Concurrent Exception Handling and Resolution in Distributed Object Systems

Presented by Prof. Brian Randell

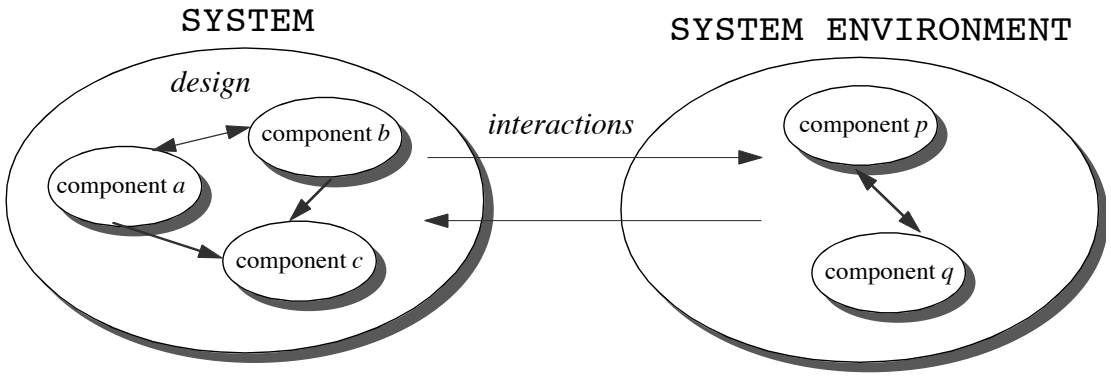
J. Xu
University of Durham

A. Romanovsky and **B. Randell**
University of Newcastle upon Tyne

Outline

- System Model and Fault-Tolerant Components
- Principles of Exception Handling
- Exceptions in Concurrent and Distributed Systems
- Atomic Actions and Error Recovery
- Coordinated Atomic Actions
- Concurrent Exception Handling and Resolution
- A Case Study

System Model



A software **system** consists of a number of **components**, which cooperate under the control of a **design** to service the demands of the system **environment**

The components and the system environment may be also viewed as systems. The design can be considered as a special component that defines the interactions between components and establishes connections between components and the system environment

Failures, Errors and Faults

- A system **failure** occurs when the delivered service deviates from what the system is aimed at (e.g. specification)
- An **error** is that part of the system state which is liable to lead to subsequent failure
- A **fault** is the (hypothesized) cause of an error
- A fault can be a physical defect, imperfection, or flaw that occurs within a hardware or software component
- In particular, a fault which occurs in the environment of a system is called an **environmental fault**; it may cause an error in the system

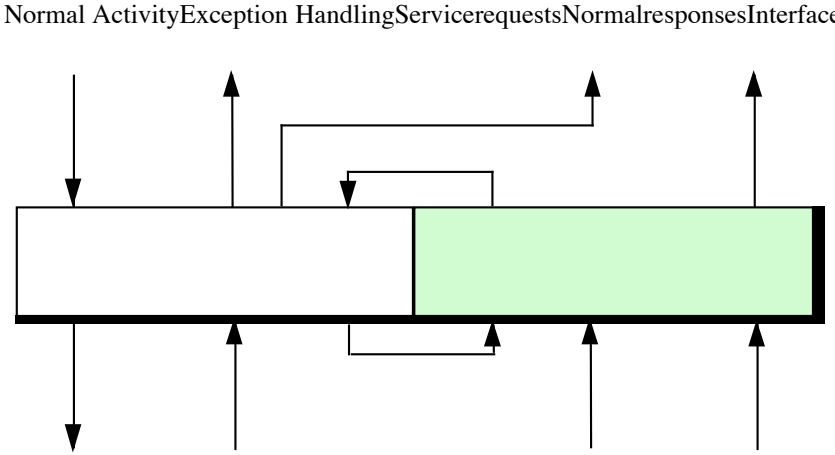
Exceptions

- There are different definitions of exceptions. For example, an **exception** may be defined as an error or an event that occurs unexpectedly or infrequently
- Traditionally, an exception does not necessarily imply the occurrence of a fault; it can be used in normal processing to supply extra information about results
- To avoid any ambiguity we define an exception as an abnormal response from, or an abnormal event inside, a component that indicates the detection of one or more errors in the component
- **Exception handling** is the immediate response and consequent action taken to handle one or more exceptions

Error Recovery Techniques

- **Forward error recovery**: the system is returned to an error-free state by applying corrections to the damaged state. Such an approach demands some understanding of the errors
- **Backward error recovery**: the system is recovered to a previous error-free state. No knowledge of the errors in the system state is required
- Forward and backward error recovery techniques complement one another
- Forward error recovery allows **efficient handling** of expected exception conditions, and backward error recovery provides a **general strategy** which will cope with faults a designer did not (or chose not to) anticipate

Ideal Fault-Tolerant Component

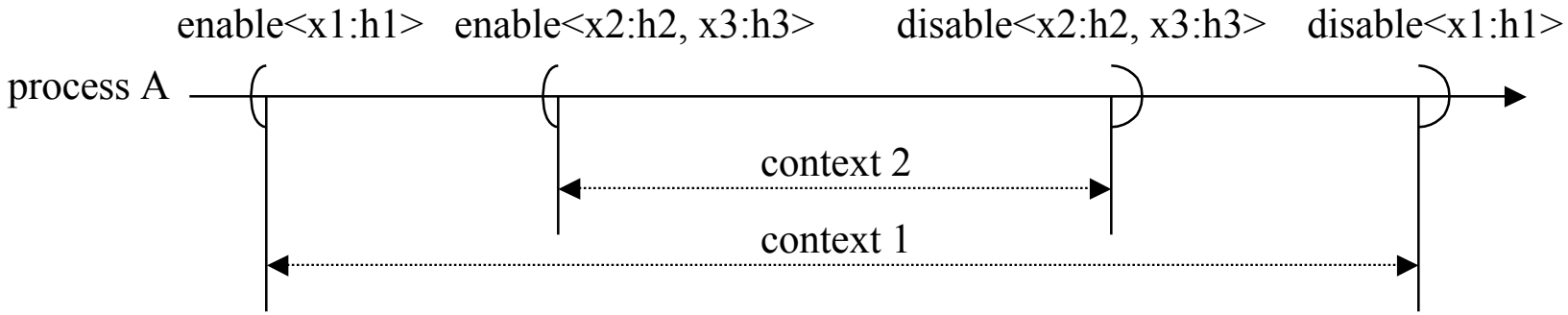


A **framework** for fault tolerance can be provided by the notions of exceptions, exception handling, and forward error recovery

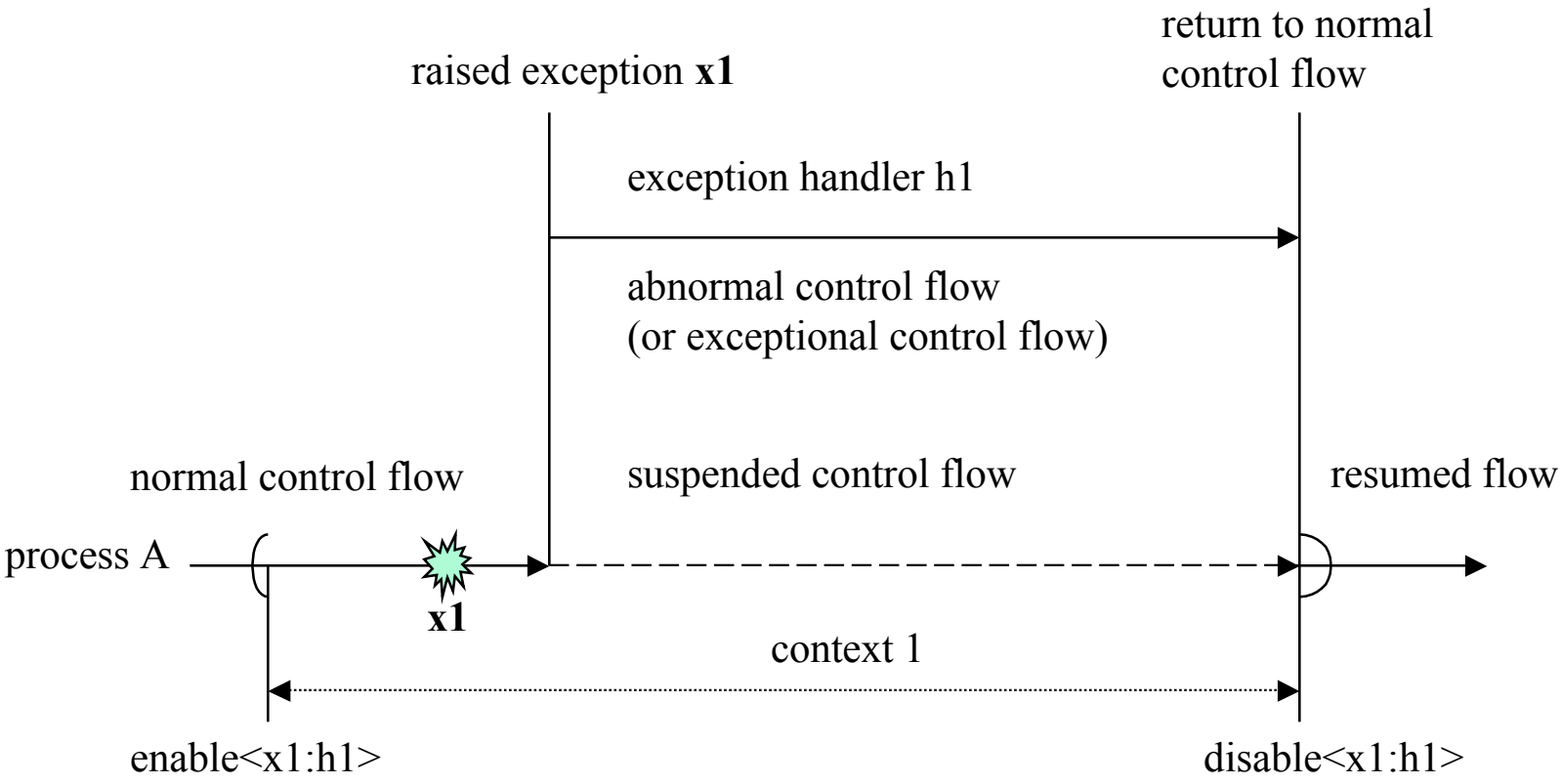
It specifies the relationship between the **normal** and **abnormal** activity and between the **raising** and **signalling** of exceptions (an exception is raised within the component, but signalled between components)

