On Fault Tolerance and Robustness in Autonomous Systems

B. Lussier R. Chatila F. Ingrand M.O. Killijian D. Powell

The dependability of autonomous systems is a particular concern, notably because of the advanced decisional mechanisms and other artificial intelligence techniques on which such systems rely. This paper sets the context of dependability and autonomy, and focuses on two non-exclusive approaches aiming to improve dependability: fault tolerance and robustness. The paper gives definitions of these approaches, and studies their relationship and applicability to autonomous systems.

INTRODUCTION

As autonomous systems play an increasing role in space exploration (*Deep Space One* in 1999, *Spirit* and *Opportunity* Martian rovers in 2004), increasing opportunities appear in other applications, such as elderly-care, tour guides or personal service. Yet with the introduction of systems capable of taking decisions without much human supervision, arises the need to ascertain their dependability, that is a justified trust that they will satisfactly perform their missions and not cause catastrophes. This paper studies the relationship between two approaches aiming at this goal from different fields: fault tolerance from the computing domain, and robustness from the robotic domain.

We describe in the first section basic concepts of dependability in computing systems, and the associated notion and mechanisms of fault tolerance. The second section introduces autonomous systems, particularly mechanisms and architectures used to enforce autonomy. Finally, we present in the third section the notion of robustness and its connection to fault tolerance, describing further robustness and fault tolerance mechanisms currently used in autonomous systems, and potential areas of application in decisional mechanisms.

I. DEPENDABILITY AND FAULT TOLERANCE

Computing systems are ubiquitous in modern society, controlling structures as critical as railroads, planes, and nuclear plants. Dependability has been for a long time a major concern in such systems, concepts and techniques are well established. This section presents basic dependability concepts in computing systems, as described in [9] and [1], and particularly the technique of fault tolerance.

A. Dependability basic concepts

The *dependability* of a computing system is its ability to deliver service that can justifiably be trusted. *Correct service* is delivered when the service implements the system *function*, that is what the system is intended to do. Three concepts further describe this notion of dependability: the attributes of, the threats to, and the means by which it is attained (Figure 1).

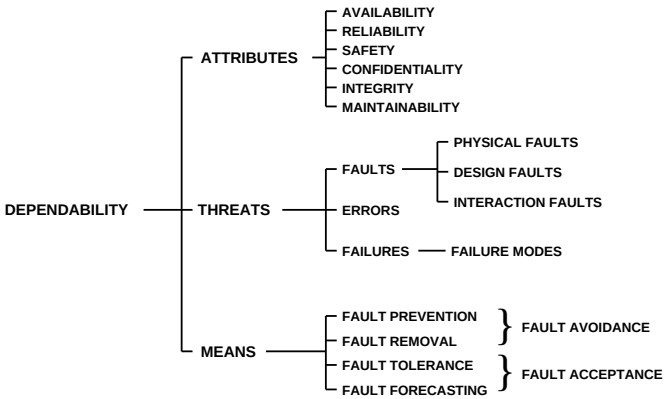

Figure 1: Dependability tree

The *attributes* of system dependability consist of:

- *availability*: the deliverance of correct service at a given time,
- *reliability*: the continuous deliverance of correct service for a period of time,
- *safety*: the absence of catastrophic consequences on the users and the environment,
- *confidentiality*: the absence of unauthorized disclosure of information,
- *integrity*: the absence of improper system state alterations,
- *maintainability*: the ability to undergo repairs and modifications.

Dependability is an integrative concept that encompasses these basic attributes; depending on the application intended for the system, different emphasis may be put on each attribute. Several other dependability attributes have been defined that are either combinations or specializations of the above.

