Dependability Issues in AI-Based Autonomous Systems for Space Applications

David Powell, Pascale Thévenod-Fosse
{david.powell, pascale.thevenod-fosse}@laas.fr

LAAS-CNRS, 7 avenue du Colonel Roche
31077 Toulouse Cedex 4, France

Abstract. *We present a literature study of dependability issues raised by the use of artificial intelligence (AI) techniques in critical systems. Hazards specific to such approaches are identified. Drawing on two case studies of AI-based critical systems, the paper concludes by a series of recommendations.*

1 Introduction

Autonomy is a desirable feature of future uninhabited spacecraft so as to reduce the need for ground intervention. First, maintaining readiness for such intervention is costly. Second, for some missions, such as deep-space probes, intervention may be, at best, slow due to propagation delays and low bandwidth, or even impossible, due to occlusion when orbiting remote planets.

Autonomy can be achieved partially by classic automation, but AI-based approaches enabling autonomous decision-making are of particular interest. However, the use of such approaches on board costly spacecraft raises the question of their dependability. This paper reports a preliminary literature study of dependability issues of using AI-based approaches in critical systems [Lécubin *et al.* 2001].

We start by identifying some key concepts about AI-based systems (mainly by abstracting from [Robertson & Fox 2000]) and then identify specific potential hazards. We then present two case studies of AI-based systems used in critical applications, where dependability is evidently a major issue. The first of these is taken from the area of medical care, the second from the space domain.

2 Key Concepts

The central concept in most AI-based systems is that the domain-specific knowledge that helps the system to solve a problem can be represented separately from the mechanisms used to draw inferences from that knowledge. The advantages of this separation are that:

- Inference mechanisms can be re-used with different knowledge bases.
- Knowledge bases can be maintained without adaptation of inference mechanisms.
- By being represented separately and, ideally, declaratively, the domain-specific knowledge is more readily understood by specialists of the problem domain.

However, it is not possible to use any inference mechanism with any knowledge base. Each inference mechanism makes assumptions about the formal language used in knowledge representation. Thus, even if a knowledge base is declarative, it must be expressed in a style that is appropriate to the inference mechanism for it to be used effectively. In practice, knowledge representation and inference are thus often more tightly linked than one would expect from theory.

2.1 Knowledge Representation

Production rules constitute one of the earliest forms of knowledge representation. These use **if** *a* **then** *b* clauses to represent knowledge. During inference, if the premise *a* can be established, the consequence *b* can be added to the working memory, which represents the current system state. Since the applicability of a rule depends on the current system state, this sort of rule does not have a straightforward declarative interpretation. Moreover, the order of application of rules is important since each rule may alter the system state. Some sort of control information, such as rule precedence, is thus usually built into the knowledge base.

This intertwining of declarative and procedural knowledge can easily give rise to inconsistencies that are difficult to detect in large knowledge bases.

In theory, this sort of problem can be avoided by using representation schemes based on logic. Such representation schemes are more susceptible to a declarative interpretation but, sooner or later, some procedural problems must be faced when performing inference (i.e., we must have a procedure for choosing axioms of the logic when solving a problem). Nevertheless, the value of logic is that a large part of the design may be separated from the choice of procedure.

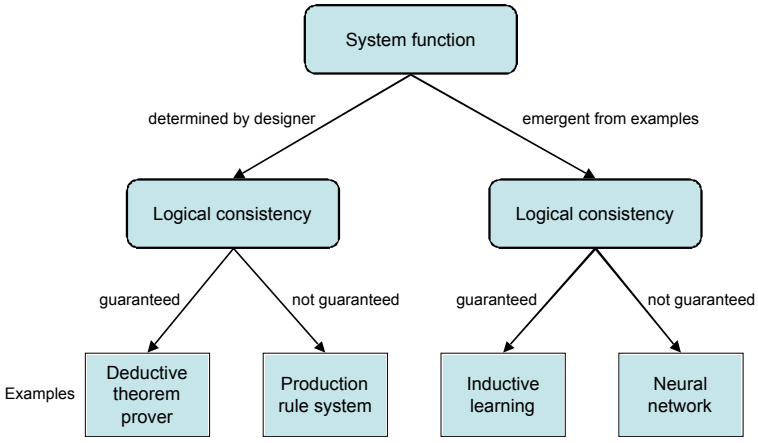

Fig. 1 — A high-level classification of AI-based systems
[Robertson & Fox 2000]