Flow of Control and Handling Context

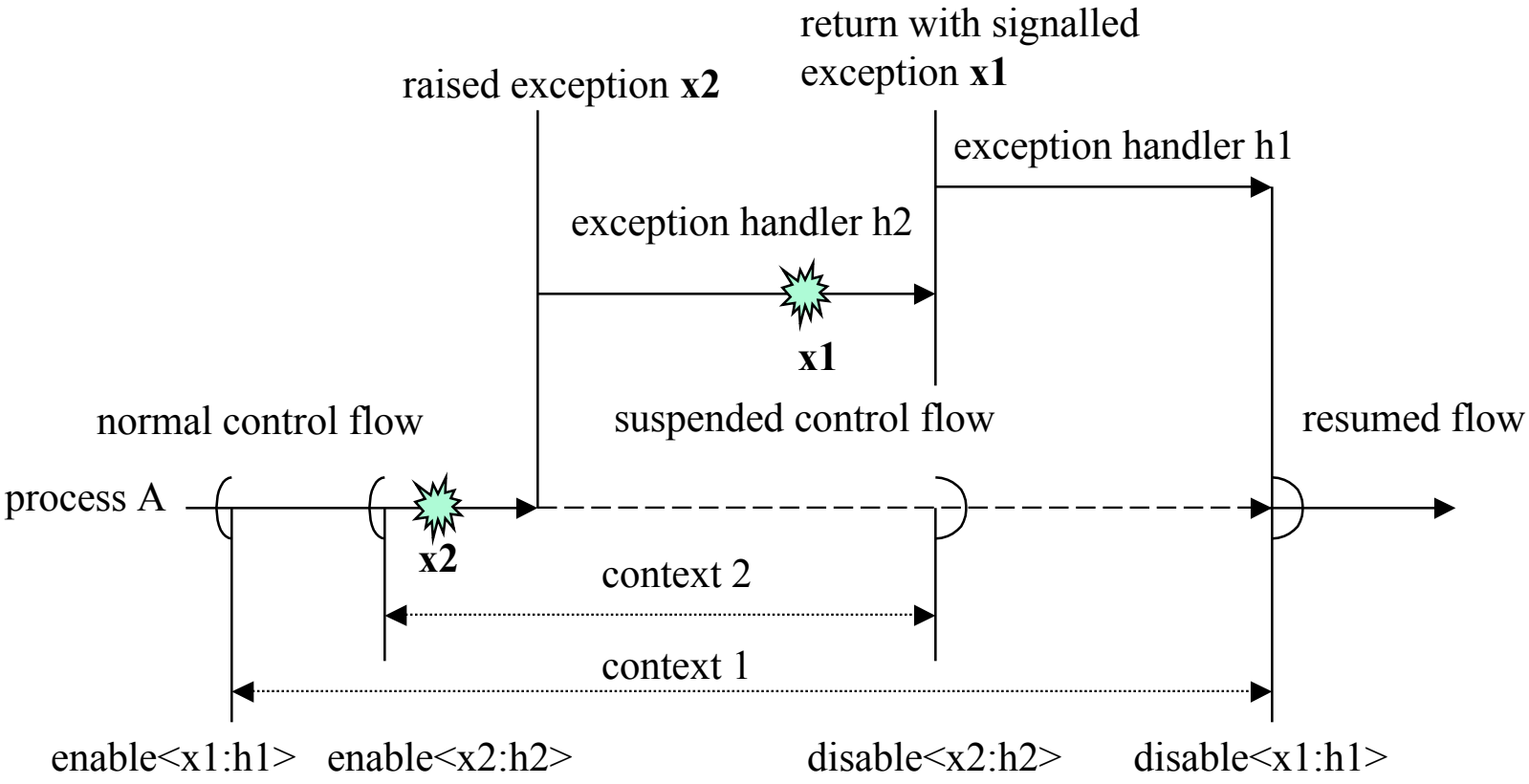
- The **flow of control** of a computation within a component should change as the result of a raised exception
- Such an **exceptional flow of control** is distinguished from the normal flow of control
- Within a program, exceptional flow of control is associated with code fragments that called **exception handlers**
- Exceptions, software components, and exception handlers are associated by a **handling context**



Example of Successful Forward Recovery



Example of Returning an Abnormal Response



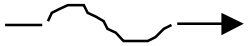
Concurrent/Distributed Systems

- Exception handling and the provision of fault tolerance are more difficult in concurrent/distributed systems than in sequential programs, e.g. several exceptions may be raised concurrently in multiple concurrent activities
- Exception propagation in concurrent programs may not simply go through a chain of nested callers, e.g. an exception may need to be propagated to the members of a cooperative process group
- Physical distribution of computation complicates further the cooperation and coordination of multiple concurrent components
- Damage confinement and assessment become more difficult in systems involving complex interactions among concurrent activities

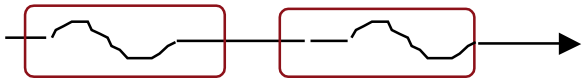
Abstract System Model: Elements

Objects: 

An object is a named entity that combines a data structure (internal state) with its associated operations that determines the externally visible behaviour of the object

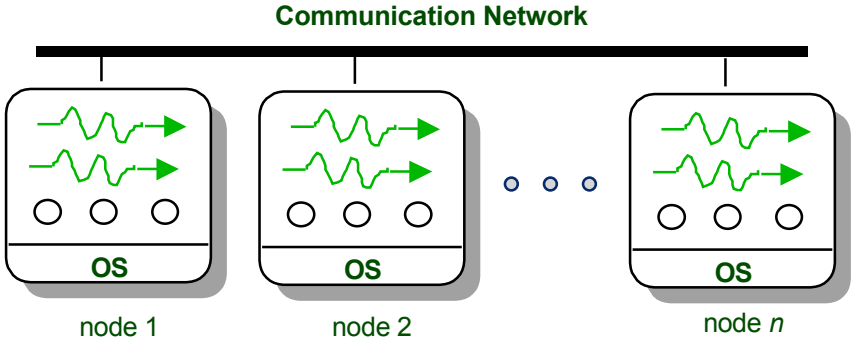
Threads: 

Threads are the agents of computation (in a concurrent system). A thread is an active entity that is responsible for executing a *sequence* of operations on objects

Actions: 

An action is a programming abstraction that allows the application programmer to group a *set* of operations on objects into a logical execution unit

Physical Distribution:



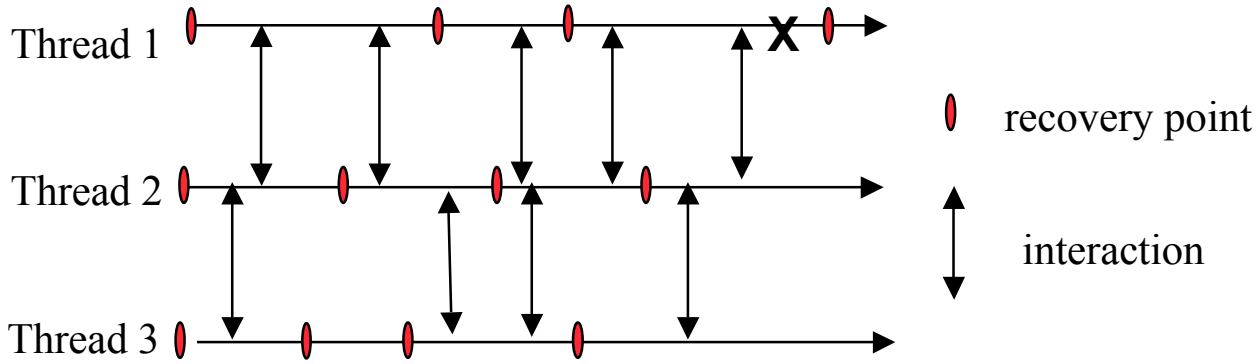
Atomic Actions

- When an exception is handled, in general, damage confinement and error recovery will follow
- An important dynamic structuring concept which assists in these activities is that of an **atomic action**
- The activity of a group of components constitutes an atomic action if there are no **interactions** between that group and the rest of the system for the duration of the activity
- **Transactions** and **conversations** are two different instances of the notion of an atomic action
- Ideally, if the system is known to be in an error-free state upon entry to an atomic action, and an exception is raised during its execution, then only those components which have participated in the atomic action need to be recovered

