Real-Time systems
Monitoring

Real-Time Industrial Systems

Marcello Cinque
Raffaele Della Corte

Roadmap

- Monitoring concept and types
- Embedded constraints
- Monitored constraints
- Fault tolerance and timing failure
- Rule-based logging
- Error propagation with event logs

- References:
  - S. Chodrow et al. «Run-Time Monitoring of Real-Time Systems»
  - M. Cinque et. al.
    «A Logging Approach for Effective Dependability Evaluation of Complex Systems»
    «Event logs for the analysis of software failures: a rule-based approach»
    «An empirical analysis of error propagation in critical software systems»
  - A. E. Goodloe and L. Pike «Monitoring Distributed Real-Time Systems: A Survey and Future Directions»
  - A. Burns and A. Wellings «Real-Time Systems and Programming Languages»
    - Chapter 13 “Tolerating timing faults”

Monitoring: motivations

- Many assumptions on both tasks and execution environments are made during the design and development of real-time systems
  - Execution time, deadline, communication delay, system overhead, …
- These assumptions allow addressing problems that would be difficult to handle
- However, they might be violated at run-time due to unexpected external or internal factors
  - e.g., hardware failures, software bugs, communication errors with devices or network, overload, …

Monitoring: motivations

- Monitoring is required to verify, mainly at execution time, if the system behavior is compliant with its specification
  - Detecting potential anomalous situations violating the initial assumptions and taking the related countermeasure
- In the hard real-time systems this is translated in verifying temporal constraints, while introducing a negligible overhead

Monitoring and *fault-tolerant* systems

- Monitoring is a relevant practice in *fault-tolerant* systems

- A fault-tolerant system represent a system that is able to provide its services even when failures occur
  - A **failure** takes place when the service offered by the system deviates from the expected behavior (according its specification)
  - The **error** is the part of the system status that can lead the system to a failure
  - The **fault** is the cause, hypothetical or confirmed, of the error

Il Fault accade generando un errore che propagandosi nel sistema causa un failure

Monitoring and *fault-tolerant* systems

- Different techniques exist to develop fault tolerant systems
  - Error correction/masking: techniques allowing the detection and correction of potential errors
  - Redundancy: the system components are replicated
  - Checkpoint & Recovery: restore the system to a previous working status
  - …
- It is necessary to monitor the system status in order to detect potential errors and activate the mitigation action before the system failure
  - Error correction, start of a replica/recovery, …

Monitoring architecture requirements

- A **monitor architecture** is the integration of one or more **monitors** with the *System Under Observation* (SUO)
- *Monitor*: entity observing the system behavior to detect if it is compliant with its specification
- Different architectural constraints must be met in fault-tolerant real-time systems to avoid that monitors affect the *nominal functionalities* of the SUO:
  - *Functionality*: the monitor does not change the functionality of the SUO unless the SUO violates its specification
  - *Schedulability*: the monitor architecture does not cause the SUO to violate its hard real-time guarantees, unless the SUO violates its specification
  - *Reliability*: the reliability of the SUO in the context of the monitor architecture is greater or equal to the reliability of the SUO alone
  - *Certifiability*: the monitor architecture does not unduly require modifications to the source code or object code of the SUO

se il monitoring aggiunge e modifica molto codice
dobbiamo rieseguire tutti i test e certificazioni del SUO

Examples of requirements violations

- *Functionality*
  - The monitor alters the functionality of the SUO also when no violations are detected
- *Schedulability*
  - The monitor leverages source code instrumentation (i.e., it requires the execution of extra code), which affect the timeliness of services
  - Distributed monitors communicate over the same channels used by the SUO, decreasing the bandwidth available for the SUO
- *Reliability*
  - Faults of the monitor propagate to a failure in the SUO, reducing its reliability (i.e., no actions are taken by the monitor when the failure occurs)
- *Certifiability*
  - A monitor framework requires rewriting or modifying the source code or object code of the SUO, then requiring the full re-evaluation of the SUO

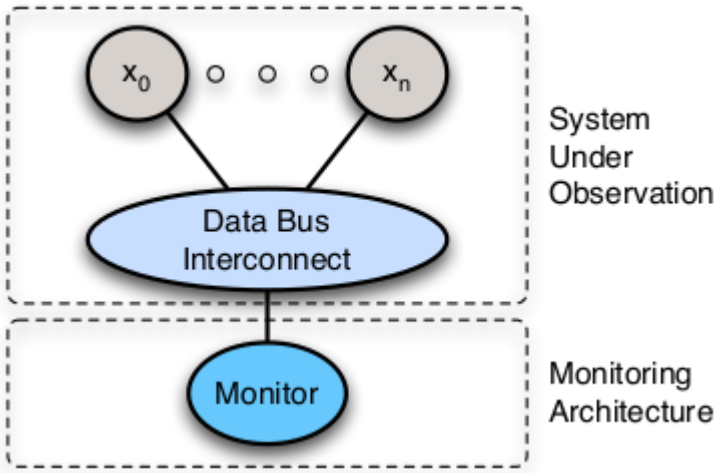
Monitor Architecture

Considering a distributed SUO, three different monitoring architectures can be identified:

- *Bus-Monitor Architecture*
  - The monitor observes traffic on the data bus of the SUO
- *Single Process-Monitor Architecture*
  - The monitor observes a dedicated monitoring bus
  - Each monitored process is instrumented to write monitoring data on the dedicated bus
- *Distributed Process-Monitor Architecture*
  - Distributed monitors one for each monitored process

