Dependability of robotic applications

L. Nana

Plan

- Brief overview of dependability
- Error prevention
- Fault tolerance
- Software fault tolerance
- Case study with PILOT and its environment

Dependability

- Property that allows users of a system to place justified trust in the service it delivers.
- 2 main approaches :
  - Error prevention:
    - Traditional approach.
    - Goal : Spread the faults before regular use of the system.
      => other name : *fault intolerance*.
    - Based on *test* and *formal verification* methods.
  - Fault tolerance:
    - Ability of a system to deliver a correct service even in the presence of faults.
    - Relies on *redundancy*.

Basic definitions

- *Service delivered* by the system: behavior of the system as perceived by its user (s).
- *User* : another system (human or physical) that interacts with the system.
- *Failure* : the service delivered no longer fulfills the function (s) of the system.
- *Error :* Part of the system state that is likely to cause a failure.
- *Fault :* cause assumed or adjudged of an error.
- Recursive causality relationship:

  ….. → Failure → fault → error → failure → …

Classes of faults

- Classification useful to avoid mistakes and to prevent their possibly disastrous consequences on the functioning of the system.
- Three main classification criteria:
  - Nature :
    - Accidental.
    - Intentional.
  - Origin:
    - Human.
    - Physical phenomenon.
    - Faults internal to the system,
    - External faults resulting from the interaction between the system and its environment.
  - Temporal persistence:
    - Permanent faults.
    - Temporary faults.

Classes of failures

- A system does not always fail in the same way.
- Three main criteria of classification :
  - Domain :
    - Value failure : the value of the service delivered does not conform to its specification.
    - Temporal failure : service delivery times are not compliant with the specification (early or late failure).
  - Users perception:
    - Coherent failures.
    - Inconsistent failures, also called Byzantine failures.
  - Consequences on environment:
    - Minor or benign failures.
    - Catastrophic failures.

Formal verification

- 2 main approaches: Model-checking and demonstration.
- Model-checking: fully automated, 2 sub approaches :
  - Synchronous:
    - Characterized by the fact that it considers cyclic systems, whose overall behavior, by synchronous composition, can involve a set of events (qualified as simultaneous) on the resulting transition.
    - Widely used for designing reactive systems.
    - Has given rise to many complete programming environments including some dedicated to control-command (STATECHARTS, SYNCCHARTS, GRAFCET, ...), or mission programming (ORCCAD).
  - Asynchronous: Petri nets, etc.
- Demonstration: tools of proof (PVS, COQ, ...).

Test

- Purpose: To minimize the chances of failure appearing when using the software.
- 2 approaches :
  - *Static test or control :* one seeks in a static way (without execution of code) simple and frequent faults.
  - *Dynamic test*:
    - Code executed.
    - Definition of *test data*, that is to say inputs that will be provided to the software during its execution.
    - *Test Set* or *Test Data Set:* A set of test data produced for testing.
    - Exhaustive test impossible => need to define a test set constituting a *representative sample* of all the possible entries.

Causes of software errors

- The software development process has an impact on the types of potential errors.
- Poor knowledge of the programming language or inexperience of the programmer.
- Distortion or loss of information during the development process.
- Bad specification or misunderstanding of specifications, etc.

Classification of software errors

- 6 classes of errors:
  - *Calculation errors* : for example, write "x: = x + 2" instead of "x: = y + 2".
  - *Logic errors :* bad predicate expression. For example, by writing "if (a <b) then" instead of "if (a> b) then".
  - *I / O errors :* bad formatting, bad access to communication medium, etc.
  - *Interface errors :* bad communication between the software's internal components (e.g. call of the P1 procedure instead of P2, incorrect parameter passing, etc.).
  - *Data processing errors :* bad access or mishandling of data (misuse of pointers, undefined variables, overflow of a table index, etc.).
  - *Data definition errors :* erroneous type in the declaration of a variable (for example, a variable was declared as integer when it should have been declared as real), error in accuracy (for example, a value is in simple precision instead of double).

