PHM 2009

Outline

- Definitions
- Backgrounds
- Approaches
- Summary

Definitions

- Software Health Management:

  A branch of System Health Management that applies health management techniques to the controlling software of a system.
- SHM goes beyond classical fault tolerance

Software Fault Tolerance	Software Health Management
Fault detected → Functionality restored	Anomaly detected → Fault source isolated → Fault mitigated → Fault prognosticated
When the fault happens, SFT reacts	Software and system health is *managed*

Software Health Management

- Goals:
  - To prevent a (software) *fault* from becoming a (system) *failure*
  - Manage 'health' of the software
    - Sense, analyze, and act upon health indicators
  - Provide (relevant) information to operator, maintainer, designer
- Assumption:
  - Software 'health' is a measurable, non-binary property

Software Health Management

- Characteristics
  - Performed at run-time, on the running system
  - Includes all phases of health management:
    - *Detection*: detect anomalous behavior
    - *Isolation*: isolate source of fault (component, failure mode)
    - *Mitigation*: take action to reduce/eliminate impact of fault
    - *Prognostics*: predict impending faults and failures
  - Can be highly mode- and mission/goal-dependent

Backgrounds:
Basics of Software Fault Tolerance

- Definition:
  - Software Fault Tolerance: Methods and techniques to implement software that can tolerate faults in itself, in the platform it is running on, in the hardware system it is connected to, in the environment

Backgrounds:
Basics of Software Fault Tolerance

- Why? → Serves as a foundation for SHM
  - See Fault-Tolerance vs. System Health Management
- What? → Follows the (HW) Fault Tolerance principles in SW
- Literature:
  - Wilfredo Torres-Pomales: Software Fault Tolerance: A Tutorial, NASA/TM-2000-210616, Langley Research Center, 2000 ← ***CREDIT***
  - Software Fault Tolerance, Edited by Michael R. Lyu, Published by John Wiley & Sons Ltd.
  - Google: “Software Fault Tolerance”

Basics of Software Fault Tolerance
Single version

- Definition: FT for **a** software component (module, application, service,…) – one version of the component (code) is used
- Architectural issues
  - Foundation for SFT: the architecture
  - Component-oriented architecture
    - Modularization – horizontal partitioning
    - Layering – vertical partitioning
    - Common thread: prevent propagation of failures (H + V)

Basics of Software Fault Tolerance Single version

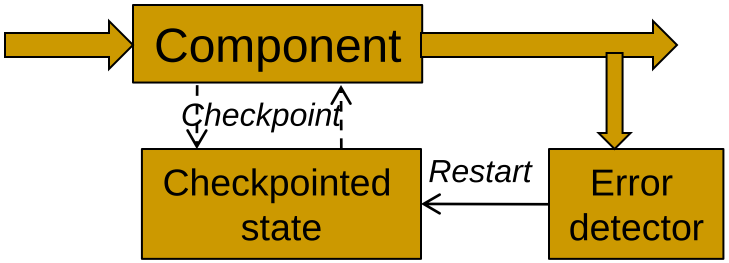
- Detection
  - Requires:
    - *Self protection*: component protects itself from outside effects
    - *Self checking*: component detects its own faults and prevents their propagation
  - Concepts / Techniques:
    - Replication checks: components replicated and results compared
    - Timing checks: deadlines, response times, …
    - Reversal checks: ‘inverse’ function: output → input
    - Coding checks: use redundancy in representations, e.g. CRC
    - Reasonableness checks: value/range/rate/sequence of data
    - Structural checks : verify data structure integrity

Basics of Software Fault Tolerance Single version

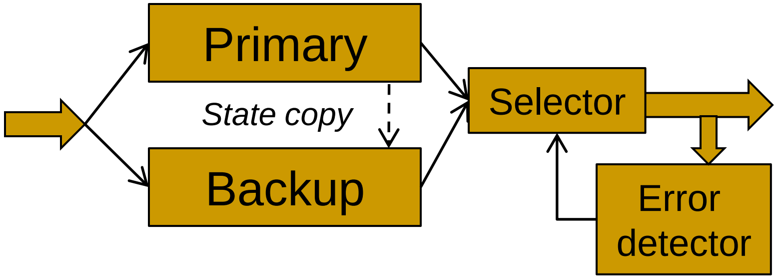
- Exceptions and their management
  - Language-based mechanisms
    - C++, Java, Ada, …not in C!
    - Hierarchical nesting (per control flow)
    - Incorrect requirements/design can lead to major problems (Ariane 5)
  - Categories:
    - Interface exceptions: self-protecting component raises it
    - Local exceptions: generated and contained w/in component
    - Failure exceptions: local management failed, global actions is needed

Basics of Software Fault Tolerance
Single version

- Checkpoints and restarts
  - Detect and restart
  - Categories:
    - Static : reset to an 'initial' state
    - Dynamic : checkpoint state, restore previous one upon failure
  - Problems: non-invertible actions



- Process pairs
  - Identical versions
  - Separate processors
  - State checkpointed
  - On fault, backup takes over