In *model-based* systems, the knowledge base contains descriptions of a system from which solutions to a problem can be inferred. In *case-based* systems, the knowledge base contains examples of previous problem-solution pairs, which are used as paradigms for solving new problems. *Frame-based* systems often use a combination of representational styles from production rules and logic, with ideas of modularity and inheritance from object-oriented design. However, frame-based systems are not used much nowadays.

Whatever the representation scheme, it is often necessary to represent *uncertainty*, either in the knowledge itself or in the inferences that are drawn from it. This normally involves labeling elements of knowledge to indicate the form and level of the associated uncertainty, and then propagating these measures of uncertainty during inference.

2.2 Inference

There is no consensus on how to classify the many different inference mechanisms that have been built. Furthermore, many "hybrid" systems use more than one style of inference to deal with different aspects of the application. Nevertheless, there are some broad distinctions that may be made from the viewpoint of dependability and safety (**Fig. 1**).

At the highest level, AI-based systems can be divided according to the extent to which the system designers determine the function of the system.

If the designers define both the knowledge base and the inference mechanism, then it should be possible to predict the behavior of the resulting system. A safety argument may then be made by relating the predicted behavior to the code. Furthermore, if the inference mechanism is based on a logic that can be guaranteed to be logically consistent (e.g., systems based on a deductive theorem prover), it may be possible to formally prove properties of the system. If not (e.g., in a production rule system), it may be possible to base a safety argument on the code structure, as in conventional software engineering.

Alternatively, some systems are designed to determine their function by presenting them with examples, either during training sessions or incrementally during operation. In this case, it is not possible to relate arguments about predicted behavior to the code. However, for systems that use formal logic in an inductive style, it may be possible to use formal proof to increase confidence in the dependability of the implemented function, conditioned on some assumptions about the underlying examples.

As in traditional software engineering, testing is necessary to obtain confidence in the implemented code. In extreme cases (e.g., neural networks), testing is the only technique on which a safety argument can be based, especially concerning robustness with respect

to new inputs for which the system has not been trained.

2.3 Specific Mechanisms

In this section, we describe briefly some specific mechanisms that are used in AI-based systems.

2.3.1 Rule-Based Systems

As already discussed in Section 2.1, production rules were one of the earliest forms of knowledge representation. Such rule-based systems, also known as expert systems, are probably the most widespread form of artificial intelligence. They have been studied for more than forty years, and have had many real-world applications.

Several formal methods for anomaly detection have been developed for rule-based systems. The use of logic to represent the rules enables the design of automatic systems to detect incompleteness, conflicts, redundancies, etc. Three categories [Marcos *et al.* 1995] of techniques can be identified:

- Techniques for checking the whole rule-base.
- Techniques focusing on specific properties of rules (e.g., both X and not(X) in a premise, meta-rules, etc.).
- Incremental-checking techniques that allow a low-cost verification whenever a modification occurs.

Among all the verifications to be done to ensure correctness of the rule-base, several are easy to check and can efficiently increase consistency. This is the case of:

- *Redundancy*: two rules are redundant if they have the same premises and the same effects.
- *Conflicts*: two rules are in conflict if they have the same premises but contradictory effects.
- *Subsumptions*: a rule subsumes another if they have both the same effect but the premise of the latter is more restrictive than the former.
- *Unnecessary conditions*: a rule contains unnecessary conditions if two rules have the same premises and the same effects except for one condition that is contradictory in the two rules (e.g., X in one rule premise and not(X) in the other).