Error Recovery: Assumptions and Complications

- In a concurrent/distributed system, the problems of error recovery vary very greatly depending on what **design assumptions** can be justified
  - if one disallows (i.e. ignores) the possibility of undetected invalid inputs or outputs, backward error recovery suffices
  - if users and hence their activities are independent, and merely **competing**, e.g. to use a database, then conventional transaction processing techniques can provide such recovery
- But the activities in a concurrent/distributed system will on occasion at least be attempting to **cooperate**, in pursuit of some common goal
- Moreover, these activities involve other entities (e.g. devices and humans) that cannot simply go backwards when an error is detected
- Thus forward rather than backward recovery will have to be used, and complex issues of cooperation and competition must be addressed

Backward Recovery: The Domino Effect

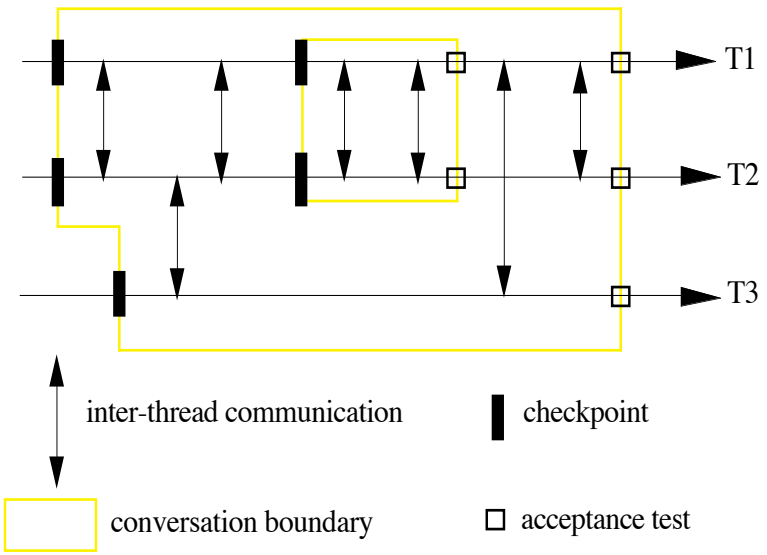


- If interactions are not controlled, and appropriately coordinated with checkpoint management, rollback of one thread can result in a **cascade of rollbacks**
- However, the possibility of the **domino effect** could be ignored if it could safely be assumed that data was fully validated before it was output, i.e. transmitted from one thread to another
- Similarly, the effect would not occur if inputs could be validated completely

Transactions and Conversations

- The **data validation** assumption underlies simple **transaction**-based systems - outputs are allowed only after a transaction has been **committed**
- In such systems the notion of commitment is regarded as absolute
- Nested transactions can be used to limit the amount of activity that has to be abandoned when backward recovery is invoked
- Typically, it is still assumed that there are absolute **outermost** transactions, and that outputs to the world outside the database system, take place after such outermost transactions end - and are presumed to be valid
- **Conversations** provide a means of coordinating the recovery provisions of interacting threads so as to avoid the domino effect, without making assumptions regarding output or input validation

Nested Conversations



Inter-thread communication is only between threads that are participating in a conversation together

- on entry to a conversation a thread establishes a **checkpoint**
- if an error is detected by any thread then all the **participating threads** must restore their checkpoints
- after restoration all threads then attempt to make further progress
- all threads leave the conversation together

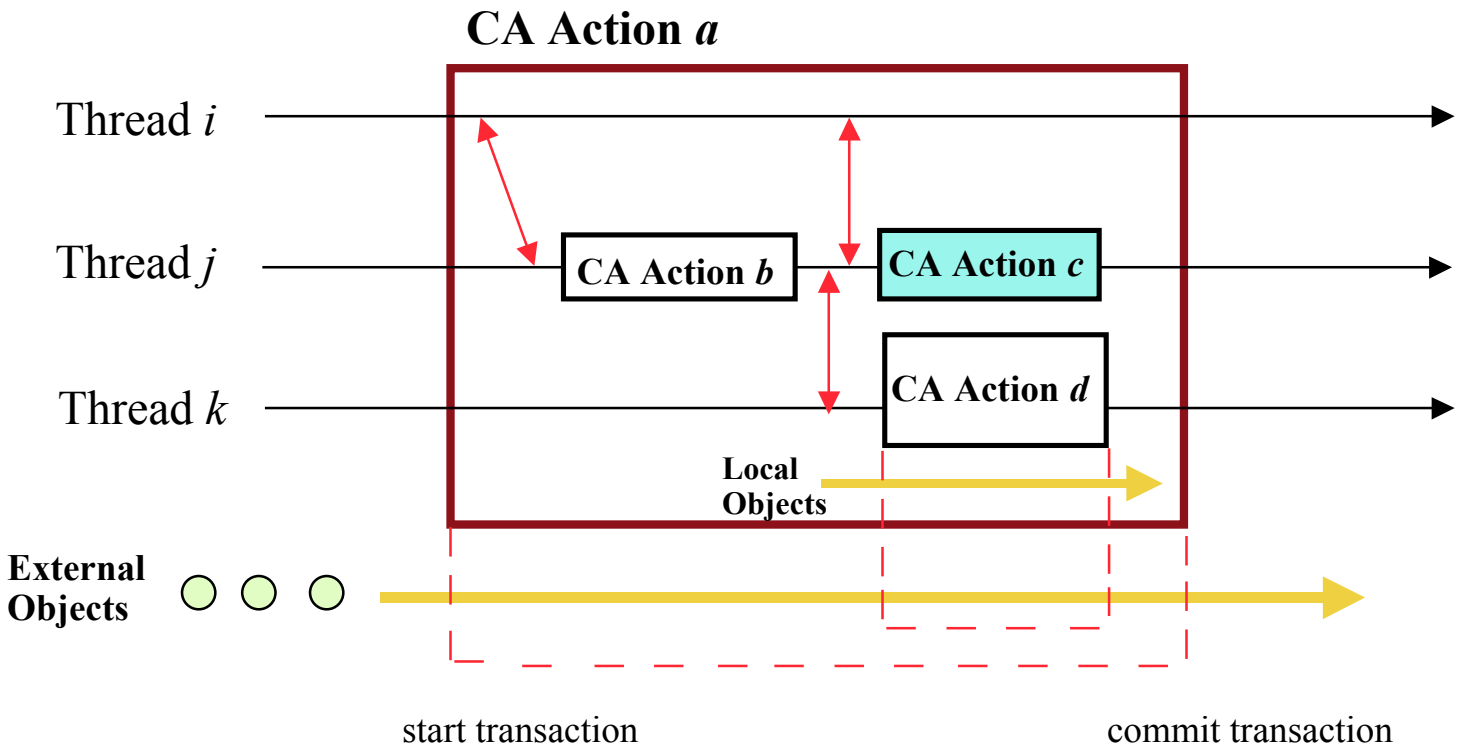
Transactions, Conversations, and Error Recovery

- **Transactions**, which can be nested and multi-threaded, provide concurrency control and backward recovery for competing activities that are sharing external resources
- **Conversations** provide coordination and backward recovery for cooperating activities, but do not support use of shared external resources
- A **Generalised Conversation** allows for exceptions - when an exception occurs, every thread in the conversation has to switch to an appropriate handler, so that forward error recovery is performed
- A **Coordinated Atomic Action** (CA Action) can be regarded as a generalised conversation that also provides controlled access to shared external resources. Equally, it can be regarded as a nested multi-threaded transaction with disciplined exception handling