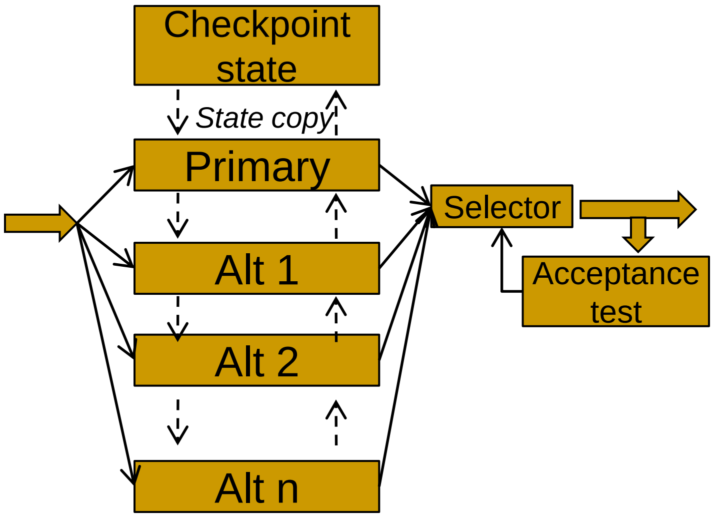


Basics of Software Fault Tolerance Multi version

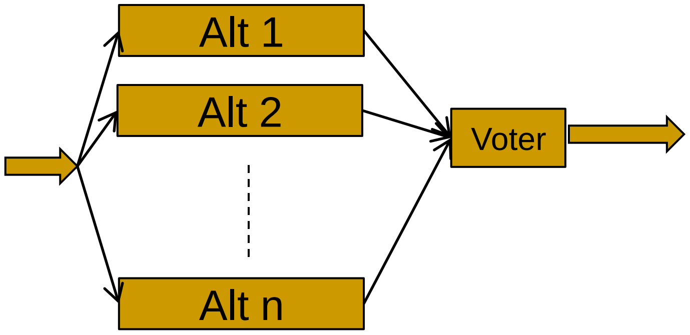
- Definition: FT for software system – multiple versions of component/s (code) are used
- Multiple versions:
  - Same spec
  - Diversity: in design, implementation, language, compiler, processor, etc. + independent teams
- Issues
  - Specification errors (e.g. omissions) could be a common source of faults
  - Experimental result: faults are not really independently distributed over the input space – underlying similarities in design/implementation/etc. and faults…?

Basics of Software Fault Tolerance Multi version

- Recovery blocks
  - Create checkpoint before start
  - If version fails, try another one (use checkpointed state)
  - Alternatives can provide 'graceful degradation'



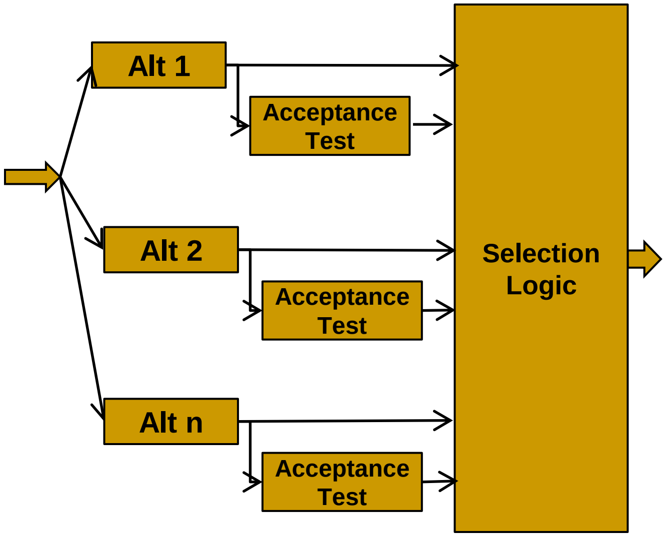
- N-version programming
  - Independent alternatives
  - Generic 'voter' selects



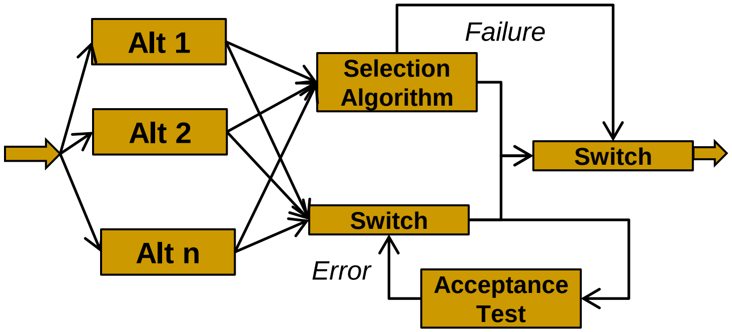
Basics of Software Fault Tolerance

Multi version

- N self-checking
  - Each alternative is self-checking
  - Selection logic selects ‘best’



- Consensus-based
  - If the selection algorithm fails to find a correct output then an output is chosen that has passed the acceptance test



Basics of Software Fault Tolerance
Multi version

- Output selection issues
  - Acceptance tests are hard to build
  - Voters may have to work with inexact comparisons
- Two-step process:
  - *Filtering* via acceptance tests
  - *Arbitration* step to choose output
- Generalized voters:
  - Majority, median, plurality, weighted averaging,…
- Choice must be based on system level issues
  - Reliability, safety, availability, etc.