This kind of system is usually designed and implemented using a *rule engine*, which provides a framework for writing rules. Currently developed rule engines are fairly easy to interface with other code modules. These techniques do not guarantee that the developed system will behave as expected, even coupled with expert domain knowledge. Nevertheless, they enable a rapid and efficient detection of problems in the design and construction of sets of rules, improving the overall dependability and safety of the future system.

However, the challenging problem of rule-based systems (and other autonomous decisional systems) is that they are required to act "sensibly" in unanticipated situations [Johnson *et al.*, 1999]. Practically the only way to increase confidence in their ability to achieve this is to carry out extensive simulation testing.

2.3.2 Probabilistic Networks

Probabilistic networks provide a mechanism for propagating measures of confidence when making inferences. Nodes of the network represent propositions with which different levels of confidence may be associated. Links between nodes designate pairs of propositions for which confidence in one proposition affects confidence in the other proposition. By fixing the confidence levels for some of the nodes to suit the problem to be solved (e.g., by setting to the highest confidence level the confidence associated with propositions that are known to be true) and then propagating the corresponding change in confidence through the network, new confidence levels can be examined for the propositions that are of interest.

It is possible to make mathematical arguments of correctness when the confidence propagation method has a logical foundation, for instance that of Bayesian probability theory. If not (e.g., networks based on Monte Carlo simulation), then the only recourse is to argue empirically through statistical sampling from probability distributions obtained by multiple runs of the network.

2.3.3 Argumentation Systems

AI-based systems are often applied in domains where there is uncertainty in the conclusions that may be drawn from inference. Even if the steps taken during inference are based on logical proof, they may be more appropriately interpreted in the real world as arguments for particular courses of action. This observation has motivated research into the use of argument structures to control the acceptability of inference, rather than using summative measures of confidence.

Argumentation systems are suited to applications (e.g., drug prescription) that require flexibility in weighing up evidence for alternative courses of action (e.g., therapies) and for which quantitative measures of confidence (e.g., probabilities) do not give sufficient information for assessing alternative strategies in terms comprehensible to human experts.

2.3.4 Fuzzy Logic

Fuzzy logic provides another way of propagating uncertainty during inference. Mathematical functions may be used to translate an observed measurement into several fuzzy truth-values with associated probabilistic measures of confidence (e.g., a measure of 25°C might be mapped to "warm" with probability 0.7 and "hot" with probability 0.5).

The main difficulty with fuzzy logic is that the probabilistic measures of confidence may not behave as in conventional probability theory (for instance, the probabilities of being warm or hot in the example above sum to more than 1). Although confidence combination functions can be defined that are well behaved in such circumstances and often suitable for solving some engineering problems (e.g., in control engineering), it is nevertheless difficult to understand the meaning of such uncertainty measures.

2.3.5 Model-Based Reasoning and Qualitative Simulation

In model-based reasoning, expert knowledge is expressed in terms of a model of a class of problems rather than being represented directly. For example, when building a knowledge-based system for fault diagnosis in electronic circuits, the experts' diagnostic procedures may be represented directly or in terms of a computational model of a family of electronic circuits. Such an approach has the advantages of being able to relate inference more explicitly to the object of study, and of being often the most natural way of expressing expertise.

Qualitative simulation is one of the most effective uses of model-based reasoning. Qualitative simulation uses qualitative rather than precise values for state variables. It can be used to predict large-scale behavior of systems and to derive there from information that was not immediately obvious. An advantage of such a qualitative approach is that it is normally possible to have a finite set of model variables, which may take values from only a small set of possible values, so an exhaustive exploration of the state space may be possible. The disadvantage is that increasing the granularity results in a loss of precision, which may cause important behaviors to be overlooked.