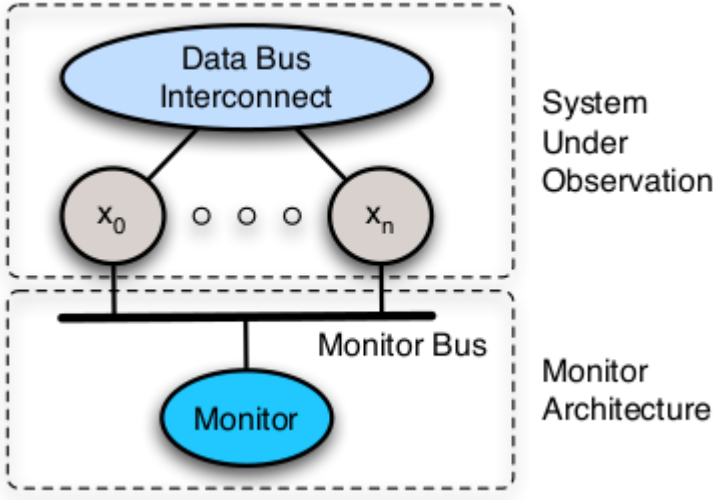
Bus-Monitor Architecture

- The monitor receives messages over the bus as any other process of the SUO
- The monitor writes messages to other processes on the bus only when it detects a catastrophic fault
- *Pros*
  - Requires the least additional hardware
  - Simplest monitoring architecture
- *Cons*
  - The monitor may consume the bus bandwidth in case the monitor becomes faulty
  - The monitor can detect a limited class of faults, since it infers the health of a process from the messages exchanged over the bus



Single Process-Monitor Architecture

- The monitor gathers and analyzes the data sent on the monitor bus by the instrumented processes
- The monitor may signal the processes if a fault is detected
- *Pros*
  - No shared bus between SUO processes and monitor
- *Cons*
  - Source code instrumentation in order to allow SUO processes to send monitoring data on the dedicated bus
    - It may impact on the functionality and certifiability requirements
  - Potential impact on the reliability requirement depending on the implementation

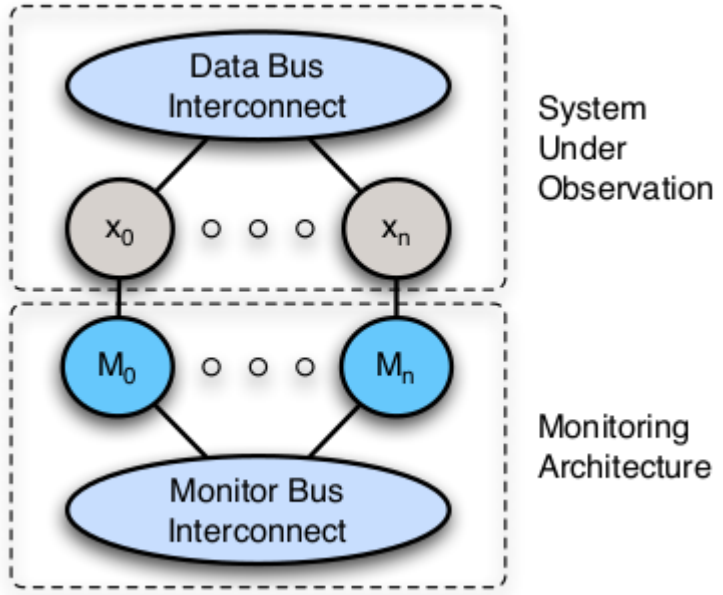


In pratica devo aggiungere del codice ai processi per inviare informazioni sul Monitor Bus quindi devo modifico il SUO e andranno rifatti i test di Reilability e Certificability

w.r.t = with respect to

Distributed Process-Monitor Architecture

- The distributed monitors communicate with each other in order to reach agreement on diagnoses
- *Pros*
  - No shared bus between SUO processes and monitor
  - More reliable w.r.t. the Single process-monitor architecture since the monitors are distributed
    - The failure of multiple monitors can be tolerated
  - A monitor for each process
- *Cons*
  - Complexity, which may impact the reliability of the monitor
  - Expensive in terms of processes and interconnections
  - Difficult to implement in case of stringent cost, size, weight or energy consumption constraints on the monitor



Monitor types

Hardware ↔ Software

On-line ↔ Off-line

Direct ↔ Indirect

Implicit ↔ Explicit

Centralized ↔ Distributed

Monitor

- Direct Monitor: the event to detect is generated directly by the monitored software (e.g., the end of the cycle)
- Indirect Monitor: the event to detect is obtained from the collection of external data/measurements extern w.r.t. the monitored software (e.g., CPU and Network usage, …)
- Explicit Monitor: the execution is verified against a (formal) specification of the correct system behavior
  - e.g., real time logic (RTL)
- Implicit Monitor: the execution is verified against a model representing the incorrect system behavior
  - e.g., failure modes, error model, …

Real Time Logic

- The execution of a real-time system is conceived as a sequence of **events occurrences**
  - **Event**: processing action performed by the system (e.g., start of a calculation, send of a message, …)
  - **Occurrence**: time instant when a particular instance of an event happens

- It is possible to distinguish:
  - **Label events**: represent the start and end of a sequence of program statements; indicated with the symbols ↑ and ↓
  - **Transition events**: represent the assignment of a value to an observable variable (indicate a potential status change)

- An **event history** stores the time (and values, for transition events) of the occurrences happened in the systems

Real Time Logic: some notations

- Used to specify the constraints the processing should respect to be considered correct, e.g.:
  - *@(e,i)*: provides the time of the i-th occurrence of the event *e*
  - *@val(v,i)*: provides the value of the i-th occurrence of a transition event on the variable v
- If the index *i* is positive, indicates the i-th absolute occurrence of the event w.r.t. the start of the processing
  - e.g., *@(e,10)* indicates the 10th occurrence of the event *e*
- If the index *i* is negative, indicates the i-th most recent occurrence of the event in the processing
  - e.g., *@(e,-1)* indicates the time of the most recent occurrence of *e*

Methods for specify and monitor temporal proprieties

- *Embedded constraints*
  - The developer can actively verify if a temporal property is satisfied in dedicated points of the program

- *Monitored constraints*
  - An external task is used to verify if a temporal properties of the monitored task is satisfied