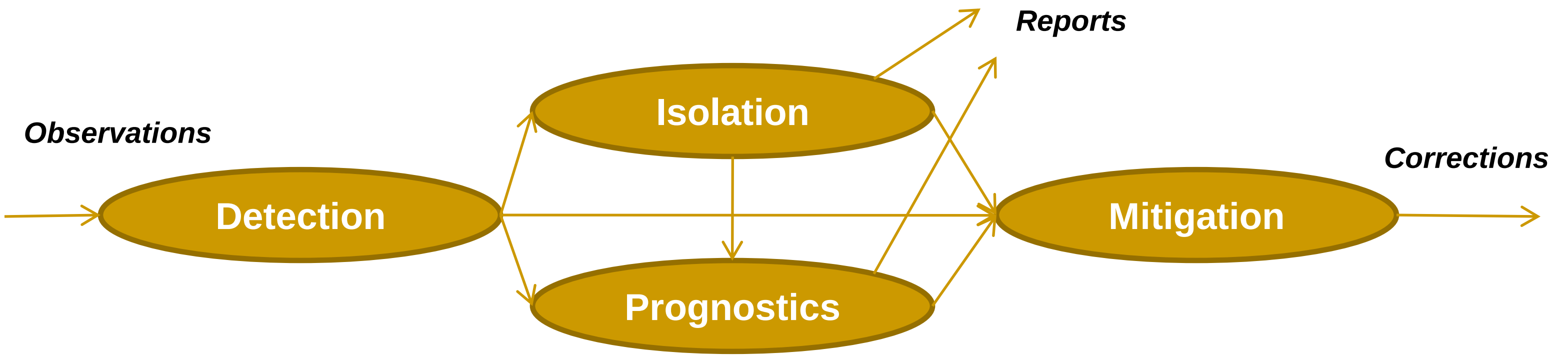
Software Fault Tolerance vs. Software Health Management

- Complexity of systems necessitates an additional layer ‘above’ SFT that manages the ‘Software Health’
- Why?
  - Software is a crucial *ingredient* in aerospace systems
  - Software as a method for implementing *functionality*
  - Software as the ‘universal system *integrator*’
  - Software could exhibit faults that lead to *system* failures
  - Software complexity has progressed to the point that zero-defect systems (containing both hardware and software) are very difficult to build
- Systems Health Management is an emerging field that addresses precisely this problem: How to manage systems’ health in case of faults ?

Software Health Management and System Health Management

- What is System Health Management?

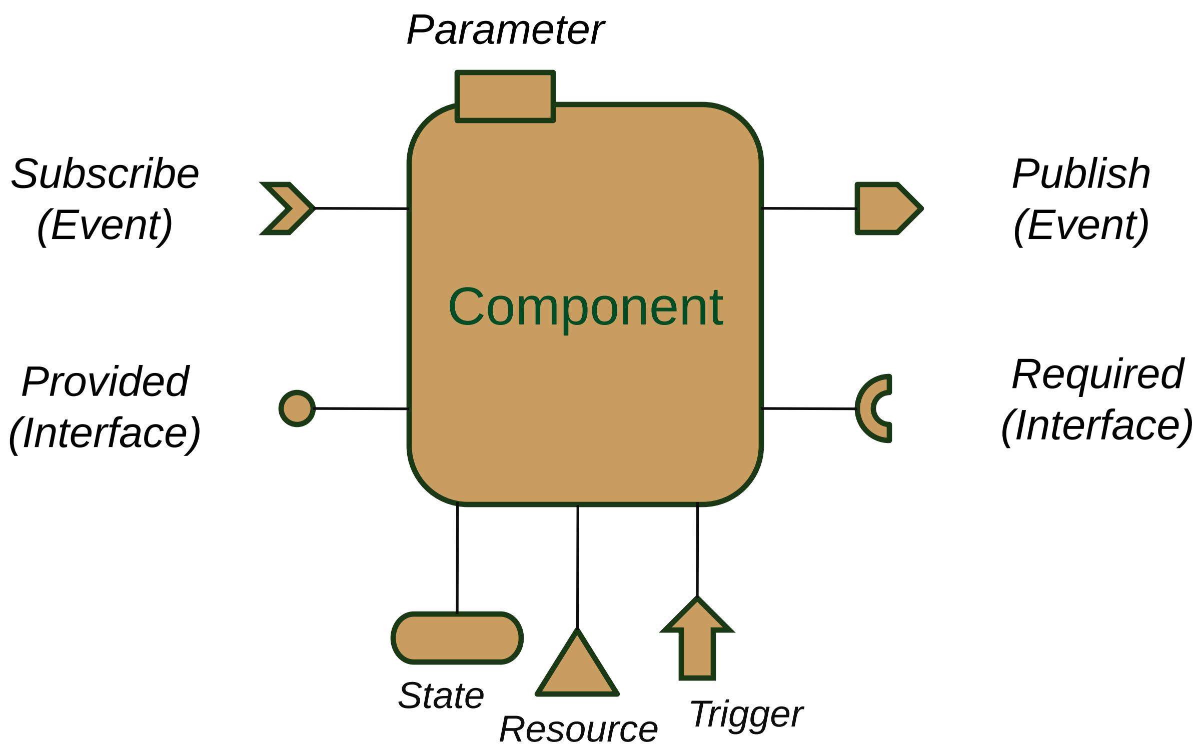
  → The 'on-line' view:
  - **Detection** of anomalies in system or component behavior
  - **Identification** and **isolation** of the fault source/s
  - **Prognostication** of impending faults that could lead to system failures
  - **Mitigation** of current or impending fault effects while preserving mission objective/s