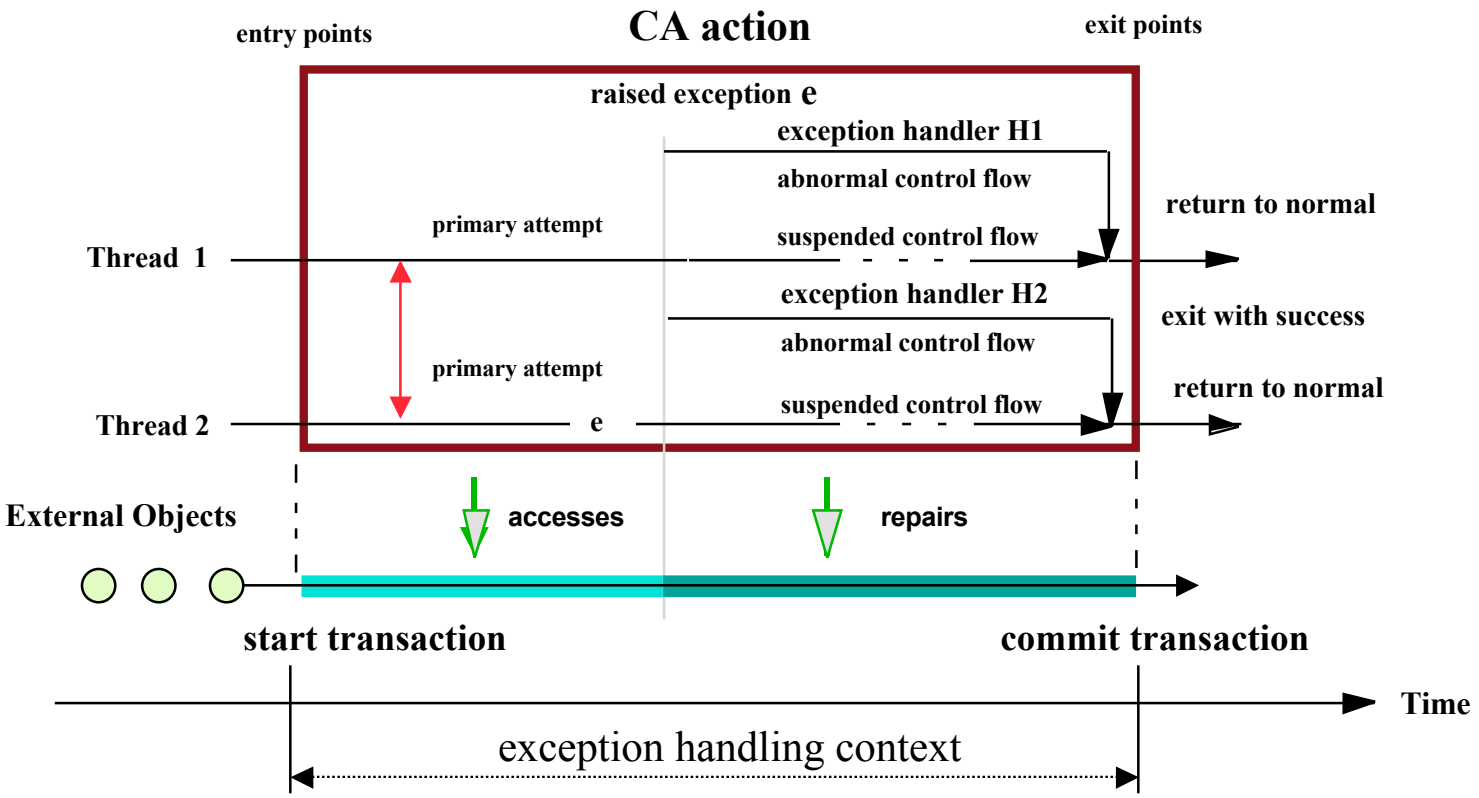
Coordinated Atomic Actions (CA Actions)

- A CA action is a generalized form of the basic atomic action structure
- **Multi-Threaded Enclosure and Coordination** - CA actions provide a mechanism for enclosing and coordinating interactions among threads, and must ensure consistent access to objects in the presence of complex concurrency and potential faults
- **Fault Tolerance** - If an exception is raised inside a CA action, appropriate forward and/or backward recovery measures will be invoked cooperatively in order to reach some mutually consistent conclusion
- **Multiple Outcomes** - CA actions combine exception handling with the nested action structure to allow multiple outcomes, e.g. a normal outcome or some possible exceptional outcomes

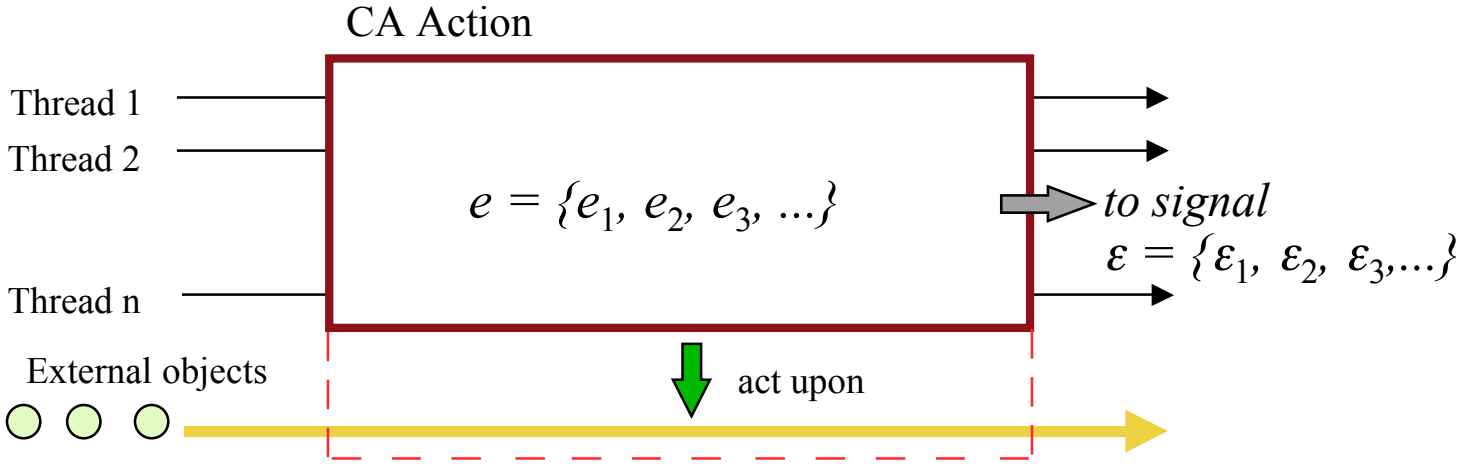
Illustration of CA Actions (Recursive View)



CA Actions - Internal Exceptions



Exception Classification and Declaration

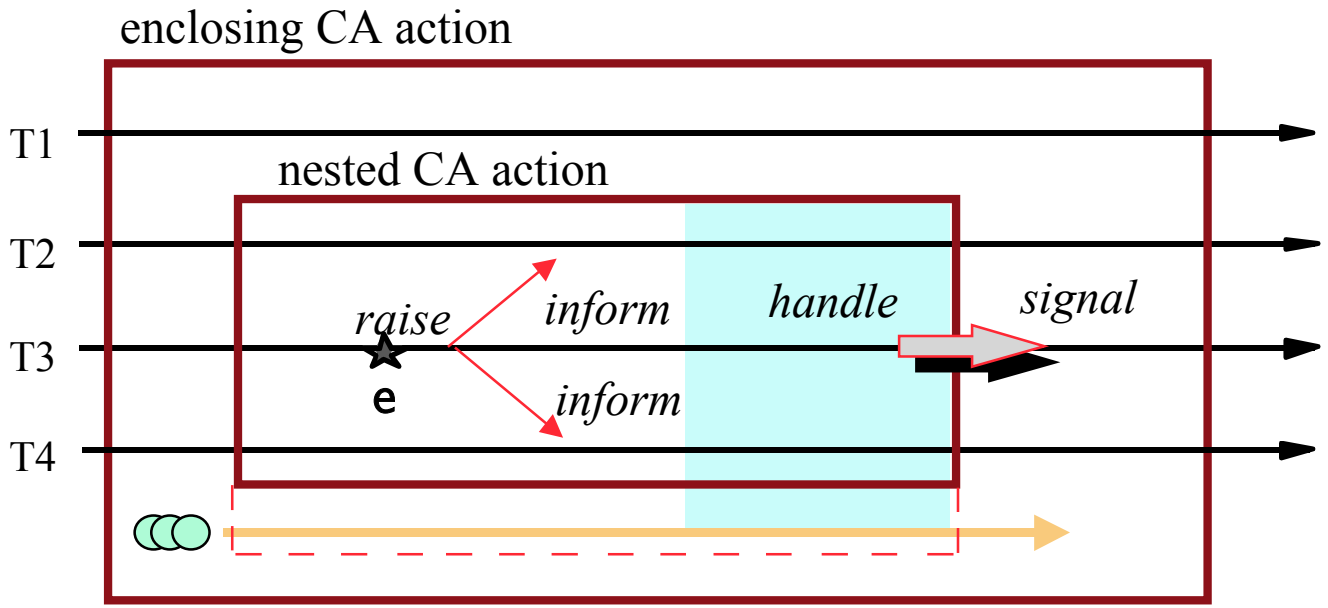


$e = \{e_1, e_2, e_3, ...\}$ Exceptions inside the CA action must be declared with the action definition and handled within the action

$\varepsilon = \{\varepsilon_1, \varepsilon_2, \varepsilon_3, ...\}$ Exceptions to be signalled from the action to its environment (e.g. the enclosing action) must be specified in the CA action interface