The *threats* to a system's dependability consist of failures, errors and faults. A system *failure* is an event that occurs when the delivered service deviates from correct service. The way in which a system can fail are its *failure modes*, characterized by the severity and the symptoms of a failure. An *error* is that part of the system state that can cause a subsequent failure. An error is detected if its presence is indicated by an error message or error signal; errors that are present but not detected are latent errors. A *fault* is the adjudged or hypothesized cause of an error. A fault is active when it produces an

error; otherwise it is dormant. Faults can be characterized and regrouped into three major fault classes: *physical faults* are faults due to adverse physical phenomena, *design faults* are faults unintentionally caused by man during the development of the system, and *interaction faults* are faults resulting from the interaction with other systems, including users.

The *means* to attain a system's dependability are regrouped in four techniques:

- *fault prevention*: how to prevent the occurrence or introduction of faults,
- *fault removal*: how to reduce the number or severity of faults,
- *fault tolerance*: how to deliver correct service in the presence of faults,
- *fault forecasting*: how to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault removal can together be considered as *fault avoidance*, that is the attempt to develop a system without faults. Fault tolerance and fault forecasting embody the concept of *fault acceptance*, which attempt to estimate and reduce the consequences of the remaining faults, knowing that fault avoidance is almost inevitably imperfect. The development of a dependable computing system calls for the combined utilization of these four techniques. In the sequel, we focus on the fault tolerance technique.

B. Fault tolerance

Fault tolerance is intended to preserve the delivery of correct service in the presence of active faults. It is generally implemented by error detection and subsequent system recovery, and possibly by error containment.

Error detection originates an error signal or message within the system. There exist two classes of error detection techniques:

- *concurrent error detection*, which takes place during service delivery,
- *preemptive error detection*, which takes place while service delivery is suspended; it checks the system for latent errors and dormant faults.

Recovery transforms a system state that contains one or more errors (and possibly faults) into a state that can be activated again without detected errors and faults. Recovery consists of error handling and fault handling. *Error handling* eliminates errors from the system state. It can take three forms:

- *rollback*, where the state transformation consists of returning the system back to a saved state that existed prior to error detection; that saved state is called a checkpoint,
- *rollforward*, where the state without detected errors is a new state,
- *compensation*, where the erroneous state contains enough redundancy to enable error elimination.

Fault handling prevents a fault from being activated again in four steps: fault diagnosis, fault isolation, system reconfiguration, and system reinitialization.

Error containment restrains the propagation of an error within a containment area, thus preventing the failure of other system components.

C. Common fault tolerance mechanisms

In the following paragraphs, we consider techniques applicable for tolerating physical faults, design faults and interaction faults.

1) Tolerance of physical faults: The detection of errors induced by physical faults is commonly attained by the following mechanisms:

- *duplication and comparison* detects errors by comparing the output of two independent and functionally identical units, under the assumption that the same fault will not affect the two units simultaneously,
- *timing and execution checks* are usually implemented by "watchdog" timers; they can be used to detect a timing error or to monitor the activity of a component,
- *reasonableness checks* use specific hardware or software to verify value invariants (invalid memory address, invalid input or output),
- *error detecting codes* introduce redundancy in the information representation to detect possible errors in that representation.

Following an error detection, system recovery is mainly attained by error handling:

- Rollback error recovery is the most popular form of recovery: hardware or software mechanisms periodically save the system state so as to be able to return the system to a previous stable state. They are however time and resource consuming, as well as ill-adapted to hard real-time deadlines.
- Rollforward error recovery consists in searching for a new state acceptable for the system from which it will be able to resume operation (possibly in a degraded mode). Reinitialization of the system and exception handling are possible approaches for rollforward error recovery.
- Error compensation requires sufficient redundancy in the system state so that, despite errors, it can be transformed into an error-free state. Error compensation can be launched following error detection (*detection and compensation*), or can be systematic (*fault masking*). The use of self-checking components in active redundancy is an example of detection and compensation: in case of failure of one of them, it is disconnected without disturbing the other components. Fault masking can be implemented by majority voting: computation is carried out by three or more identical or similar components whose outputs are voted.