Design for Software Health Management

- Component-oriented software architecture
  - Systems are built by composing components via well-defined interfaces and composition principles
  - There is a (highly robust and reliable) component framework that mitigates all component interactions
    - Component framework is built to higher integrity/quality standards than 'application' software (e.g. RTOS vs. app)
  - Beyond classical architecture-based SFT:
    - No 'single fault' assumption – multiple faults are possible
    - Cascading fault effects are also possible
    - Software Health Management is a system-level function – it must be integrated with System Health Management

Example: Component Model



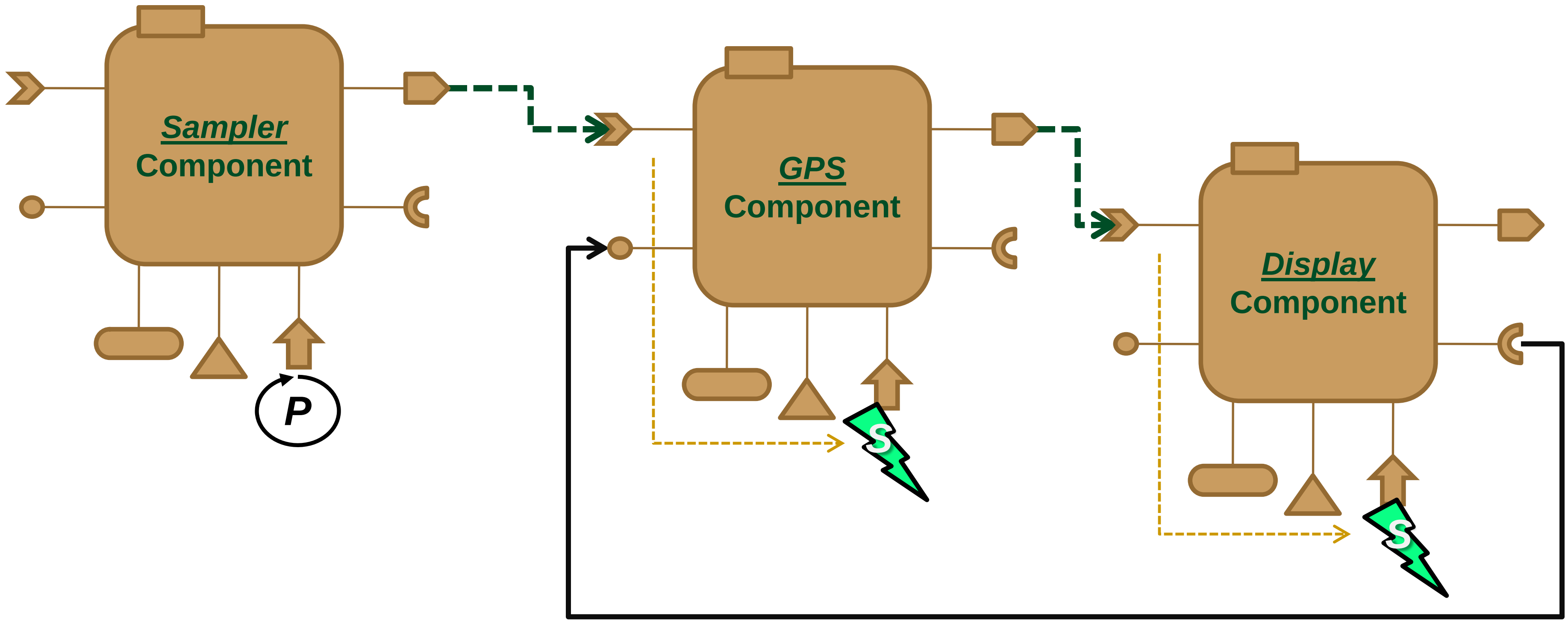
A component is a unit (containing potentially many objects). The component is parameterized, has state, it consumes resources, publishes and subscribes to events, provides interfaces and requires interfaces from other components.

Publish/Subscribe: Event-driven, asynchronous communication

Required/Provided: Synchronous communication using call/return semantics.

Triggering can be periodic or sporadic.