Classification of test techniques

- According to the criterion adopted for the choice of representative test data :
  - *Functional techniques* or *black boxes* : production of test data based on the specification of the software without worrying about the internal structure of the software.
  - *Structural techniques* or *white boxes* : Test data are produced by analyzing the source code.
- Depending on the execution or not of the binary code:
  - *Dynamic test techniques* : The binary code is executed and the actual behavior of the program is examined.
  - *Static testing techniques*: The passive form of the program (source code) is examined.

Notes on the test

- The testing process often uses a combination of functional, structural, dynamic and static techniques.
- Each correction inevitably leads to a risk of new errors appearing with a frequency often greater than during the previous development of the software.

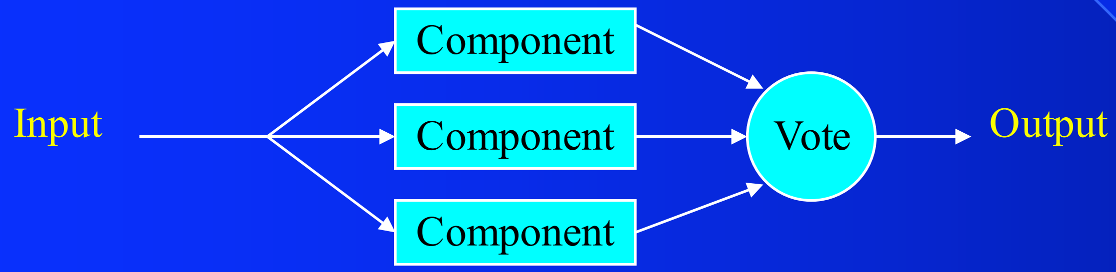
  => Need to re-run on the tested program, a significant part of the old TD (*non-regression technique*).

Redundancy

- Presence in the system of elements fulfilling the same functions as all (total redundancy) or part (partial redundancy) of the system.
- *Primary element :* part of the system whose errors are tolerated by the redundant part.
- Implementation: Duplication in terms of specifications or implementation of elements ensuring degraded specifications.
- 2 types of redundancies: static, dynamic.

Static redundancy

- The redundant element participates in performing the task before an error is detected and remains active.
- Typical example: triple modular redundancy



  - Tolerance of errors of one among the 3 components.
  - Implicit assumption that the probability that more than one element produces the same erroneous result is negligible.
- Generalization to N elements (odd N): *N modular redundancy*.
- Another name: *masking redundancy*.

Dynamic redundancy

- The redundant element only takes part in carrying out the task after detection and reaction to an error.
- 2 forms: active, passive.
- Active redundancy:
  - The redundant element is permanently active.
  - It receives the same information as the primary element and processes it in parallel.
  - Only the primary element outputs the results.
- Passive redundancy:
  - The redundant element remains passive in the absence of error.
  - In case of error, uses information stored before the detection of error to take over from the primary element

The different phases of fault tolerance

- 4 main functions:
  - Error detection.
  - Damage assessment.
  - Error handling.
  - Fault treatment.
- Error detection is always the first function performed.
- Order of intervention of the other steps not predetermined and possibility of strong interaction between them.

Error detection

- Based on hardware or software tests mechanisms dedicated to the monitoring of the state of the system.
- Finished by issuing an error signal (s).
- Ideal criteria for controls:
  - Be based solely on the specification of the service delivered.
  - Check the absolute conformity of the behavior of the system to its specification.
  - Be independent of the controlled system itself.
- Ideal criteria difficult to meet in practice:
  - Specification very often expressed in terms of information external to the computer system not taken into account by the computer verification.
  - Cost and performance constraints unacceptable in most cases, for runtime checks.
  - The independence between the system and its control can not be absolute.

=> Limitation to a standard lower than the ideal.

Damage assessment

- Purpose: Identification of failed components.
- Justifications:
  - Error detection signal very often insufficient to identify all the failed components.
  - Possible propagation of invalid information between the occurrence of the fault and its erroneous consequences.
- Two approaches:
  - Static approach:
    - Estimate a priori of the consequences of any error.
    - In case of error, all the components involved in the estimation of its consequences are supposed to be reached.
    - Difficult to adopt in complex systems.
  - Dynamic approach:
    - Exploring the state of the system after error detection to estimate the extent of damage.
    - Need to memorize the information on the different transfers and to control these transfers.