Recursive relation: ε_{nested} is a subset of $e_{enclosing}$

Exception Handling and Propagation



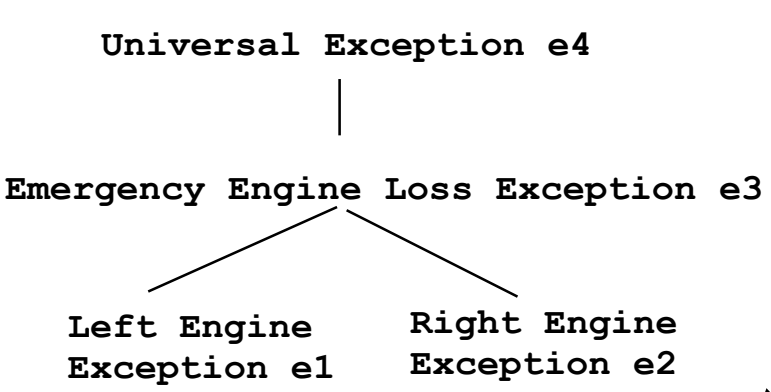
Concurrent Exceptions

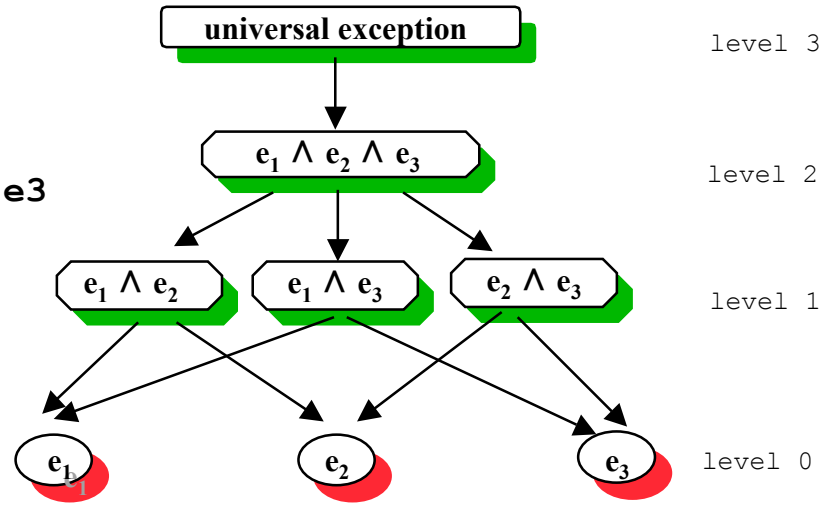
- In a concurrent/distributed system, different processing nodes may raise different exceptions and the exceptions may be raised simultaneously
- Concurrent exceptions must be handled in a coordinated manner

  Example: If just the left (or right) engine of a twin-engine aircraft fails, the controls can be adjusted appropriately to compensate for the loss of the left (right) engine. However, if both the right and left engine fail at the same time, handling both engine-loss exceptions separately and in some order can lead to catastrophic consequences
- There are a variety of **reasons** why several exceptions may be raised concurrently. For example, it is often difficult to interrupt the normal operations of the other nodes immediately after an exception has been raised

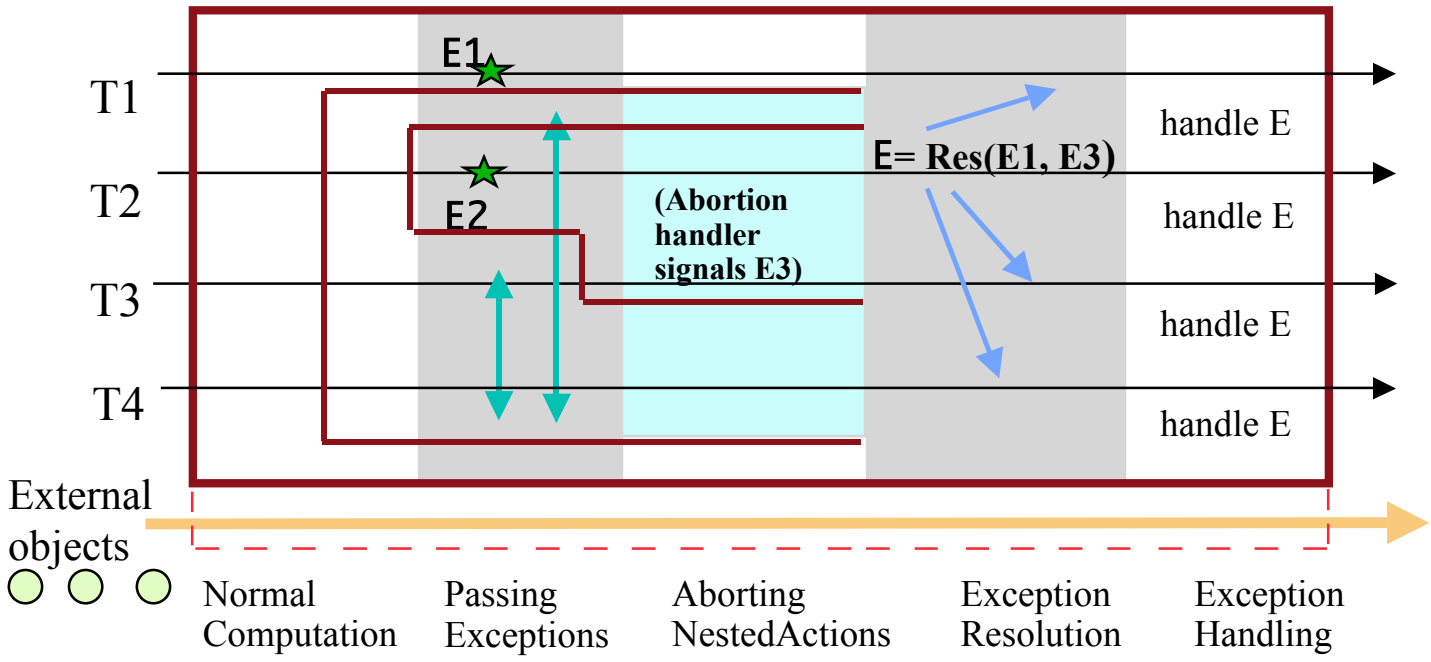
Concurrent Exception Resolution

An **exception graph** approach is developed in order to find the exception that "covers" all the exceptions raised concurrently