Embedded constraints

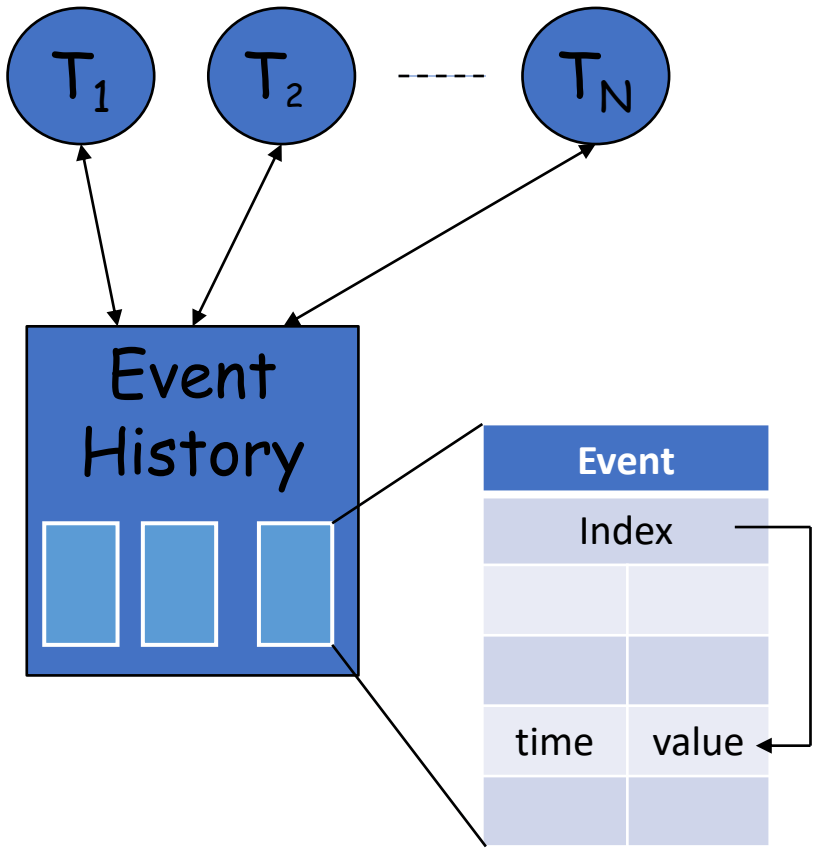
- Example: (*temp*: observable variable)

```
temp = read_sensor( ) ;
if(@val(temp,-1)- @val(temp, -2) > 200) {
     shutdown-reactor();
else {
     raise_rods();
}
```

verifico che la differenza tra il precedente valore temp e quello ancora prima sia maggiore di 200

- Allow the developer to directly access to the *event history* and to use it to verify (in dedicated points) if the processing is correct

Embedded constraints



- Tasks write the events in the history and take them when a given constraint has to be verified
- The insert of a new occurrence in the history requires:
  - obtain the lock on the event history event history
  - get the current timestamp
  - Insert the timestamp (and the value, if present) of the event
  - release the lock

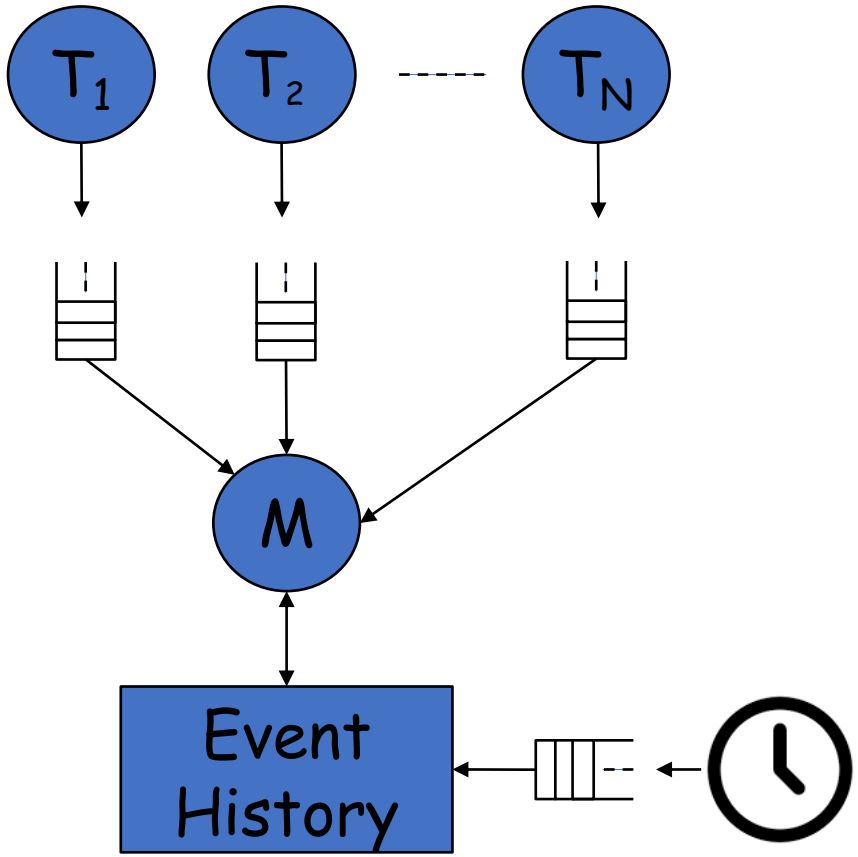
Monitored constraints

- Introduce some benefits:
  - Separate the temporal problems from the functional specification of the program
  - Allow to define constraints that cannot be verified in dedicated execution points
- The constraints are verified during the whole processing
- Example:

$$@(send, i) \leq @(ack, i) \wedge @(ack, i) \leq @(send, i) + 5$$

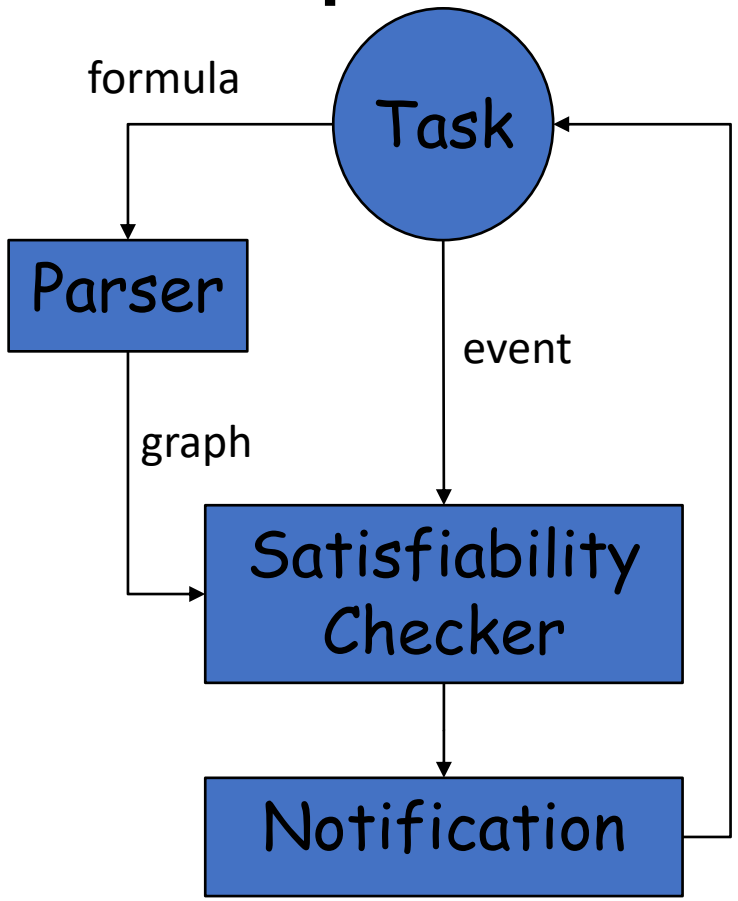
  - the formula specifies a deadline constraint on the ack: for each message sent an ack message should be received within 5 ms

Monitored constraints



- The tasks write events on their own queue
- The queues are read by a Monitor task (M) that holds the history
- When an event occurs, the constraints are verified
  - The tasks are notified about a potential violation, or terminated
- A timer is programmed for the events that require a verification in the future

Monitored constraints: possible implementation



- The tasks specify the RTL formulas that must be verified
- A lexical/syntactical (LEX/YACC) parser generates a graphs of the constraints
- The graph is used by the SAT checker when an event occurs to verify if the formulas are satisfied

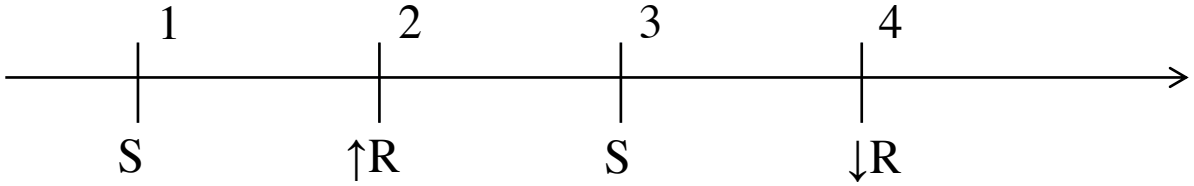
Monitored constraints: example

- Consider a system that must execute a *RESPONSE* action when the *SIGNAL* event occurs
- The formula:

$$@(\uparrow RESPONSE, -1) \leq @(SIGNAL, -1) \rightarrow$$
$$@(\uparrow RESPONSE, -1) \leq @(\downarrow RESPONSE, -1) \wedge$$
$$@(\downarrow RESPONSE, -1) < @(SIGNAL, -1)$$

  Indicates that if the last occurrence of *RESPONSE* starts before the last occurrence of a *SIGNAL* event than the action must be completed before the *SIGNAL*

- The sequence of events in the figure violates the constraint in instants 3 and 4



R non termina prima del successivo Signal quindi Viola

Fault tolerance and timing faults

Dynamic software tolerance phases in the context of timing failures

- **Error detection**: Most timing faults will eventually manifest themselves in the form of missed deadlines
- **Damage confinement and assessment**: Confinement techniques should ensure that only faulty tasks miss their deadline
- **Error recovery**: The response to deadline misses requires that the application undertakes some recovery, perhaps providing a degraded service
- **Fault treatment and continued service**: Timing errors often result from transient overloads, but in case of persistent deadline misses may they may indicate more serious problems. In this case, some form of maintenance is required to be undertaken

Why using monitoring after a successful schedulability analysis?

Deadline could be still miss also after the schedulabilty analysis proved that our system is correct in the timing domain

- WCET calculations were inaccurate
- Blocking times were underestimated
- Assumption made in the schedulability checker were not valid
- Error in the schedulabilty checker
- The scheduling algorithm could not cope with a load event though it is theoretically schedulable
- The system is working outside its design parameters, e.g., sporadic events occurring more frequently than was assumed in the schedulability analysis

Examples of fault, error, failure chain

	FAULT	ERROR	ERROR PROPAGATION	FAILURE
1	WCET calculation of task τ_i	Overrun of task τ_i	Deadline miss of task τ_i	Unable to deliver service in a timely manner
2	Minimal inter-arrival time assumptions for task τ_i	Greater computation requirement for task τ_i	Deadline miss of task τ_i	Unable to deliver service in a timely manner
3	WCET calculation of task τ_i or assumptions when using a shared resource	Overrun of the resource usage of task τ_i	Greater blocking time of higher-priority tasks sharing the resource Deadline miss of higher-priority tasks	Unable to deliver service in a timely manner

How to be tolerant to timing faults?