2.3.6 Planning

The objective of a planning system is to decide on the appropriate sequences of actions to achieve a given goal from a given situation. The distinguishing characteristic of planning systems with respect to other AI-based systems is their temporal aspect, i.e., they are concerned with sequences or partial orderings of actions, or even with explicit physical timing. Action timing and precedence constraints may be expressed in various ways, ranging from simple linear sequences to non-linear interval-based representations. Some planning systems base knowledge representation and reasoning on a temporal logic. Recently, a Planning Domain Definition Language (PDDL) has been defined to encourage empirical evaluation of planner performance and development of standard sets of problems, all in comparable notations [McDermott 1998][Fox & Long 2001].

2.3.7 Inductive Machine Learning

Since expert knowledge is often difficult to acquire simply by observing or questioning experts, an important field of research has been that of acquiring problem-solving knowledge through induction, by generalizing from observed examples of behavior. To be

effective, inductive machine learning requires an application domain capable of providing both positive and negative training examples, i.e., examples that establish what the problem-solving rules should cover and what they should avoid. The major disadvantage of inductive machine learning is sensitivity to this training set, compounded by the fact that the derived rules may not be readily understandable by human experts and thus difficult to validate.

2.3.8 Case-Based Reasoning

The idea of case-based reasoning is to automate a style of human problem-solving in which recollections of past problem-solving situations are reinterpreted to suggest solutions to similar problems, instead of trying to follow chains of reasoning from assumptions to conclusions. Past problems and their solutions are collected into a library of formal descriptions. Features of a new problem described in the same formalism are compared to features of past problems by means of a matching algorithm. The closest matching past problem and its solution are then retrieved and automatically adapted to the new problem. If appropriate, the solution may be refined and the new problem-solution pair added to the case library. Case-based reasoning systems are thus learning systems. The effectiveness of this approach depends on the matching algorithm, which may range from simple keyword matching to complex systems of structural comparison and inference. The choice of algorithm is domain-dependent and often heuristic. It seems that only empirical testing can raise confidence about the dependability of the matcher.

2.3.9 Artificial Neural Networks

Artificial Neural Networks (ANNs) aim to model the processing of information by the human brain. Only the most basic elements of the brain are represented, using a basic structure called a neuron.

A group of inputs are weighted, processed by two functions (a simple one and a transfer function), and finally output. The basic function may currently be a sum, a maximum extraction, an average value, an OR, an AND, etc. whereas the transfer function may be a sigmoid, a hyperbolic tangent, etc. Sets of replicated neurones are then organised in layers, of which three different kinds are traditionally distinguished:

- The Input Layer, where each neurone is connected to real-world sensors.
- The Hidden Layer (there may be several) having inputs coming directly from the outputs of other neurones and outputs going directly to inputs of other neurones.
- The Output Layer, giving its outputs to the real world or to another subsystem.

Knowledge in an artificial neural network (ANN) system is therefore not encoded symbolically but is represented indirectly through the states of the neurons and of the interconnections between them. Neurons normally have simple states of activation or deactivation dependent on adjustable weights. Before deployment, a neural network must be trained on a series of positive and negative examples, with appropriate “rewards” or “punishments” (weight adjustments) depending on whether or not it proposes a matching solution. Confidence in the dependability of a neural network may normally only be based on statistical arguments about the training sets and the network structure. Nevertheless, neural networks have been very effective in some problem-solving domains, such as language processing, character or pattern recognition, image compression and servo-control in unpredictable environments.

3 Hazards of AI-Based Systems

AI-based systems can provide significant benefits for implementing autonomy, including in certain cases, improved dependability and safety. However, they also introduce new hazards (in addition to the usual hazards associated with software in automated systems). For knowledge-based systems, the following hazards can be identified [Boden 1989][Fox & Das 2000]:

1. Knowledge base “wrong”: beliefs may be incorrect or data may be missing.
2. Unsound inference: the knowledge base may be correct, but inferences drawn from it may be wrong because the inference

procedures being used may be unsound in some way.