Example: Component Interactions



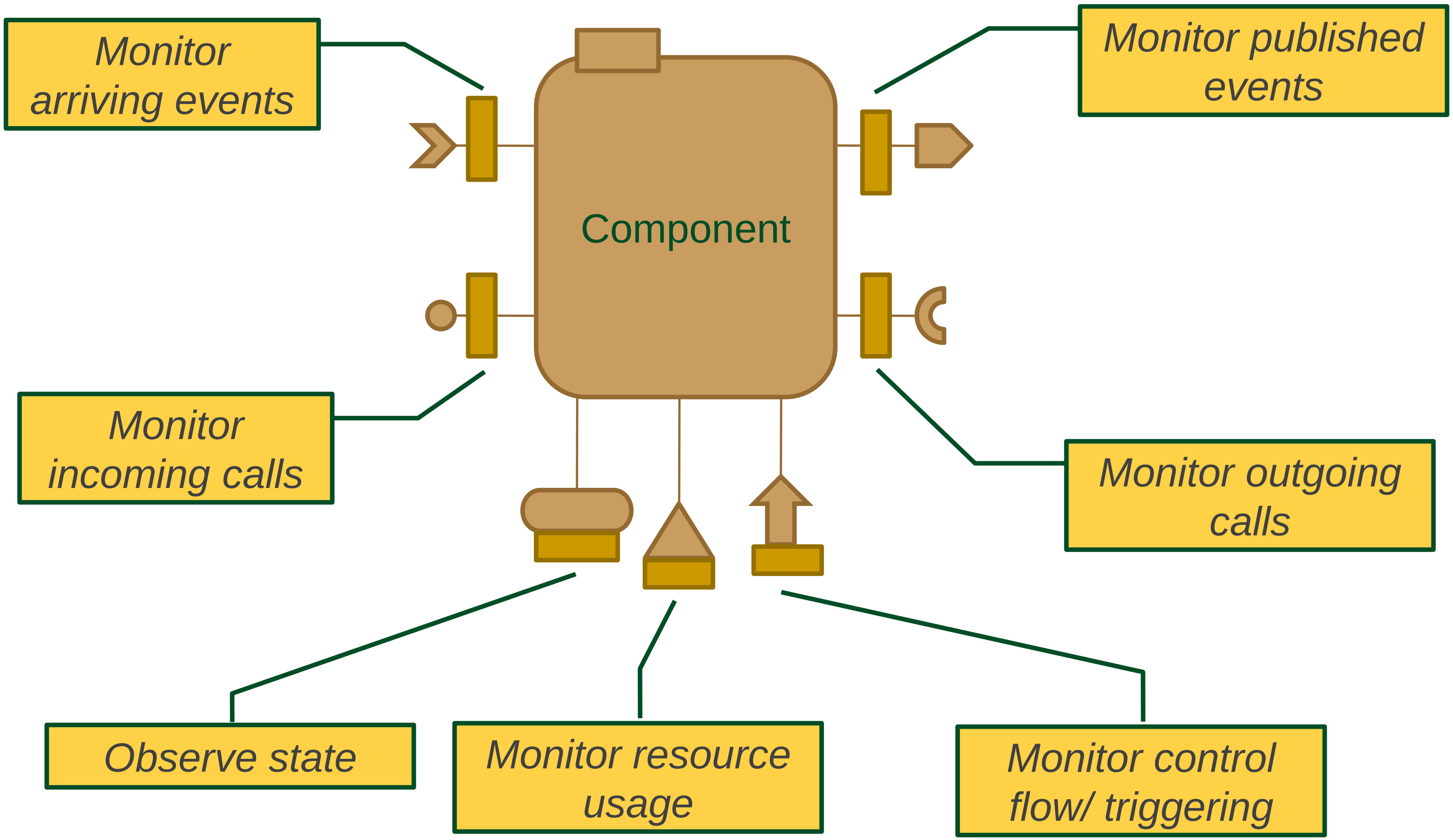
Components can interact via asynchronous/event-triggered and synchronous/call-driven connections.

Example: The *Sampler* component is triggered periodically and it <u>publishes</u> an event upon each activation. The *GPS* component <u>subscribes</u> to this event and is triggered sporadically to obtain GPS data from the receiver, and when ready it publishes its own output event. The *Display* component is triggered sporadically via this event and it uses a <u>required</u> interface to retrieve the position data from the *GPS* component.

Design for Software Health Management

- Component-level health management
  - Very localized → limited capability, yet needed for higher levels
  - Monitor component – detect anomalies
    - What to monitor
      - Input and output: pre- and post-conditions on incoming and outgoing synchronous calls and asynchronous events
      - State: invariants over the component state
      - Timing: component operation execution time
        - Execution (response) time
        - Frequency of invocation
      - Resource usage: component resource consumption patterns
        - Memory, resource lock/unlock, etc.
    - How to monitor
      - Momentary values
      - Rates
      - History/trends

Component Monitoring



Component-level Health Management

A *Component Level Health Manager* reacts to detected events and takes mitigation actions. It also reports events to higher-level manager/s.