- It is necessary to be able to detect:
  - Miss of a deadline
  - Overrun of a Worst-Case Execution Time
  - Overrun in the usage of a resource
  - A sporadic event occurring more often than predicted

- There is the need to provide applications with mechanisms to detect errors generated by timing faults
  - Considering the fault -> error -> failure chain, **the detection is performed at error level**

How to be tolerant to timing faults?

- It is necessary to be able to detect:
  - Miss of a deadline
  - Overrun of a Worst-Case Execution Time
  - Overrun in the usage of a resource
  - A sporadic event occurring more often than predicted

- There is the need to provide applications with mechanisms to detect errors generated by timing faults
  - Considering the fault -> error -> failure chain, **the detection is performed at error level**

Deadline miss detection (RT-POSIX)

- Leverage timers provided by the RT-POSIX API, which generate user-defined signal when they expire
- Example of solution:
  - Components
    - A **server** thread representing the SUO
    - A **monitor** thread, which aims to monitor the progress of the *server* and to verify if it meets its deadline
  - Interaction
    - The *monitor* creates
      - The *server*, whose deadline is specified by a `struct timespec deadline`
      - A per process *timer* indicating a signal handler to be executed if the timer expires
    - The *server* executes its actions and then deletes the timer
    - If the alarm goes off, the server is late

in pratica il server mantine il timer attivo prolungando il tempo in cui deve scadere,

Deadline miss detection (RT-POSIX)

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t timer; /* timer shared between monitor and server */

struct timespec deadline = ...;
struct timespec zero = ...;

struct itimerspec alarm_time, old_alarm;

struct sigevent s;

void server(timer_t *watchdog) {
  /* perform required service */
  TIMER_DELETE(*watchdog);
}

void watchdog_handler(int signum, siginfo_t *data,
                      void *extra)
{
  /* SIGALRM handler */

  /* server is late: undertake recovery */
}
```

```
void monitor() {
  pthread_attr_t attributes;
  pthread_t serve;


  sigset_t mask, omask;
  struct sigaction sa, osa;
  int local_mode;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, SIGALRM);

  sa.sa_flags = SA_SIGINFO;
  sa.sa_mask = mask;
  sa.sa_sigaction = &watchdog_handler;

  SIGACTION(SIGALRM, &sa, &osa); /* assign handler */


  alarm_time.it_value = deadline;
  alarm_time.it_interval = zero; /* one shot timer */

  s.sigev_notify = SIGEV_SIGNAL;
  s.sigev_signo = SIGALRM;

  TIMER_CREATE(CLOCK_REALTIME, &s, &timer);

  TIMER_SETTIME(timer, TIMER_ABSTIME, &alarm_time, &old_alarm);

  PTHREAD_ATTR_INIT(&attributes);
  PTHREAD_CREATE(&serve, &attributes, (void *)server, &timer);

}
```

WCET Overrun detection (RT-POSIX)

- Tasks non-preemptively scheduled
  - the CPU time is equal to the elapse time; therefore, the same mechanism of deadline miss detection can be used
- Tasks preemptively scheduled
  - the CPU time usage measurements requires support of the host operating system
  - RT-POSIX provides both *clocks* and *timer* facilities to support execution-time monitoring
    - `clock_settime(CLOCK_THREAD_CPUTIME_ID, &some_timespec_value);`
    - `clock_gettime(CLOCK_THREAD_CPUTIME_ID, &some_timespec_value);`
    - `clock_getres(CLOCK_THREAD_CPUTIME_ID, &some_timespec_value);`
    - `int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);`
  - In RT-POSIX a timer can be used to generate process-signals when the execution time set has expired, but…

It is difficult to guarantee the accuracy of the execution-time clock in the presence of context switches and interrupts

Overrun of resource usage

- Problem caused by errors in accessing resources are difficult to handle
- They can corrupt shared state, potentially leading to deadlock
- Inheritance and ceiling protocols can be not the solution if the schedulability analysis is incorrect
  - A task may overrun its allotted access time for the resource, or