Error handling

- Aim: to eliminate errors, if possible before a failure occurs.
- 2 forms: error compensation and error recovery.
- Error Compensation: The faulty state has enough redundancy to allow the delivery of a non-faulty service from the erroneous internal state.
- Error recovery:
  - Substitution of an error-free state to the wrong state.
  - 2 forms of substitutions: recovery and continuation.
  - Backward error recovery :
    - Data storage (recovery data) in the course of evolution.
    - Recovery points: Data storage points.
    - System returned to a healthy state that occurred before the occurrence of error (restoring recovery points).
  - Forward error recovery :
    - Apply fixes to the current state to transform it into a healthy state from which the system can operate.

Comparison of error processing mechanisms

- Temporal and financial over-costs :
  - Error recovery: higher cost during the occurrence of error than in its absence.
  - Error compensation: shorter and constant duration, but more expensive financially.
- Backward / Forward error recovery:
  - Backward ER: no assumption on the nature of the fault (except no compromising of the recovery mechanism)
    - ⇒ Evaluation of the damage not necessary.
    - ⇒ Recovery possible after any type of error, even unforeseen errors in the design of the system.
  - Forward ER: greater interest when restoration to an earlier state is not possible (e.g. printing).

Fault treatment

- Error handling is not always enough to eliminate the error or guarantee that it will not happen again.
- Goal: To prevent one or more faults from being activated again.
- First step: diagnosis of fault (locate the causes of errors and their nature).
- Repair strategies:
  - Replacement: relies on the presence of redundant components in reserve, initially inactive, to directly replace the defective components.
  - Reconfiguration:
    - Distribution of faulty component responsibilities between the healthy system components that are running.
    - Can be static or dynamic.
- Consequence of fault treatment: Reduction of the potential of available redundancy

=> Need for manual intervention to maintain the potential.

Influence of the distribution

- Distributed system: set of processing nodes or processors (with embedded memory) interconnected by a communication network and communicating only by messages.
- The distribution of the processes and data on different processors makes it possible to structure and manage the redundancy.
- Specific aspects for fault tolerance:
  - Error and faults mainly handled by message.
  - Need to maintain the coherence of the overall state of the system, although not directly observable nor manipulable, despite the concomitance of executions.
- 2 approaches: backward recovery of distributed operations on distributed data, process replication.

Backward recovery of distributed operations

- Purpose: To move distributed data from one consistent state to another consistent state.
- Risk of cascade restoration, called "domino effect".
- Establishment of coherent recovery points to avoid the domino effect.
- Let *p1, p2, ..., pn* be a set of processes that have set recovery points at times *t1, t2, ..., tn*. The set of recovery points is consistent at a later time if:
  - Between t_i and t_j, p_i and p_j have not interacted.
  - Between t_i and t, p_i has not interacted with any process that does not belong to the considered set.
- A consistent set of recovery points is called a recovery line.

Process replication

- Data recovery approach is not adapted in the event of a processor failure (access to recovery data).
- First possible solution:
  - Stable storage server.
  - Problem: The server can be a bottleneck and therefore limit the benefits of the distribution.
- Another approach :
  - Creating multiple copies of processes on different processors.
  - Different replication approaches: active, passive, semi-active, with respectively the same operating principle as masking redundancy, passive dynamic redundancy and active dynamic redundancy.
  - Fault treatment required, in case of failure, to retrieve the initial level of redundancy.

Software fault tolerance

- Main mechanisms:
  - Exceptions mechanisms.
  - Functional diversification.
- Functional diversification:
  - Recovery Blocks.
  - N-versions programming.
  - N-self-testing programming.

Exceptions mechanisms

- Forward technique: application of corrections to the erroneous state.
- Efficient for the treatment of certain failures.
- Limitations :
  - Makes programs more difficult to maintain in languages such as C because of mixing of exception processing code and normal code.
  - Any type of probable error must be anticipated and appropriate exception treatments must be provided.
  - Useless for unanticipated faults like design faults.