2) Tolerance of design faults: The same principles as for physical faults apply to design faults, except for the type of redundancy used for detection and recovery.To achieve (or at least aim for) independence with respect to design faults, redundant elements must be of dissimilar or diversified design.

Tolerance of design faults addresses two major concerns: to limit the consequences of task failure on the rest of the system or its environment, and to maintain service continuity.

- In the former case, one tries to detect an erroneous task as early as possible and to halt it to prevent propagation of errors; examples are an approach called *"fail-fast"*, and the notion of *safety-bag* [8]. A safety-bag intercepts

the actions requested by the users or components of the system, rejecting those that invalidate its set of safety rules.
- In the latter case, one makes use of *design diversity*, which relies on several copies of a component (called *variants*), designed and produced separately from the same specification. A *decision maker* is also required, which aims to produce an error-free result from those produced by execution of the variants. There exist three basic approaches for design fault tolerance using design diversity: *recovery blocks* [15], *N-version programming* [4], and *N-self-checking programming* [10].

3) Tolerance of interaction faults: Distinction is made between accidental interaction faults, such as an operator mistake, and intentionally malicious interaction faults.

- *Accidental interaction faults* can be tolerated both by error processing and treatment of error causes. *Error processing* is achieved through error detection using *task models* or diversified sources of information, and error recovery such as the replacement of the erroneous action, automatically or after the user's approval. The *treatment of error causes* aims to establish the diagnosis of error causes and then to design solutions to act on these causes.
- *Intentionally malicious interaction faults* may be caused by intruders external to the system attempting to penetrate the system, registered users trying to extend their privileges, or privileged users abusing their privileges. Tolerance of such intrusions aims to protect the availability, integrity and confidentiality of the information, using techniques such as replication, fragmentation-redundancy-scattering and cryptography, or to act on the intrusions themselves via detection and recovery.

II. Autonomous Systems

Artificial Intelligence (AI) originally aimed to develop machines with reasoning capabilities similar to or better than human. Although far from such a goal, techniques and mechanisms have been successfully introduced in various domains, such as autonomous systems. This section proposes a definition of autonomous systems, and describes some decisional mechanisms and architecture principles used to support autonomy. It introduces in particular the LAAS architecture.

A. Definition of autonomy

Common definitions of autonomy ("self-independence", "ability to self-manage") are not adapted to characterize the systems that we are interested in, so we attempt to formulate a functional definition of an autonomous system:

> an *autonomous system* implements selection and execution of actions to be taken through one or more of the following AI functionalities: planning, execution control, situation recognition, diagnosis, and may also incorporate learning mechanisms.

- *Planning* consists in choosing and organizing actions to be taken, according to their estimated results, in order to achieve one or more objectives.
- *Execution control* acts as coordinator and supervisor of the execution of plans. It mainly decomposes high-level actions into sequences of behaviors or simpler tasks, and controls their execution in order to react to possible failures due to the system itself or to its environment.
- *Situation recognition* aims to identify the circumstances confronting the system that are likely to affect its behavior, generally the system state and that of its environment. Situation recognition usually rely on pas event observations in order to draw conclusions on the current situation, and eventually on the intentions of other agents involved.
- *Diagnosis* identifies an erroneous system state, generally after an error detection. Although diagnosis may be viewed as a specific application of situation recognition, their approaches and techniques are distinct.
- *Learning* seeks to improve the system capabilities by using information related to preceding executions. The learning process does not make decisions, but typically develops some models that can be used by other functionalities.