  - Unanticipated resource contention that has not been considered in the blocking-time analysis

Overrun of resource usage

- The use of timeouts is limited in this context
  - With Inheritance protocol, blocking is cumulative
    - Timeouts can be used on entry of critical sections, but the developer should keep a running track of total blocking time
  - With immediate ceiling protocols, the blocking occurs before execution of the task
    - No contention when accessing the critical section -> timeout has no use
  - Absence of timeout mechanisms for critical section
    - RT-POSIX has a timed `mutex_lock`, but it does not return any information about the blocking time
- A potential solution is to use detection of WCET overruns at block level, but …
  - All resource accesses would have to be policed to ensure that calling task did not overrun its allotted usage
  - Detecting overruns on every resource usage access may be expensive

What about Damage Confinement?

- Damage confinement aims to prevent propagation to other components of the error resulting from time-related faults
  - Protecting the system from the impact of sporadic task overruns and unbounded aperiodic activities
  - Supporting composability and temporal isolations
- Potential solutions
  - **Aperiodic servers**, e.g., Sporadic Server and Deferrable Server
    - They protect the processing resources needed by periodic task but otherwise allow aperiodic and sporadic tasks to run as soon as possible
    - To support aperiodic execution, it is sufficient that the aperiodic server *consumes no more than its budget each period*
  - **Time isolation** with hierarchy schedulers and reservation-based systems
    - Two levels of scheduling are used: a global scheduler and multiple application-level schedulers (*group server*)
    - The group server be *guaranteed its budget each period*, i.e., the tasks contained in the group can consume all the budget of the group on each release

Damage Confinement (RT-POSIX)

- RT-POSIX supports the Sporadic Server approach as one of the scheduling policies, at both process and thread levels
- A Sporadic Server process is a process scheduled according to the sporadic server policy
  - All the threads contained within the process share the allocated budget
  - Timing errors are confined to those threads
- RT-POSIX also supports shared memory objects between process
  - Allow to partition a single application between multiple processes communicating through the shared memory

Explicit Monitor

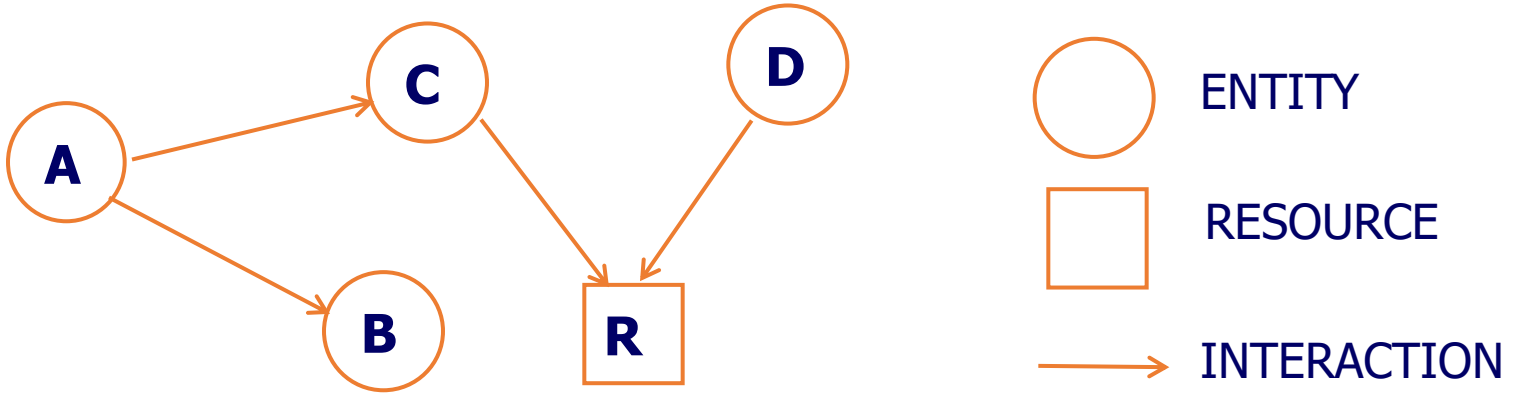
- Main limits:
  - Constraints specification effort
  - Potential incomplete specification
  - Increase of the complexity in distributed systems

Rule-based logging

- Definition of a model of a distributed systems based on entities, resources, and interactions
- Definition of an error model
- Definition of simple rules to follow to «instrument» the source code and generate events of interest, which allow the detection of an error belonging to the model

System model

- *Accurate enough* model and independent from the technological details



- Entity: active elements that execute processing action and interact with other entities or resources

- Resource: passive object the entities interact with

Error model

- **Service Error (SER)**: the service, provided by an entity, does not terminate in a given time range
- **Service complaint (CMP)**: the service, provided by an entity, terminates with an error code
- **Interaction Error (IER)**: the interaction between either two entities or between an entity and a resource does not terminate within a given time range

Logging rules

- Consider the possibility to enable an external monitor to:
  1. Detect the errors of an entity
  2. Discriminate local error from the ones due to the interaction with other entities
- The rules define the **type** of event and **where** it must be logged

R1, *Service Start*: the SST event must be logged before the first instruction of each service provided by an entity

R2, *Service End*: the SEN event must be logged before the last instruction of each service provided by an entity