3. Unforeseen contingencies: the knowledge base may be correct, but reasoning based on it may break down when it is confronted with some unusual situation not foreseen by the designer.
4. Specificity of decision criteria: the decision criteria built into the system may not be universally acceptable, i.e., they could have adverse side effects in certain situations.

When interfacing a knowledge-based system with a human user, further hazards include (in addition to usual human-factor issues in automated systems):

5. Ontological mismatch: the knowledge base may be correct, but a mismatch occurs between the meaning of the term as used by the system and the meaning that the user attaches to the term.
6. Overconfidence: the user may confer too much faith on the knowledge-based system and be deluded into a false sense of safety. This may be compounded by false precision of the results when quantitative measures of confidence are used.
7. Incredulousness: the user may not believe the recommendations offered by the knowledge-based system, especially if little or no explanation of the reasoning is provided.

4 Case Study 1 — Agent Technology in Medicine

In this section, we present the application of AI-based systems in medicine, as exposed in a recent book [Fox & Das 2000] that has some interesting insights into the use of autonomous agent technology in critical applications.

4.1 Overview

The agent approach followed in [Fox & Das 2000] is based on a common framework for human and machine cognition, called the *domino model*, shown in **Fig. 2**. The domino represents a collection of proposition databases (nodes) and inference procedures (arrows), which add propositions to the databases.

The model is inspired from human decision procedures in medicine:

- The *situation beliefs* database contains a symbolic representation of the state of the agent's knowledge of its environment. In a medical decision support system, this would be, for example, the data concerning a patient: symptoms, test data, diagnoses, current medical treatments, etc.
- The *problem goals* database contains proposed goals for decisions that will modify the state of the agent's environment, and its knowledge thereof. In a medical decision support system, the setting of problem goals is the responsibility of the doctor. Typical goals would be to diagnose, test, treat or prescribe.
- The *candidate solutions* database contains propositions for decisions aimed at meeting a goal. In a medical decision support system, these propositions can be made on the basis of the patient's individual history together with the body of medical knowledge.
- The *decisions* database contains the set of possible decisions with arguments for and against each candidate so as to establish an order of preference. In a medical decision

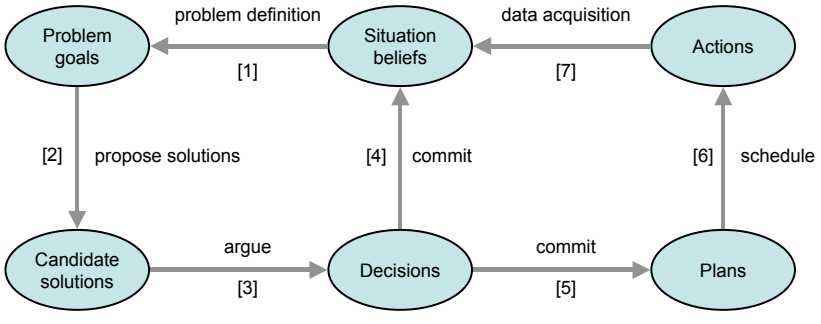

Fig. 2— The domino model [Fox & Das 2000]

support system, the argumentation might be expressed in terms of supporting or conclusive evidence for or against a clinical decision. The set of possible clinical decisions is presented to the doctor in the form of substantiated recommendations. The decision may either lead to a direct addition to the situation beliefs database (e.g., a diagnosis, a test result, treatment or prescription) or to initiate a more complex plan.

- The *plans* database contains the set of plans to which the system has currently committed. A plan may be destined to acquire information about the agent's environment or to create some change that will be consistent with its goals. In a medical decision support system, a plan might be a therapy protocol, e.g., for cancer treatment, involving surgery, radiotherapy and chemotherapy actions.
- The *actions* database contains descriptions of the elementary tasks scheduled as a result of executing a plan. In a medical decision support system, elementary actions might be injections, blood tests, tissue analyses, etc. The execution of actions results in updates to the *situation beliefs* database.