B. AI approaches for autonomy's functions

Several approaches developed in the AI field may be used to implement the functions listed above; they are commonly referred to as *deliberative* approaches, in opposition to the *reactive* mechanisms of classical automation. Deliberative mechanisms are either executed *off-line* before activation of the system, or *online* concurring to its execution. The most common approaches used to implement autonomy are [16]:

- *States space search* manipulates a graph which nodes are the states of the system, and transitions are events and actions leading from one state to another. The search consists in examining possible sequences of actions, then choosing the most appropriated one to achieve some given goal. This approach is mainly used for planning, situation recognition and diagnosis.
- *Constraint satisfaction techniques* seek to resolve a Constraint Satisfaction Problem (CSP), defined by a set of variables and constraints upon them. A solution of the problem is found when all variables have value ranges satisfying the constraints. *Temporal constraint planning* is achieved through extension of a CSP to include a temporal dimension.
- *Markov Decision Processes* (MDP) are sequential decisional problems on the actions (deterministic or stochastic) and states of the system. A given solution has the form of a *policy* which gives the best action for the system to take in each possible state. The MDP technique supposes that the system knows exactly in which state it is; an extension called *Partially Observable Markov Decision Processes* (POMDP) addresses this limitation. MDPs are mainly used for planning and learning.
- A *Bayesian network* is an oriented acyclic graph representing the state variables of the system, and their influences on each other; such a network is used to manipulate probabilities and uncertainties. Dynamic Bayesian networks extend this approach with a discrete temporal

dimension. Bayesian networks are mainly used for diagnosis and learning.

- *Hidden Markov Models* (HMM) are discrete temporal and probabilistic models of the system state; this state is not supposed to be directly observable but produces observable outputs. With some manipulations, an equivalent Bayesian network can be found for each HMM (and vice versa); each representation is more appropriated for different algorithms. HMMs are mainly used for diagnosis and learning.
- A *neural network* is a compound of units, also called artificial neurons, which defines a complex non-linear function. The units are linked by directed weighted connections, and organized in different layers. Neural networks can be cyclic or acyclic, and are mainly used for learning.

Several other approaches have also been implemented for autonomy, such as genetic algorithms, contradictory search, and expert systems; in our opinion, these approaches are less viable than those mentioned previously.

C. Types of architecture for autonomous systems

We describe in the following paragraphs three types of architecture most popular for the implementation of autonomous systems: the *subsumption* type, the *"three layer"* type, and the *multi-agent* type. The subsumption type and the "three layer" type are discussed in [6].

1) Subsumption type of architecture: This type of architecture [3] is "behavior-based": it rejects the need for a symbolic representation of the system and its environment, proposing instead layers of progressively more complex task-specific control programs (called *behaviors*) on top of each other. At each execution cycle, each behavior may generate an output; the different outputs are then combined into the task to be executed, for example: by executing only the behavior with the utmost priority, or combining all of the outputs (Figure 2).

In our opinion, the lack of symbolic representation in the subsumption type prevents an efficient use in complex environments and situations. The subsumption type is currently used on toy robots but not for complex critical systems.

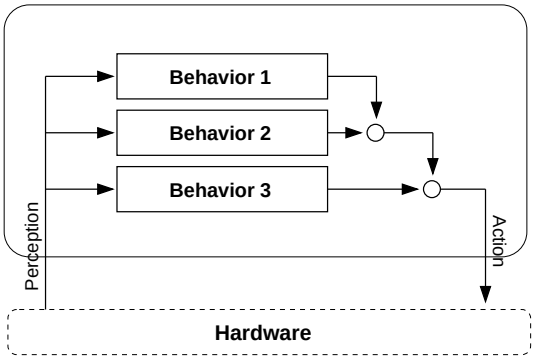

Figure 2: Subsumption type of architecture

2) "Three layer" type of architecture: The "three layer" type of architecture consists of several hierarchical components (or *layers*), considering different level of abstraction for the symbolic representation of the system and its environment. There are typically three layers (hence the name), but some architectures focus only on some of these layers, or regroup two layers into one. The three classical layers are the decisional layer, the executive layer, and the functional layer (Figure 3):

- The *decisional layer* is situated at the top level of abstraction. It carries out the most complex decisions, producing the plan required to achieve the objectives of the system, and taking into consideration problems or errors raised by the executive layer.
- The *executive layer* selects sequences of elementary functions needed to execute the high level plan of the decisional layer. It also reacts quickly to errors or failed tasks, raising the problem to the decisional layer when unable to solve it.
- The *functional layer* offers an interface between the higher layers and the hardware, combining sensors and actuators into elementary functions controlled by the executive layer. It does not possess a symbolic representation of the system, but must guarantee real-time constraints to control the hardware efficiently.