Recovery blocks

- Do not need to foresee all possible faults and associated recoveries.
- Shape :

  Ensure <validity test>

  By <primary alternative>

  Else by <second alternative>

  …

  Else by <nth alternative>

  Else error;
- Validity test : condition (e. g. predicate on system variables) that must be satisfied by the system after execution of the recovery block.

Recovery blocks : case of interactive processes

- Take the domino effect into account.
- Different propositions, e. g.
  - For a set of cooperating processes, all of these processes enter into a conversation before any interaction.
  - Each process saves its state before entering a conversation.
  - One process can interact with another only if it is part of the same conversation.
  - Processes only leave the conversation after having each passed their validity test.
  - All processes restore the saved state if one of the processes in the conversation has not passed its acceptance test.
- Approach similar to transactional processes in database systems.

N-versions programming

- N-modular redundancy.
- Concurrent execution of N versions of a program (N> 2) of independent but functionally equivalent designs.
- Results compared based on a majority vote that eliminates erroneous results.
- A specific program called supervisor controls the N versions and is responsible for:
  - The call of each version,
  - Waiting for the outcome of all versions,
  - The judgment of the N results.

N-self-testing programming

- Self-testing component : addition of error detection mechanisms in the component to its functional processing capabilities.
- Parallel execution of at least two self-testing software components.
- Active dynamic redundancy case:
  - Only one component outputs the result.
  - In case of failure, another component that has not failed, is selected for the output of the result.

Case study with PILOT

- Advantages and drawbacks at the beginning
- Propositions for reinforcing dependability

Advantages and drawbacks

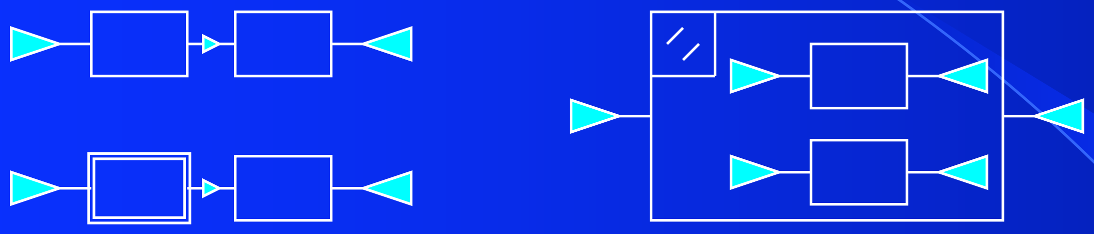
- Advantages of PILOT for dependability:
  - Language level: operational semantics available, preconditions and supervising rules, possibility to modify missions during execution.
  - Control system level : availability of interpretation algorithms, of Finite State Machines for the modules.
- Drawbacks :
  - « Lack of precisions » regarding the context of use of continuous actions.
  - Risk of incorrect plans execution.
  - Interpretation algorithms and FSM not rigorously tested nor formally checked.

Improvements

- Precision of the context of use of continuous actions and of their termination.
- Syntax oriented edition.
- Static and dynamic testing of the interpreter.
- Modeling, simulation, testing and verification of interpretation algorithms.
- Security of plans modifications during execution.

Context of use of continuous actions

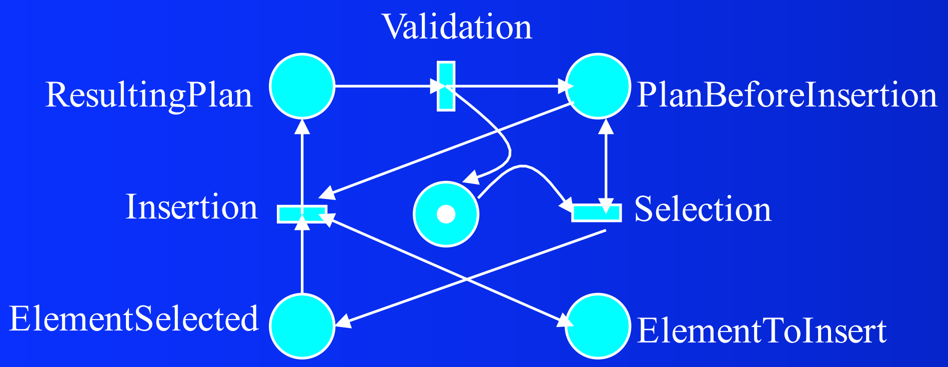
- Illustration of the problem:



- Solution proposed and implemented:
  - Notions of normal sequence and specific sequence.
  - Context of use: parallelism, preemption.
  - At least one normal sequence in a parallel or preemption structure.
  - For preemption (parallelism), stop continuous actions when one (all) normal branch (es) end.

Syntax-driven edition

- Principle: ensure syntactic validity after each operation.
- Definition of default blocks used during insert operations.
- Operation:
  - Start of construction with an empty sequence.
  - Effective consideration of an operation only if the resulting plan is syntactically correct.
- Compliance verification of the approach



- Properties checked:
  - Could an insertion lead to a syntactically incorrect plan?
  - Is there a syntactically correct plan that can not be constructed?
  - Environment used: SWI-Prolog.

Interpreter test

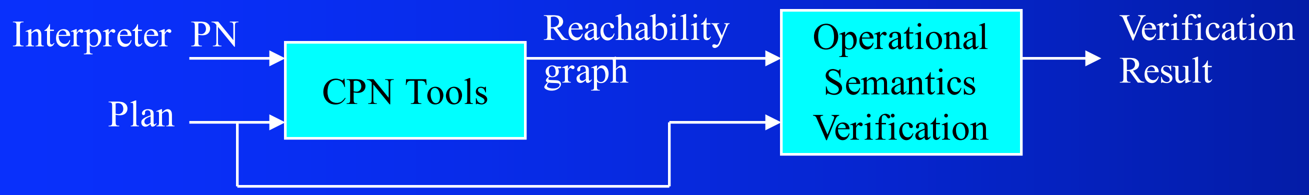
- Specificity: reactive system.
- Static test:
  - Code reading.
  - Errors detected (management of interruptions, management of the termination of continuous actions, inexperience errors, etc.).
- Dynamic test:
  - Incremental approach (empty sequence, unique primitives, combinations in length, width and depth of primitives).
  - Problems: choice of the appropriate length, width and depth; relevant combinations.
  - Solution: definition of rules for choosing a representative sample of data (stability hypotheses + feedback from the tests performed).

Modeling, simulation, testing and verification of plan interpretation algorithms

- Goal: "more rigorous" approach than the previous one.
- Approach:
  - Modeling of plan and interpretation algorithms.
  - Definition of a representative sample of the test data.
  - Simulation, test and verification of operational semantics.
  - Correction of possible errors and code regeneration from validated models.
- Formalism used: colored Petri nets.
- Reasons: graphic nature, simple representation of the concepts of algorithmic and programming, potential for property verification, availability of tools.
- Environment: CPN Tools (Ex - Design / CPN).

Modeling, simulation, testing and verification of plan interpretation algorithms (cont'd)

- Modeling of algorithms:
  - Modular approach: 1 *subnet* per algorithm; *subnets* communication through merged places.
  - Variables : creation of a colored token by instance, *life cycle* and *range* of the token reflecting those of the variable, access to *input variables* by *bidirectional arcs* contrary to the *output variables*.
  - Introduction of *runtime nodes* with the run state of the node (ready to run, running, executed).
- Verification:
  - Principle :



  - Difficulties in implementation: Translation of operational semantics into CPNML, taking into account the structure of the reachability graph, extraction of essential information.

Securing changes to plans that are running

- Aim: avoid dangerous modifications.
- Taking into account the semantics in the modification of the plan during its execution.
- Examples of litigious cases:
  - Inserting a primitive after an action or primitive that is running.
  - Deleting an active primitive.
- Specification of dynamic semantic rules based on the investigation of problematic cases.
- Implementation of the controller: creation of a separate window for the modification of the plan, need for a validation protocol.