The domino model is fully supported by PRO*forma*, the authors' systematic method of building intelligent systems, and a practical technology supporting the method. The PRO*forma* method maps the basic concepts of the domino model into a standard set of basic tasks including decisions, plans and actions. The development software helps designers to systematically assemble complex cognitive functions from these components, and to verify and test the resulting application. The adopted engineering lifecycle is shown in **Fig. 3**.

Task analysis is the development of a model of expertise by setting out a collection of decisions, plans, actions and enquiries that are required to achieve a goal. Tasks may be connected into a network, with simple scheduling constraints. The tool supports knowledge acquisition for each type of task and expertise modeling in a formal language called the Red Representational Language (R^2L). A knowledge base written in R^2L is a declarative specification of tasks and their interrelationships. Software tools are provided

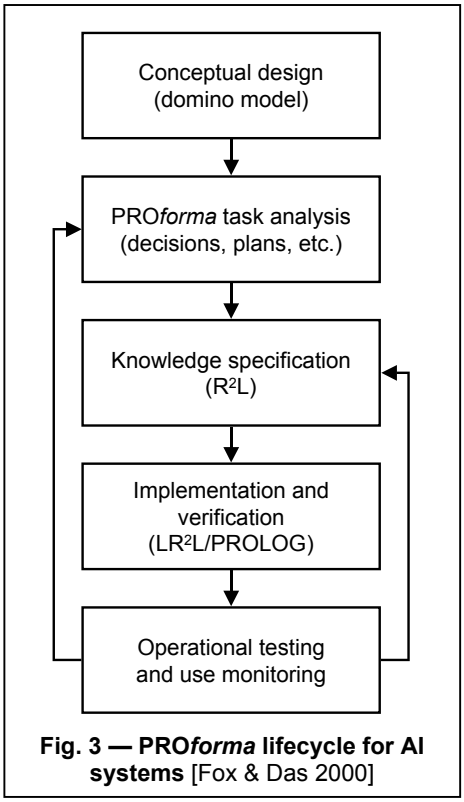

Fig. 3 — PRO*forma* lifecycle for AI systems [Fox & Das 2000]

to analyze the specification and check that it satisfies certain completeness and consistency criteria. With a formal model of the general properties of decisions, plans and many of the constraints within and between tasks, it is possible to automatically identify many problems or potential problems in an R^2L specification. Among the detectable errors are the following:

- Incorrect data types.
- Invalid syntax of attribute values.
- Critical missing values for tasks (e.g., missing candidates or commitment rules in decisions).
- Concepts referred to in inference rules but not defined.
- Inconsistent scheduling or temporal constraints.
- Inconsistent data references.

Once all the syntactic and other formal errors have been removed, it is possible to execute a specification to test the adequacy of its expertise model and its operational behavior. To avoid the technical drawbacks of interpreting R^2L specifications directly, they are translated into another level called the *Logic of* R^2L (L_{R2L}), which is essentially a

temporal propositional logic extended with certain modal operators. Finally, tasks described in L_{R2L} are executed by means of an interpreter written in Prolog.

Although originally intended for designing knowledge-based medical decision support systems, Fox and Das also consider the applicability of the domino model in an autonomous agent setting. In this case, the inferences [1], [4] and [5] on **Fig. 2** would be automated instead of being under a doctor's responsibility. This raises some interesting issues, for example, regarding the automatic commitment of decisions. Although at first sight a commitment can be made automatically by choosing the decision that is highest in rank, a potential problem is *when* to commit. At some point, the balance of argument might strongly favour one option, but if we wait a little, more information might become available that changes the order of preference. Fox and Das propose a safety constraint that essentially says that there are no further arguments that can alter the preferred decision. Such a safety constraint would include the following:

- Demonstrating that in the current state of knowledge, there are no unknown sources of information that could form the grounds of further arguments that would result in a different best action.
- Demonstrating that the expected costs of seeking further information exceed the costs of inappropriately committing to the current preference.