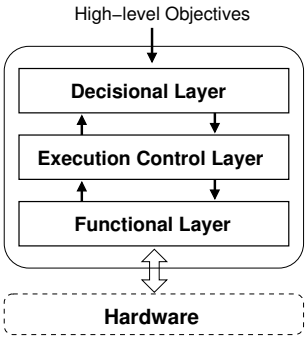

Figure 3: "Three layer" type of architecture

The "three layer" type of architecture is the one most often used to develop complex autonomous systems. It has been implemented in the RAX architecture during the NASA's Deep Space One mission [13], and is currently used in the CLARAty architecture [18] and the LAAS architecture (described in the next section).

3) Multi-agent type of architecture: The multi-agent type of architecture considers a group of autonomous systems (or *agents*). These agents may be homogeneous or heterogeneous, evolve in the same environment, and interact with one another in order to achieve common or self-centered objectives.

Amongst other works, the IDEA architecture [12] considers an autonomous system as a group of multiple agents, each possessing deliberative mechanisms and the same symbolic representation (Figure 4).

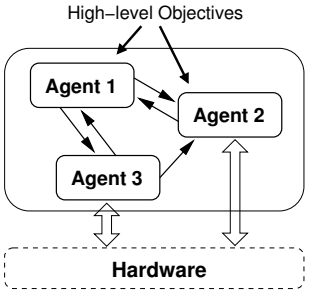

Figure 4: Multi-agent type of architecture: IDEA principle

D. The LAAS architecture

The LAAS architecture [2] [14] [11] possesses three layers (Figure 5):

- The *decisional layer* encompasses both decisional and executive layers of the typical "three layers" based architecture. A temporal executive planner called IxTeT-eXeC produces high level plans by constraint planning. A procedural executive called OpenPRS controls plan execution and decomposes the high-level actions into sequences of simpler tasks.
- The *request control level* is implemented by the Request and Resource Checker (R2C) component. It checks the validity of requests produced by OpenPRS according to the current system state and a set of conditions defined during the development of the system. Requests that invalidate the rules are then rejected.
- The *functional level* is composed of a hierarchy of software components (called *modules*) offering services to specific hardware or software resources (sensors, actuators, data...). The modules' generic structure is automatically generated via the GenoM tool.

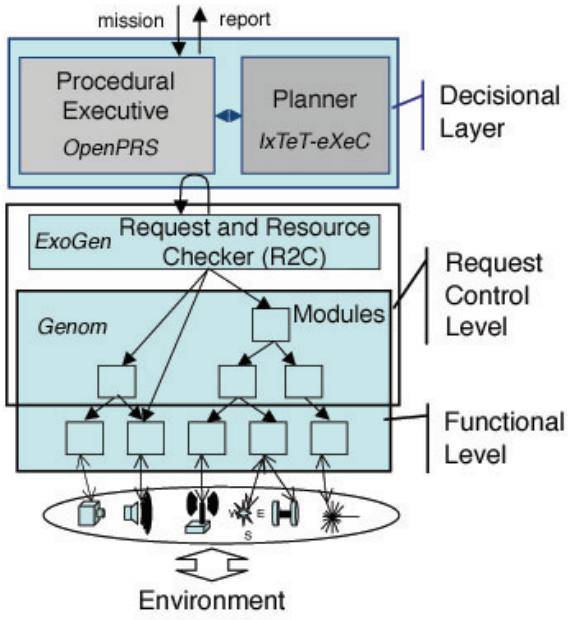

Figure 5: LAAS architecture

This architecture has been used on several autonomous robots, such as Rackham: a tour guide currently exhibited at the Toulouse *Space City Museum*.