Events: detected by monitoring

Actions:

Basic mitigation: reset, init, shutdown, destroy, checkpoint/restore

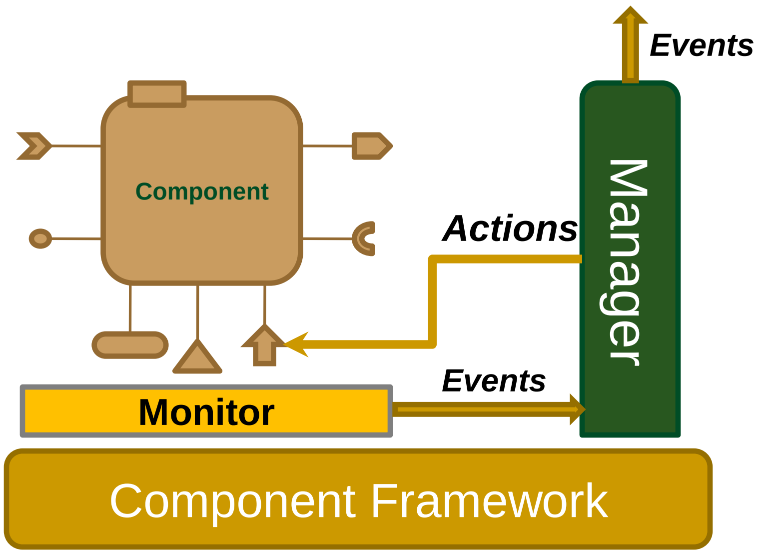
Intercept related: allow/block call

Specialized mitigation: inject event, call method, deallocate memory, release resource, …

Event or time-triggered activation

Reporting

Report events/actions to other managers



Manager's behavioral model:

- Finite-state machine
- Triggers: monitored events, time
- Actions: mitigation activities

Manager is local to component container (for efficiency) but ***must*** *be protected from the faults of functional components.*

Component-level Health Management

<u>Manager behavior:</u>

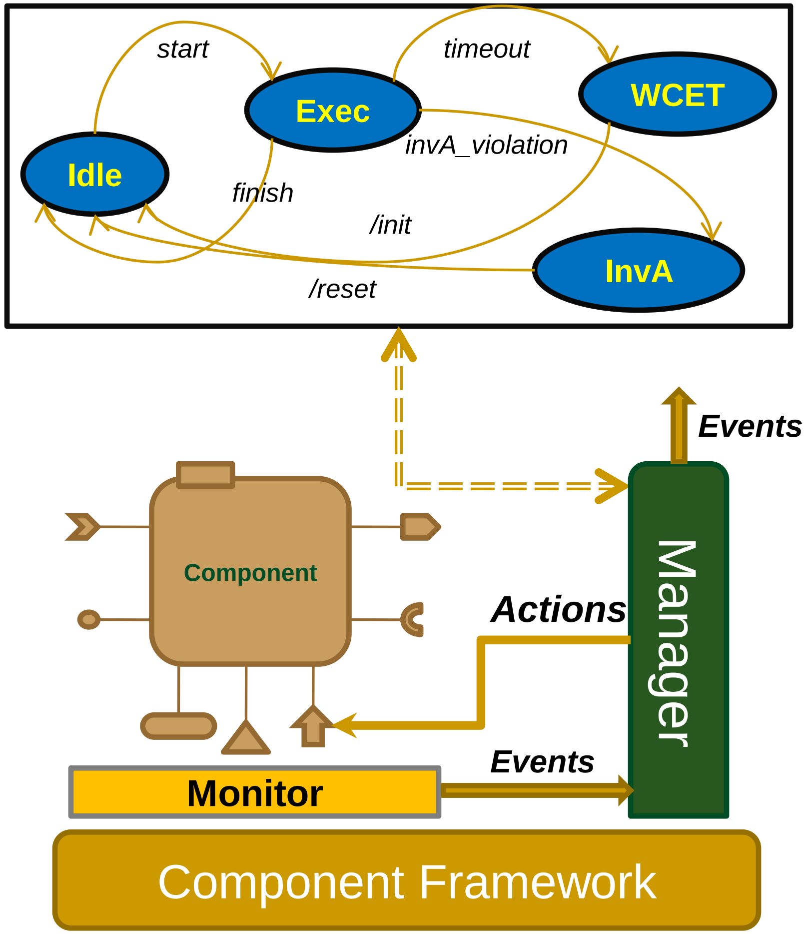
Track component state changes via detected events and progression of time

Take mitigation actions as needed

<u>Design issues:</u>

- Co-location with component
  - Fault containment
  - Efficiency
- Local detection may implicate another component
- Mitigation action may include blocking the call, overriding data…
- Complexity of mitigation actions
- Verification of mitigation logic
  - Safety conditions
  - Performance issues