```
void  A::serviceA(int x){
      log(SST);      //R1
      100/x;         //I1
      b.serviceB();  //I2
      log(SEN);      //R2
}
```

Logging rules

Service Error e Interaction Error

- SST and SEN: do not distinguish SER and IER
  - Example: no SEN for serviceA → is it due to I1 or I2?

```
void  A::serviceA(int x){
      log(SST);      //R1
      100/x;         //I1
      b.serviceB();  //I2
      log(SEN);      //R2
}
```

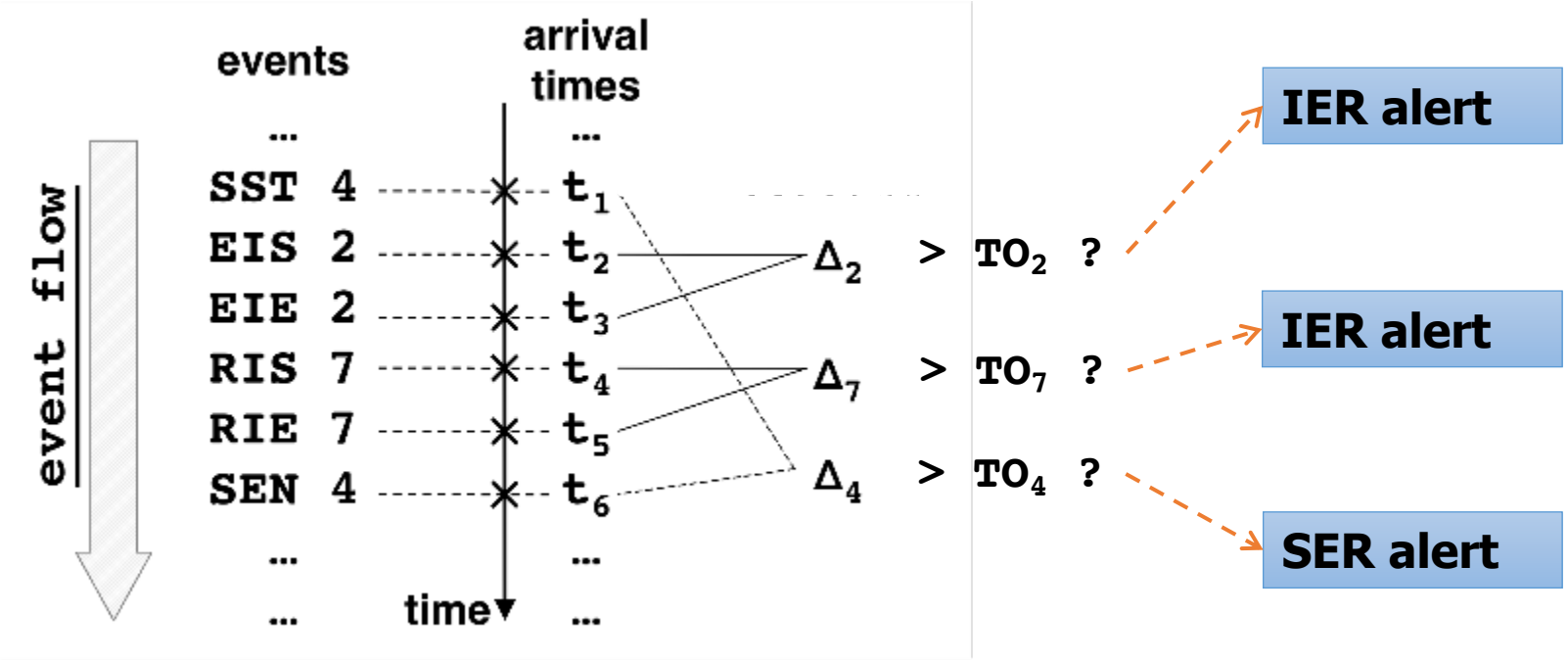
```
void  A::serviceA(int x){
      log(SST);      //R1
      100/x;         //I1
      log(EIS);      //R3
      b.serviceB();  //I2
      log(EIE);      //R4
      log(SEN);      //R2
}
```

R3, *Entity (Resource) Interaction Start*: the EIS (RIS) event is logged before the interaction of a service

R4, *Entity (Resource) Interaction End*: the EIE (RIE) event is logged after the invocation of a service

Events processing and alert

- An **external detector**, based on timeout, can rise alerts related to the error model



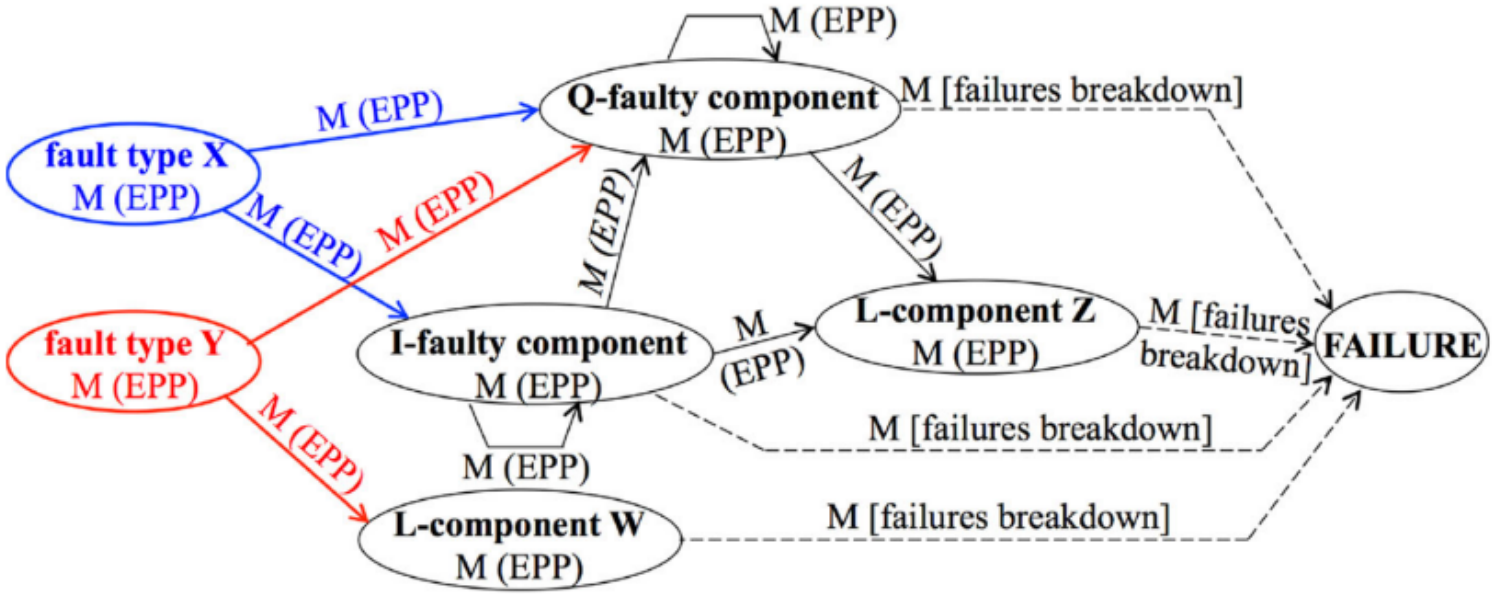
Can we use event logs to understand the error behavior of a system?

- The use of source code instrumentation is not always allowed
  - Unavailability of source code
  - Legacy systems
  - Production environments
  - For certifiability reasons
  - For performance reasons

- A potential solution is to leverage event logs
  - PROS:
    - Naturally-emitted data by software systems in production environments
    - Generated by different software modules, and available at different system levels