III. Robustness and fault tolerance

The notion of robustness appeared in the robotic field as an answer to the large variability of execution context due to an open environment. To some extents, it may be considered similar to fault tolerance as both techniques seek to cope with adverse situations that may arise during system operation. This section introduces a definition of robustness and its connection to fault tolerance, and presents examples of both approaches in autonomous systems. It finally discusses their applications to decisional mechanisms.

A. Definition of robustness

The term *robustness* is frequently used in the scientific community, although in rather a vague way (a common definition of *robust* is "strong and healthy"). The *Santa Fe Institute* has recorded seventeen definitions of robustness from diverse scientific domains [5]. These various definitions may be interpreted as the deliverance of *correct service* despite possibly adverse situations, and then classified into four types of "situation tolerance" (Figure 6):

1) tolerance of *any situation*: deliverance of correct service in both adverse and nominal situations,
2) tolerance of *adverse situations*: deliverance of correct service in non-nominal situations,
3) tolerance of *explicitly-specified adverse situations*: deliverance of correct service in adverse situations mentioned in the system's specifications,
4) extra-tolerance, or tolerance of *unexpected adverse situations*: deliverance of correct service in adverse situations over and above those mentioned in the system's specifications.

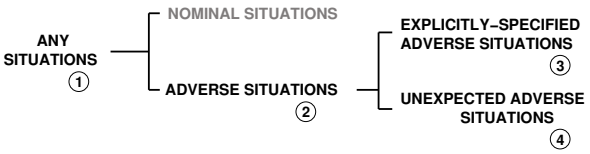

Figure 6: Types of "situation tolerance"

Strictly speaking, robustness is thus a superset of fault tolerance since the latter focuses on faults whereas the former considers adverse situations in general. However, a useful distinction between the two can be made by restricting the use of the term robustness to the tolerance of adverse situations *not* due to faults. We therefore adopt the following definitions in the context of autonomous systems (Figure 7):

- *robustness* is the delivery of a correct service in implicitly-defined adverse situations arising due to an uncertain system environment (such as an unexpected obstacle or a change in lightning condition affecting sensors),

- *fault tolerance* is the delivery of a correct service despite faults affecting system resources (such as a flat tire, a sensor malfunction or a software fault).

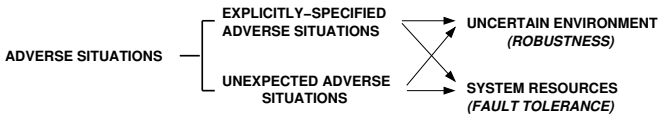

Figure 7: Robustness vs. fault tolerance

In practice, it is not always easy to distinguish situations due to an uncertain environment from situations due to faults affecting system resources, especially input/output devices, but this goes beyond the scope of this paper.

B. Robustness mechanisms in autonomous systems

Robustness in robotic systems is typically achieved either by functional redundancy, aimed at compensating the limitations of hardware components or software algorithms (such as a combined use of camera, laser sensor and bumper to detect obstacles, or complementary localization algorithms), or by using uncertainties management, aimed at compensating environment uncertainties for control and observation (such as fuzzy logic or Kalman filtering).

Autonomous functions mainly improve the robustness of a system through the use of decisional mechanism and recovery: the ability to act and react according to the current environment and system state.

1) Planning: Planning improves system robustness in general since it allows the autonomous decision-making that is a prerequisite for operation in an uncertain environment. Moreover, *least commitment planning* allows flexibility with respect to action ordering, temporal deadlines and resource consumption.

Planning can itself be made robust through replanning and plan repair:

- *replanning* can be activated when a plan has failed; it consists in stopping the plan execution, eventually positioning the system in a safe state, and developing a new plan from the current situation and the remaining objectives,
- *plan repair* can be activated when part of a plan has failed, before replanning occurs; it consists in developing a new plan from the failed one by backtracking and eliminating the failed and impossible actions.