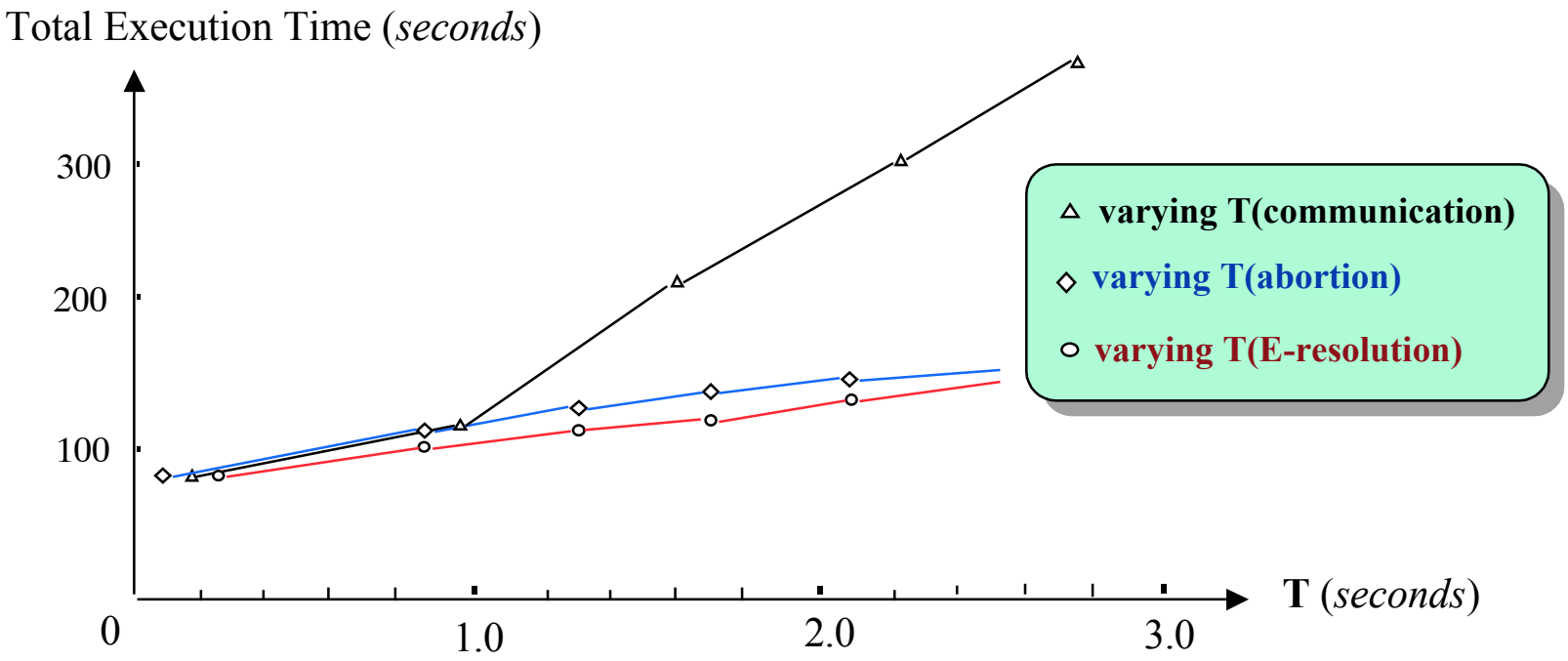




Dealing with Concurrent Exceptions



Performance-Related Evaluation



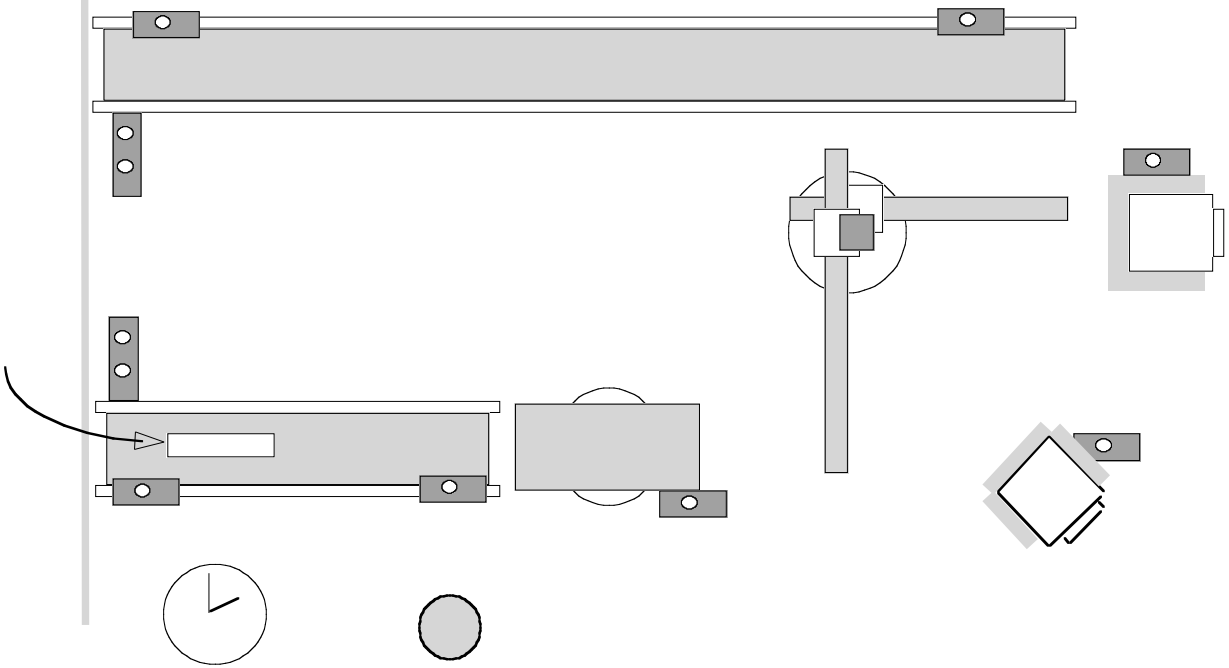
The cost of **message passing** is of the major concern, while concurrent exception handling does not introduce a high **run-time overhead**

A Case Study

- The FZI (Forschungszentrum Informatik, Germany) have specified and provided a simulator for the **Fault-Tolerant Production Cell**
- This represents a manufacturing process involving six devices: two conveyor belts (a feed belt and a deposit belt), an elevating rotary table, two presses and a rotary robot that has two orthogonal extendible arms
- The **task of the cell** is to get metal blanks from its "environment" via the feed belt, transform them into forged plates by using one of the presses, and then return them to the environment via the deposit belt
- **The challenge posed by FZI** - to design a control system that maintains specified safety and liveness properties even in the presence of a large number and variety of device and sensor failures, and which will continue to operate even if one of the presses is non-operational
- **Our aim** - to show how concurrent exception handling and CA Actions aid both the design and validation of this control system

The FZI “Fault-Tolerant” Production Cell

!deposit belttraffic light for deposittraffic light for insertion**robotarm_1arm_2syste**



Failure Definition and Analysis

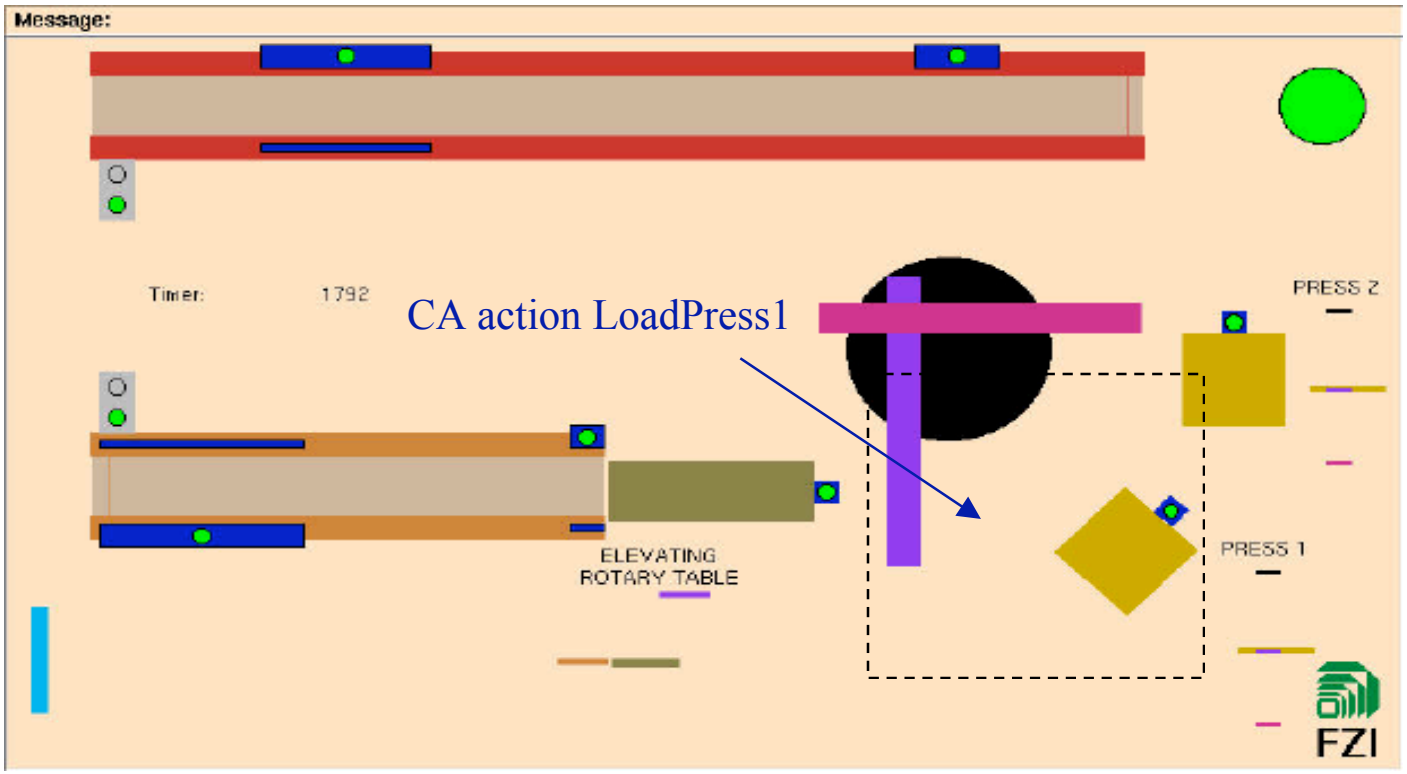
- For a given device, we classify possible failures into: i) sensor failures, ii) actuator failures, and iii) lost or stuck blanks
- A failure may be detected by sensors, actuators, stop watches, singly or in combination
- We discuss failure detection only, because in many cases certain different types of failure cannot be distinguished using just the on-line information available; subsequent off-line diagnosis is often needed

  Example: For a press, failures of the position sensors and failures of the press actuator can be detected by assertion statements in the control program. Such failures must be reported to the user through an alarm, but normal operations can be maintained using a single press
- A device or sensor failure should not affect normal operations of other devices. CA actions help confine damage and failures, and minimize the impact of any component failure on the entire cell