    - Textual data encompassing useful information about the runtime behavior of the system
  - CONS:
    - Incompleteness and inaccuracy issues
    - Logs production and management is left to the developers experience and attitude
    - Heterogeneity issue
    - Unstructured nature

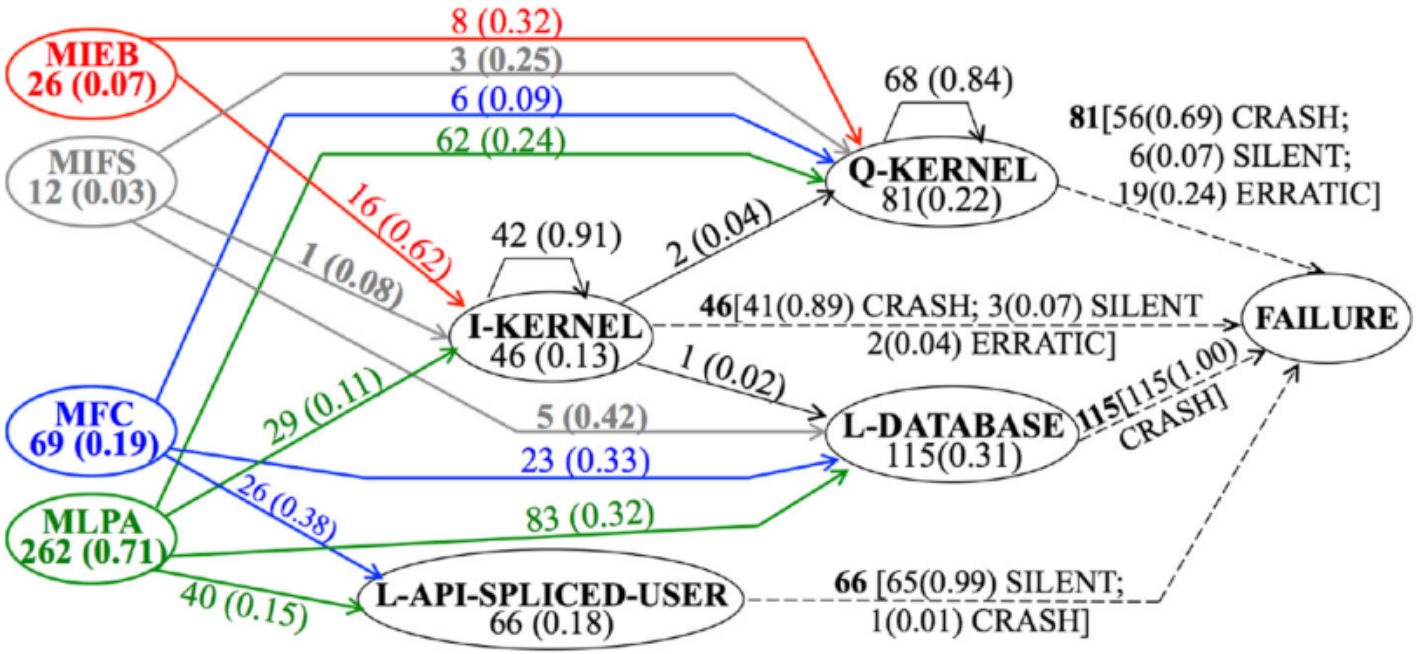
Can we use event logs to understand the error behavior of a system?

- We developed a methodology leveraging field data (including event logs) to:
  - Characterize the error behavior of complex critical software systems
  - Measure the effectiveness of field data sources at reporting failures occurred in the system
  - Understand how the errors propagate through the system components leveraging **error propagation graphs**



Error propagation with event logs

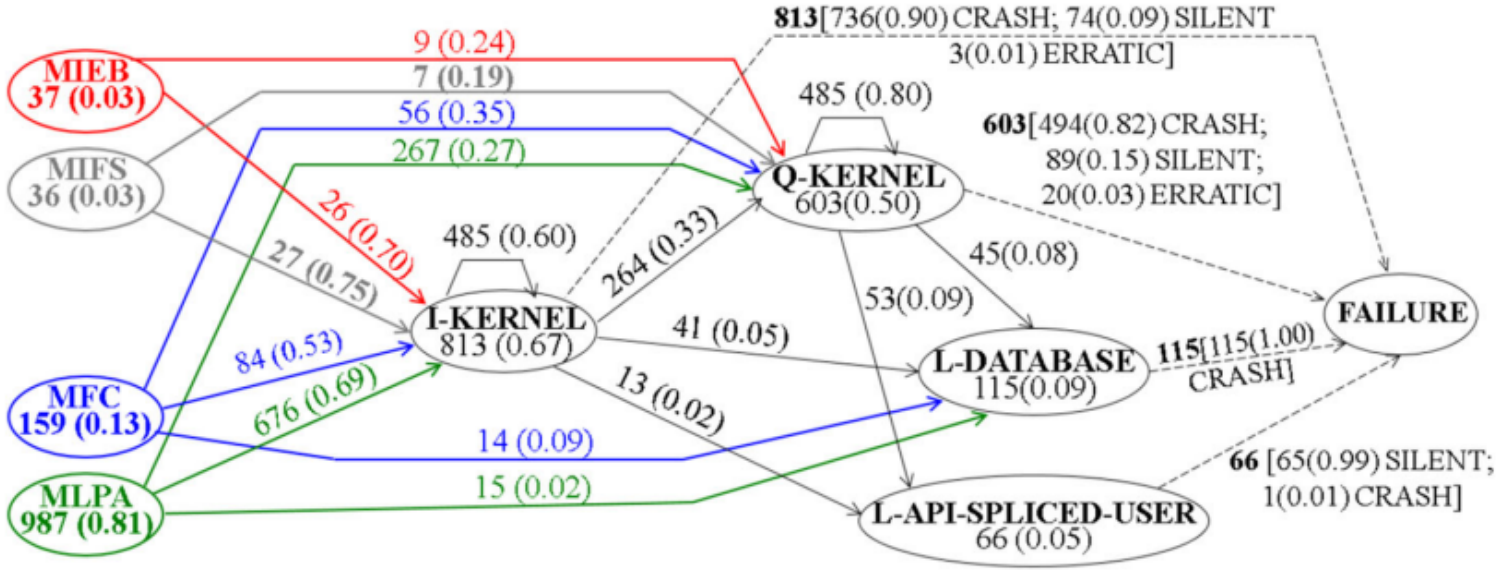
- Error propagation graph for algorithm faults (i.e., Missing Function Call - MFC, Missing small and Localized Part of the Algorithm - MLPA) from event logs of an ATC middleware



- Many errors are not reported by the immediate and quick components (i.e., Kernel)
- The latest error propagation steps determine the type of failure encountered by the system

But if we can instrument the code?

- Error propagation graph for algorithm faults combining both event logs and source code instrumentation trace of an ATC middleware



- Significant improvement of the reporting ability of the kernel component
- Allow to report previously undetected errors