2) Execution control: Execution control improves robustness by:

- managing the flexibilities left in the plan,
- detecting possible problems in the plan execution concerning failed tasks or exceptions raised by the lower layers,
- allowing recovery of a failed task by selecting and executing an alternative one (in a similar way as recovery block tolerance of software design faults); it reports a plan failure when all alternative tasks have failed.

C. Fault tolerance mechanisms in autonomous systems

Faults affecting system resources can be divided in faults affecting non-computational resources (mechanical or electrical components such as tires, joints, sensors and actuators), and faults affecting computational resources (such as memories, CPU and software). The latter refers specifically to mechanisms detailed in subsection *Fault tolerance* 1.B.

1) Faults affecting non-computational system resources: Faults affecting non-computational system resources are mainly treated by the function of *diagnosis* and reconfiguration. Diagnosis is activated after an error detection and identifies the cause of an erroneous state by reasoning based on fault and error models. Reconfiguration may correspond either to replacement of the failed component, or relocation of its supported functions to other components by functional redundancy.

2) Faults affecting computational system resources: The following paragraphs describes the computational fault tolerance mechanisms currently identified in autonomous systems.

Error detection is mainly implemented through:

- timing and execution checks; watchdogs check the liveness of critical functions in the RoboX autonomous system [17],
- reasonableness checks; they are either implemented by checking a value with invariants (such as the maximum robot speed in RoboX), or by detecting incoherences between the current system state (characterized by sensor outputs) and a mathematical model (typically used to activate diagnosis, as in the RAX architecture).
- safety-bag checks; a set of safety properties is checked by the R2C component in the LAAS architecture.

Error recovery is implemented through:

- positioning in a safe state; this can occur during replanning (LAAS and RAX architectures), or after failure of a critical component (RoboX and Care-O-Bot [7]),
- software reconfiguration; this can be implemented by switching between control modes, or by applying a software patch.

Error containment is implemented through dedicated CPUs: RoboX runs one CPU for critical functions (obstacle avoidance, navigation, localization) and one CPU for interaction functions (face tracking, speech out).

D. Robustness and fault tolerance for decisional mechanisms

Decisional mechanisms can be characterized by three properties:

- *soundness*: inferred conclusions are “true” under the system assumptions,
- *completeness*: a true conclusion will eventually be inferred,
- *tractability*: the conclusion can be inferred in polynomial time and space.

Although soundness and completeness are verified for some decisional approaches (usually the case for state space search or constraint planning, but not for neural networks), tractability

is often impossible due to NP-hard complexity or semi-decidability of the considered problems. Therefore soundness and completeness are sometimes sacrificed for efficiency, for example with the application of heuristics. Robustness and fault tolerance can compensate to a point these drawbacks by improving reliability and safety of the decisional mechanisms.

Robustness techniques can aim to improve the reliability of an autonomous system through decisional recovery. They can treat specifically adverse situations affecting the acceptability of decisions, that is unexpected changes of situation due to environment and system dynamics, and incorrect knowledge due to lack of observability on the environment.

Fault tolerance techniques can aim to improve the reliability of an autonomous system through error detection and the use of alternate procedures. They can treat specifically: incorrect or incomplete knowledge due to system deficiencies (faults), design compromises in favor of efficiency of the decision procedure, faults in design or implementation of the decision procedure.

Fault tolerance techniques can also aim to improve the safety of an autonomous system through error detection and positioning in a safe state.

CONCLUSION

In this paper, we have briefly discussed several notions:

- *dependability*, that is the ability of a system to deliver service that can justifiably be trusted; this notion further encompasses attributes (such as safety and reliability), threats (faults, errors and failures) and means (fault prevention, fault elimination, fault tolerance and fault forecasting),
- *autonomy*, that is the ability to select and execute actions in order to achieve stated objectives, using AI functionalities: planning, execution control, situation recognition, diagnosis and learning,
- *robustness* and *fault tolerance*, approaches respectively from the robotic domain and the computing systems domain.