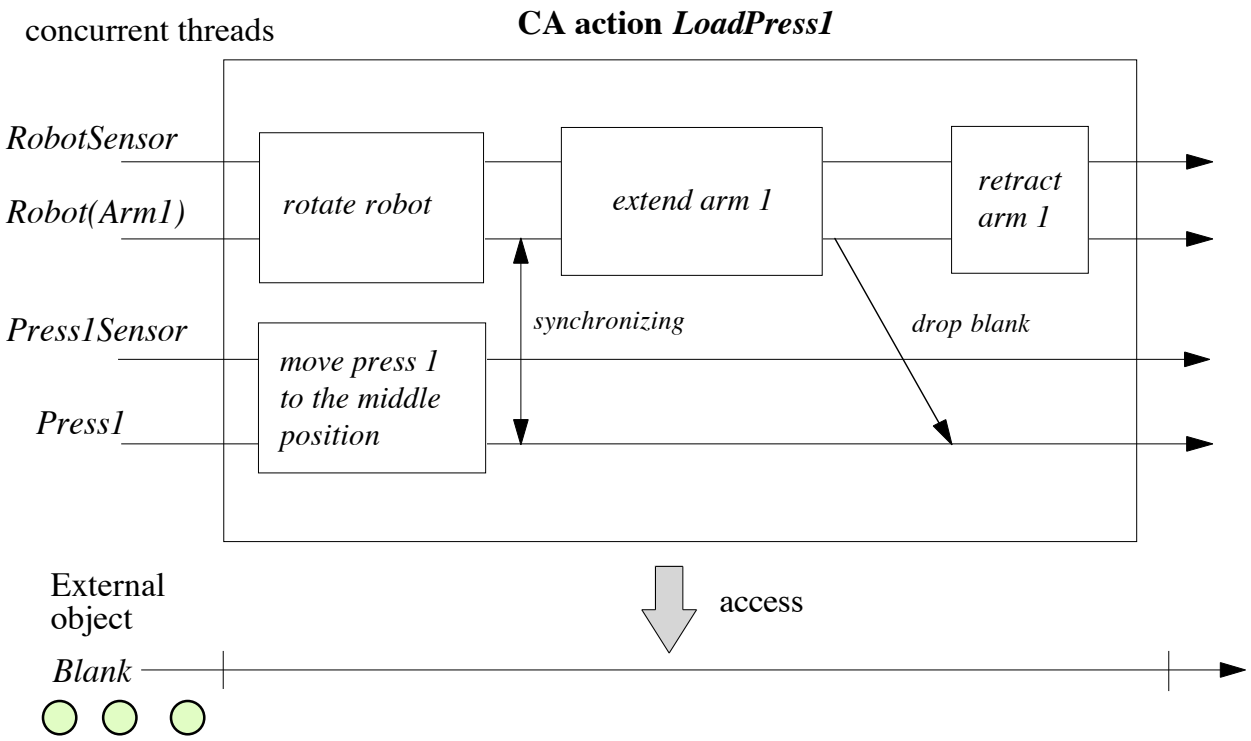
Design of a Control System

- Our design for the Fault-Tolerant Production Cell uses 12 main CA actions; each action controls one step of the blank processing and typically involves passing a blank between two devices
- We have implemented this design for a control program using a Java implementation of a distributed CA action support scheme (this scheme makes use of the nested multi-threaded transaction facilities provided by the Arjuna transaction support system)
- Our system controls the FZI simulator. Overlaid on the screen are rectangular outlines showing the scopes of these CA actions
- As the simulator is operated each of these outlines is gradually shaded in each time the corresponding CA action is executed - the colour of this shading is changed when the CA action is involved in exception handling, in response to simulated faults

Production Cell with One CA Action



Example: CA Action `LoadPress1`



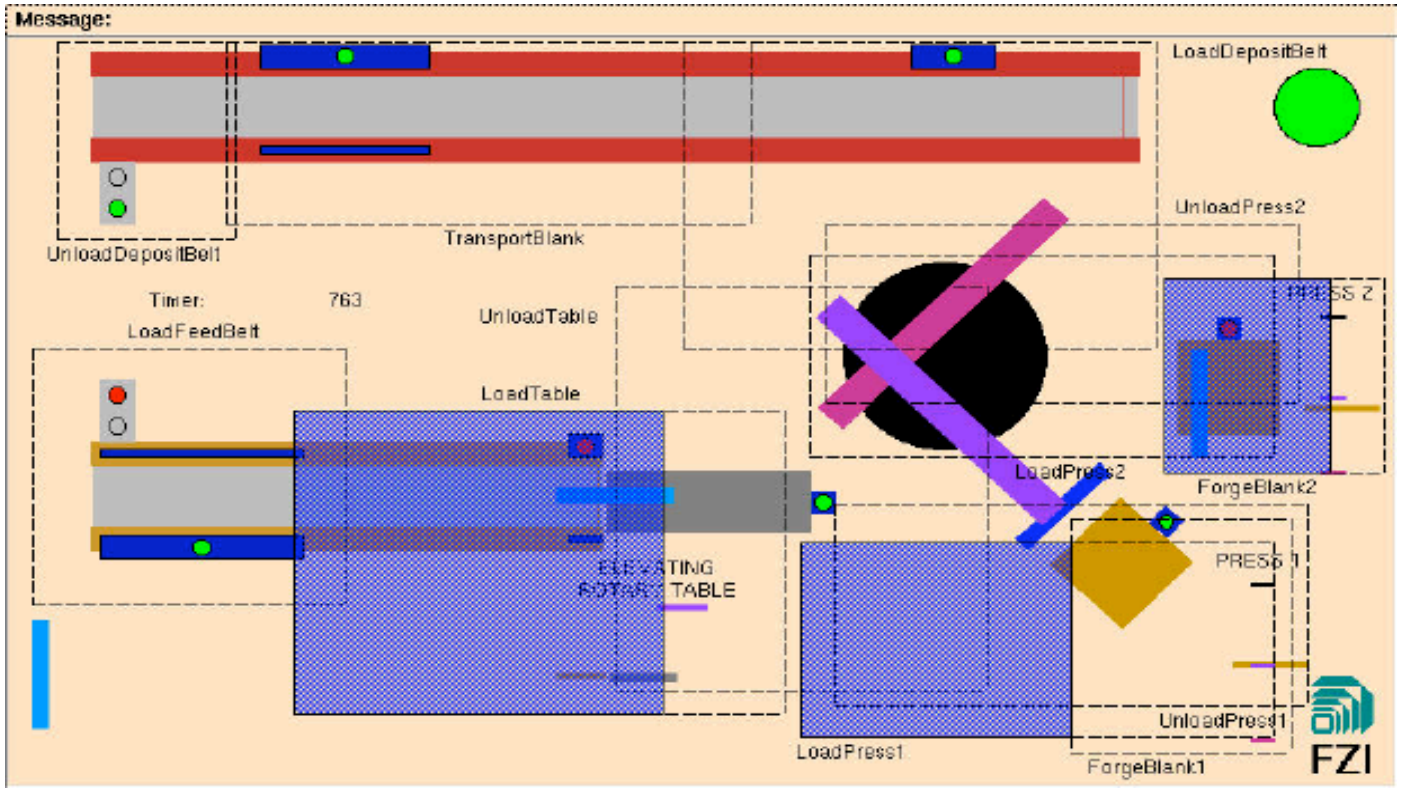
Example: CA Action LoadPress1 (Specification)

```
CAA LoadPress1;Interface    Use MetalBlank;    Roles Robot: blankType, robotActuator;Pre
```

Design Strategy

- The main characteristics of our design are the way it separates safety, functionality, and efficiency concerns. In particular, the **safety requirements** are satisfied at the level of CA actions, while the other requirements are met by the device/sensor-controllers
- Each CA action encloses a set of devices that must interact. If two such actions are shown as overlapping, this indicates that they must not be performed in parallel because they both involve the same device (the CA action semantics guarantee this)
- The CA actions were **designed**, and **validated**, **independently** of each other, and of the set of device/sensor-controllers that dynamically determine the order in which the CA actions are executed