Manager encapsulates <u>all</u> HM Logic

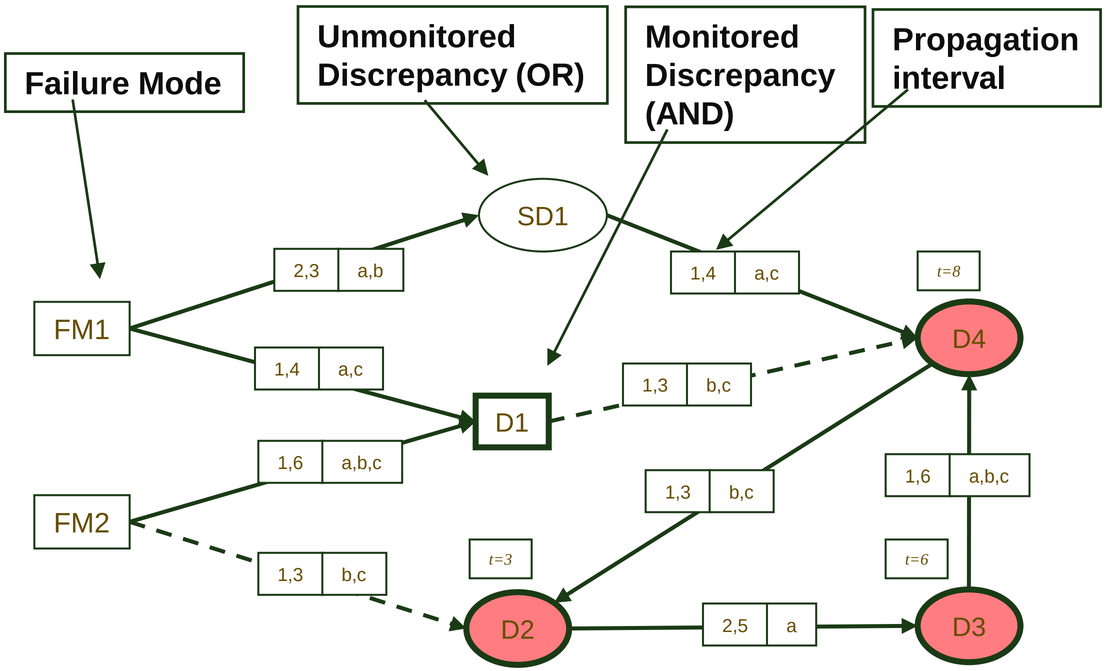


Design for Software Health Management

- System-level health management
  - Multiple components can fail, independently
  - Fault effects cascade through components
  - Anomalies (with cascading effects) and faults propagating through components and assemblies must be correlated and managed
- *Diagnosis*: Isolate the fault source component
- *Mitigation*: Take (component-)local or global action to mitigate effect of fault/s

Design for Software Health Management

- A system-level fault model: Timed Failure Propagation Graph



- Current Time = 10
- Op. Modes = {a,b,c}, Current Mode = b
- Alarm sequence: {(3,D2), (6,D3), (8,D4)}

- **F: set of failure modes**
- **D: set of discrepancies**
- **Discrepancy attributes:**
  - **Type: {AND, OR}**
  - **Condition: {Monitored, unmonitored}**
- **M: Set of operating modes**
- **E: set of edges**
- **Edge attributes:**
  - **Propagation interval: [t_{min},t_{max}]**
  - **Activation modes**

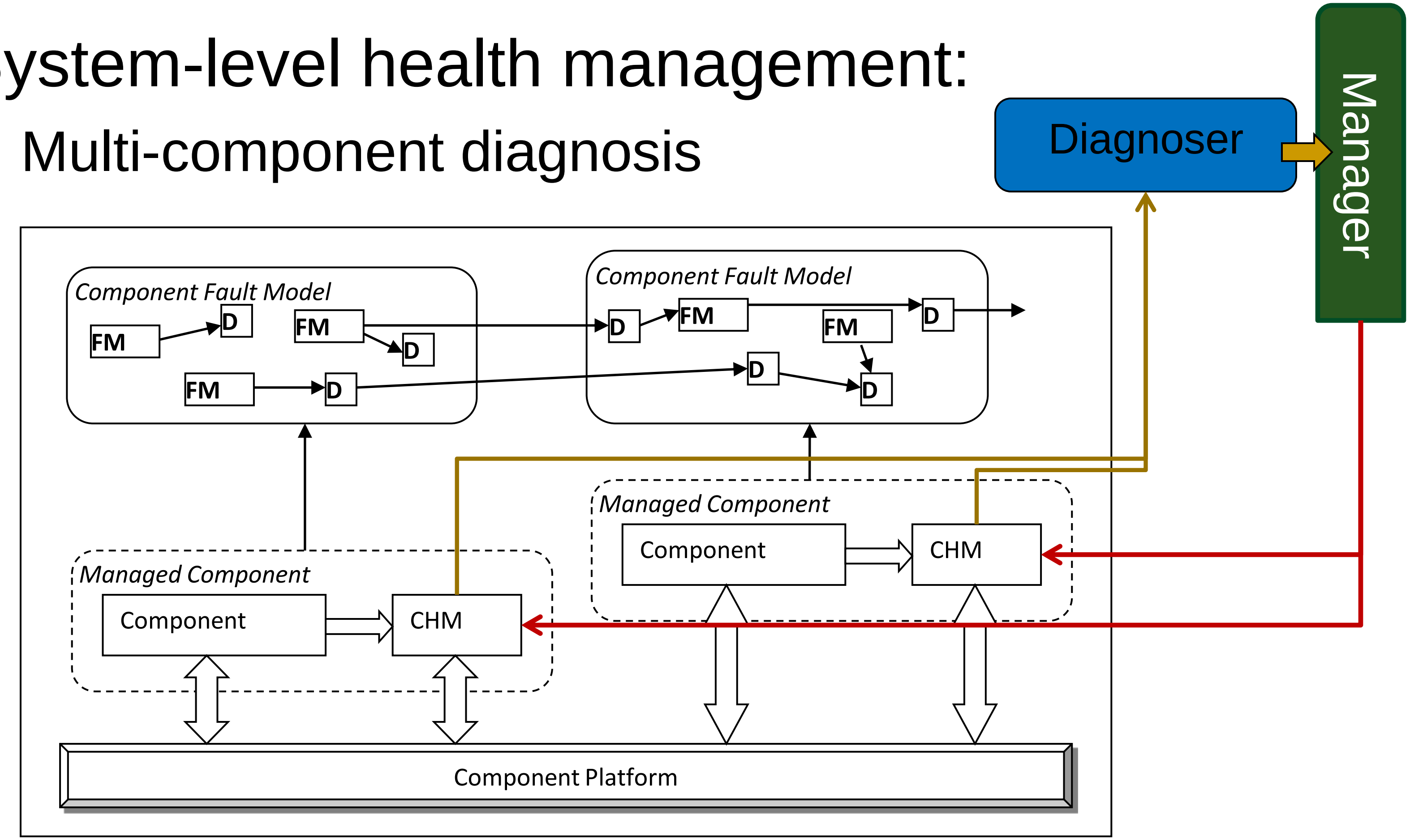
Abdelwahed, S., G. Karsai, and G. Biswas, "A Consistency-based Robust Diagnosis Approach for Temporal Causal Systems", 16th International Workshop on Principles of Diagnosis (DX '05), Monterey, CA, June, 2005.