Fault tolerance and robustness both characterize the ability to deliver correct service, but we have distinguished them according to the type of adverse situations that they try to confront:

- fault tolerance characterizes tolerance towards faults affecting the system resources,
- robustness characterizes tolerance towards uncertainties of the environment.

We have introduced the main mechanisms used by these two techniques:

- fault tolerance consists mainly in *error detection* and *error recovery* through the use of redundancy,
- robustness consists mainly in *functional diversity*, *compensation of uncertainties*, and *decisional recovery*.

We have identified areas of application for both techniques in decisional mechanisms:

- fault tolerance can be implemented to improve on one hand safety, and on the other hand reliability towards incorrect or incomplete knowledge due to system deficiencies, design compromise for efficiency, and faults in the decision procedure,
- robustness can be implemented to improve reliability towards unexpected changes of situation due to the environment or system dynamics, and incorrect or incomplete knowledge due to lack of observability.

REFERENCES

[1] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Verissimo, M. Waidner, and A. Wespi. Conceptual Model and Architecture of Maftia. Technical Report 03011, LAAS-CNRS, 2003.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An Architecture for Autonomy. *The International Journal of Robotics Research*, 17(4):315–337, April 1998.

[3] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Magazine on Robotics & Automation*, 2(1):14–23, March 1986.

[4] L. Chen and A. Avizienis. N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th International Symposium on Fault Tolerant Computing (FTCS-8)*, pages 3–9, Toulouse, France, 1978.

[5] Santa Fe Institute document reference RS-2001-009, Posted 10-22-01. http://discuss.santafe.edu/robustness.

[6] E. Gat. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. P. Bonnasso, and R. Murphy editors, MIT/AAAI Press, pages 195-210, 1997.

[7] B. Graf, M. Hans, and R. D. Schraft. Mobile Robot Assistants - Issues for Dependable Operation in Direct Cooperation With Humans. *IEEE Magazine on Robotics & Automation*, 11(2):67–77, 2004.

[8] P. Klein. The Safety-Bag Expert System in the Electronic Railway Interlocking System Elektra. *Expert Systems with Apllications*, 3(4):499–506, 1991.

[9] J. C. Laprie, J. Arlat, J. P. Blanquart, A. Costes, Y. Crouzet, Y. Deswarte, J. C. Fabre, H. Guillermain, M. Kaâniche, K. Kanoun, C. Mazet, D. Powell, C. Rabéjac, and P. Thévenod. *Dependability Handbook (2nd edition)*. Cépaduès - Éditions, 1996. (ISBN 2-85428-341-4) (in French).

[10] J. C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and Analysis of Hardware-and-Software Fault-Tolerant Architecture. *IEEE Computer*, 23(7):39–51, 1990.

[11] S. Lemai and F. Ingrand. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proceedings of AAAI-04*, pages 617–622, San Jose, California, July 25-29 2004.

[12] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA : Planning at the Core of Autonomous Reactive Agents. In *AIPS 2002 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 22 2002. http://citeseer.nj.nec.com/593897.html.

[13] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[14] F. Py and F. Ingrand. Real-Time Execution Control for Autonomous Systems. In *Proceedings of the 2nd European Congress ERTS, Embedded Real Time Software*, Toulouse, France, January 21-23 2004.

[15] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[16] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach (2nd edition)*. Prentice Hall, 2002. (ISBN 0-13-790395-2).

[17] N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. O. Arras, and R. Siegwart. Designing a Secure and Robust Mobile Interacting Robot for the Long Term. In *Proceedings of the 2003 IEEE International Conference on Robotics & Automation*, pages 4246–4251, Taipei, Taiwan, September 14-19 2003.

[18] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. CLARAty : Coupled Layer Architecture for Robotic Autonomy. Technical Report D-19975, NASA - Jet Propulsion Laboratory, 2000.