Production Cell in Operation

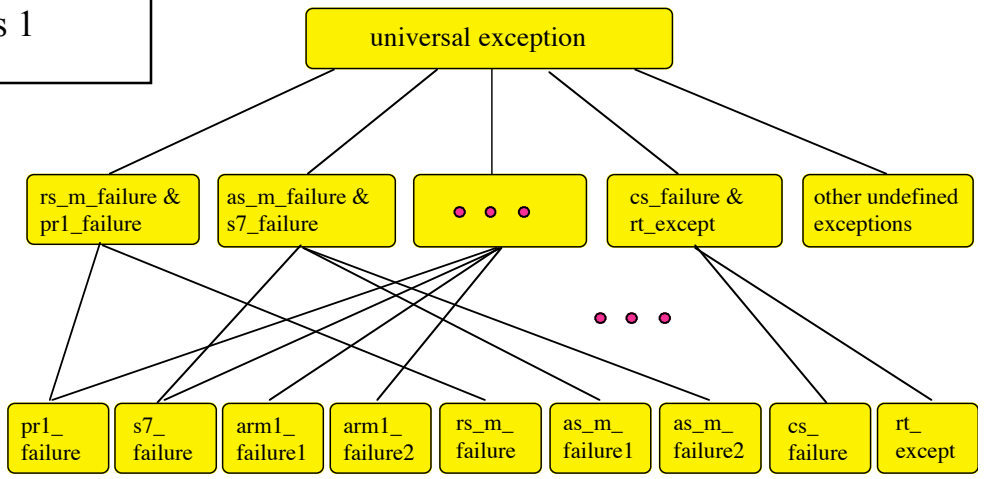


Handling Concurrent Exceptions

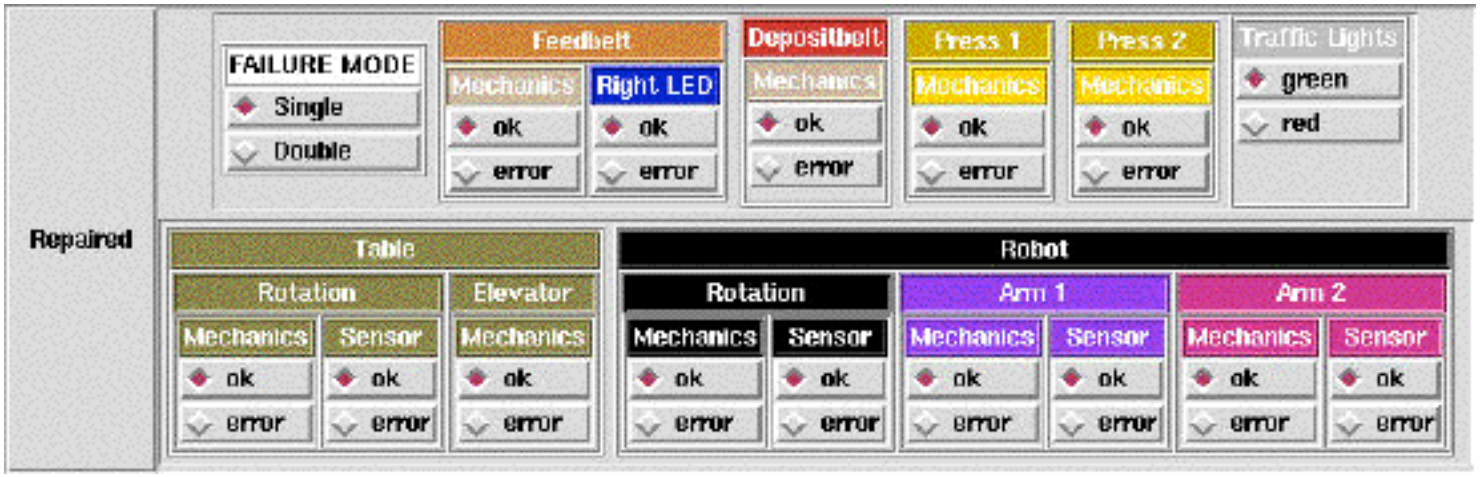
exception to signal	exceptional post-conditions
(robot's rotary sensor or motor failure) & press1 failure	robot off blank on arm 1 both arms retracted press 1 off no blank in press 1

In the cell concurrent failures can be detected effectively but often cannot be distinguished. In those cases, our control program is designed simply to bring the system to stop in a safe state

For each CA action, various exceptions are defined based on failure analysis, and an exception graph for resolving concurrent exceptions is constructed



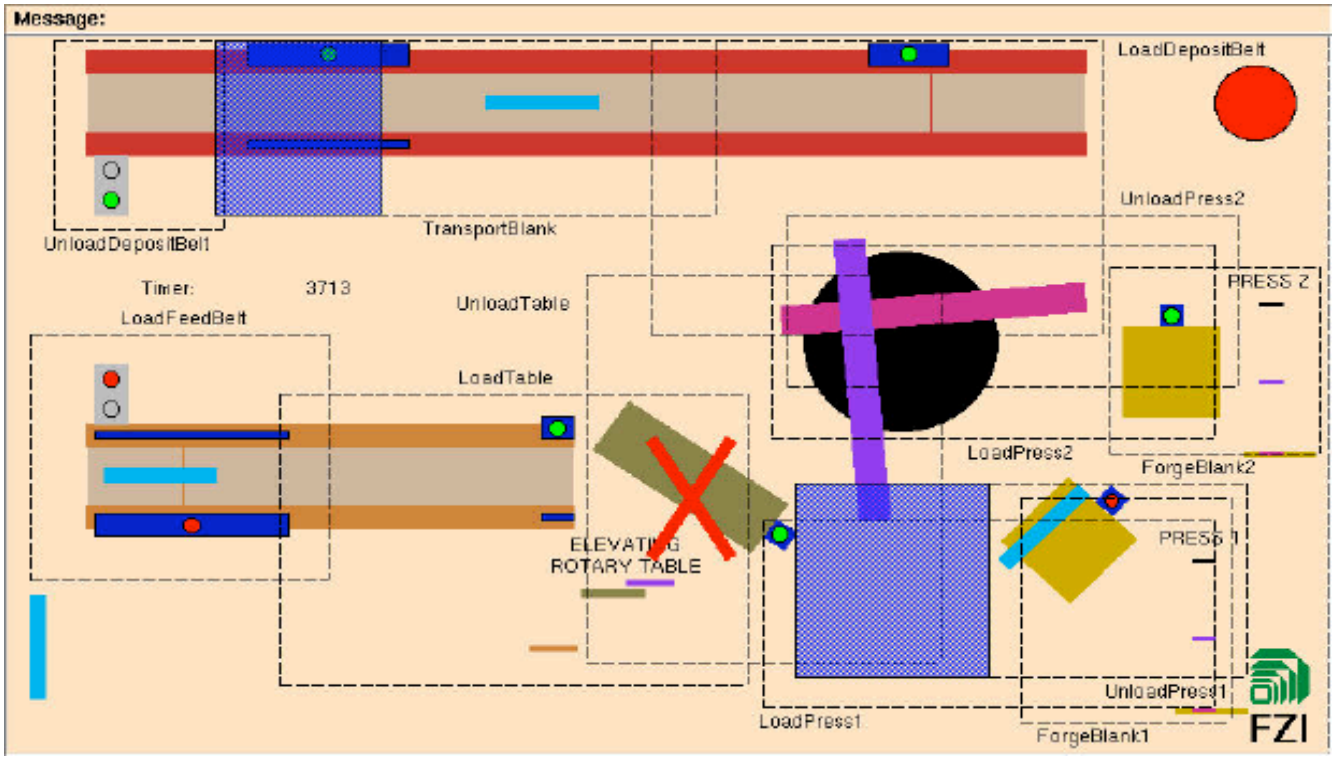
The Fault Injection Panel



Single faults and multiple concurrent faults, of many different types, can be injected via this on-screen control panel

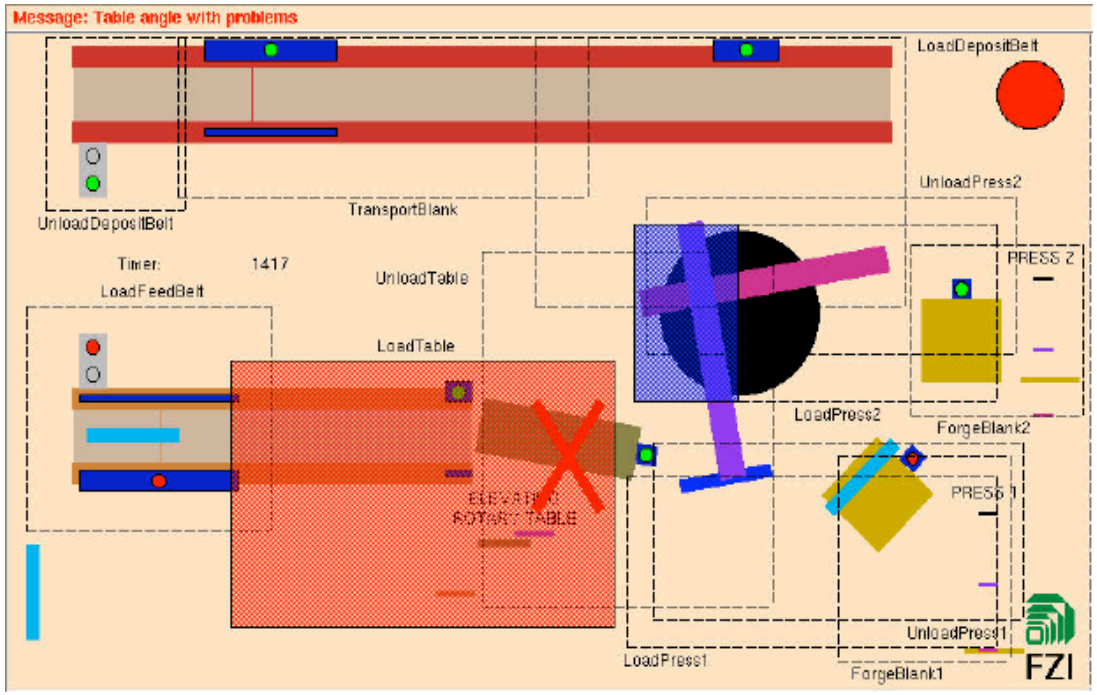
After faults are injected into the simulator, error detection measures embedded in our control program will detect the errors caused by the faults and raise one or more corresponding exceptions

An Injected Fault



Effective Exception Handling

All injected device or sensor failures were caught successfully and handled immediately by our control program. A previously unknown bug in the FZI simulator was detected by a CA action and recovered automatically by the action using the retry operation



Summary

- An atomic action is an important dynamic structuring concept that assists in damage confinement and error recovery; a **coordinated atomic action** is a generalized form of the basic atomic action structure with disciplined exception handling
- Exception handling in concurrent/distributed systems is more difficult than in sequential programs; **concurrent exceptions** must be handled in a coordinated manner
- The Fault-Tolerant Production Cell **case study** demonstrates how CA actions can act as a structuring tool, supported by exception handling and fault tolerance, and aid both the design and validation of a dependable control system

In Conclusion

- We have introduced a **systematic** and **model-driven approach** for building dependable software systems based on a variety of **exception handling** techniques and **fault tolerance** schemes (from basic concepts to systems designs, from sequential programs to concurrent/distributed systems, and from experimental analysis to realistic industrial applications)
- It was very pleasing to confirm from our experience that the **combination** of advanced **fault tolerance techniques** and powerful **system structuring mechanisms** (e.g. object-oriented structuring methods and high-level control abstractions) often offers a quite straightforward solution to complex reliability and safety problems