Design for Software Health Management

- System-level health management
  - Model:
    - Faults (failure modes) and discrepancies (observed anomalies) can be located in different components
    - Fault propagation occurs along component communication links / call chains
  - Diagnosis:
    - Correlate observations across multiple components, deduce fault source
    - Features: modal, robust, ranked results, multiple faults

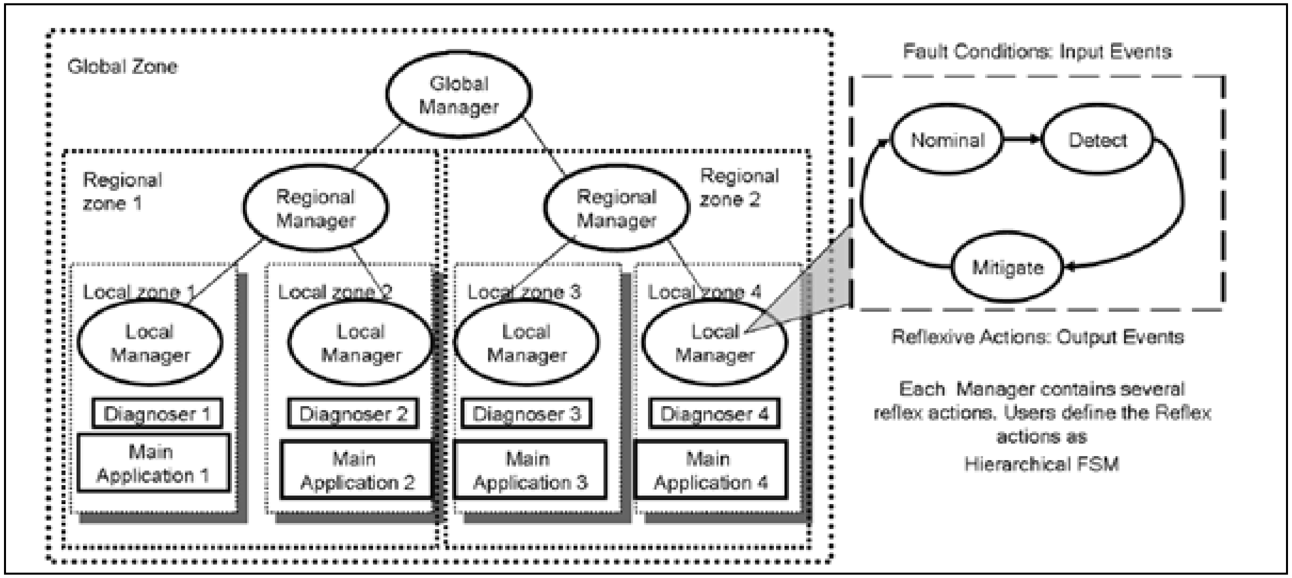
Design for Software Health Management

- System-level health management:
  - Multi-component diagnosis



Design for Software Health Management

- System-level health management:
  - Multi-component, hierarchical mitigation



Local: reflex reactions

Regional: mitigation in an area

Global: system-level mitigation

Dubey, A., S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai, "Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems", ISSE, vol. 3, pp. 33--52, 2007.

Summary

- Software Health Management: A branch of System Health Management that applies HM techniques to the controlling software of a larger system.
- Software Fault Tolerance provides useful techniques for SHM, but SHM reaches beyond SFT as it has a comprehensive approach to anomaly detection, diagnosis, mitigation and prognostics.
- Initial progress in the area of component-level and system-level software health management shows promise, but it is subject of ongoing research.