4.2 Dependability Techniques

The use of a knowledge representation based on formal logic, a structured lifecycle and a development method like PRO*forma* adapted to knowledge-based systems, all help the developer to reach a certain confidence in the correctness of the implemented system, but they are insufficient to guarantee safety in face of the hazards listed in Section 3 (except possibly for hazard 2). Quoting from [Fox & Das 2000] (page 133):

> "There are many ways in which we might address the challenge of making agent systems sound and safe. For example, adopting a formal method for software design and development can do much to improve safety. Formal specification, refinement, and verification of software can substantially improve the integrity of a program, as can the use of rigorously tested standard components such as reasoners, decision-procedures, and even knowledge bases. However, even the most stringent empirical testing cannot guarantee against events or situations not foreseen during the design process."
>
> "Safety problems are difficult enough for 'closed' systems where the designers can be relatively confident of knowing all the parameters which can affect performance, and be able to design the software to respond to abnormal states or trends (such as flight control systems). But many systems are to a greater or lesser extent *open*: they operate in an environment which *cannot* be comprehensively monitored or controlled, and in which unpredictable events *will* occur. This may be exactly the kind of application where we want to deploy autonomous agents."

Consequently, Fox and Das go on to investigate techniques for "active safety management" while a knowledge-based agent is in operation. An active safety management system is one that operates in parallel with the agent's primary problem-solving and decision-making functions. They propose an approach called "Guardian Agents" inspired directly from the safety bag approach pioneered by Alcatel Austria in the railway interlocking system ELEKTRA [Klein 1991]. This system contains a logic channel, which processes commands, and a safety channel, which checks the commands according to safety rules. Commands that are only allowed under certain circumstances are carried out only if the instructions generated by processing the command in the logic channel are checked and committed by the safety channel. To minimize the possibility of common errors in both channels, different programming paradigms are used. The logic channel is implemented in a procedural programming language and the safety bag is implemented in PAMELA, a rule-based expert system language. The rule-oriented programming paradigm is well suited since the safety requirements themselves are represented by rules, the rules of the Vienna

railway station's safety policy. This safety policy is thus made explicit and relatively readable for both the original designers and independent inspectors. Since the rules of the safety policy are executable, one can be more confident than if the rules were just the designers' documented intentions.

Guardian Agents are a generalization of the safety bag concept whereby any number of safety bags can be implemented as reactive agents that operate actively and independently, sometimes cooperatively and sometimes competitively, with the primary application agent. They give an example of a Guardian Agent for monitoring a chemotherapy plan for cancer treatment.

Observing that the railway and chemotherapy safety bags rely on a set of domain-specific safety rules, Fox and Das go on to explore the possibility of a domain-independent safety bag that agent designers could use in a wide range of applications. They propose a simple safety protocol, shown in **Fig. 4**, based on the domino model of **Fig. 2**.

The rules of the protocol in Fig. 4 are to be interpreted as *reactive* (situation-driven) rules whose consequences become true whenever their premises are true. Strings with initial capital letters are variables that are universally instantiated. Section 1 of the protocol captures the agent's knowledge of how to assess and respond to potential hazards. Section 2 embodies a simple policy for when the agent must seek authorization for its action. Without describing the protocol in detail, it is nevertheless interesting to note the relation *causes* and the modality *possible* in the premises of rules (4) and (6).

Evaluation of the relation *causes* requires domain-specific knowledge capable of

Section 1

Detect any potentially hazardous anomaly and raise a goal to deal with it.

if results of enquiry is State and (1)
State is not safe
then goal is remedy State

If the abnormal state is a known hazard with a known remedial action then propose it as a candidate solution to the goal.

if goal is remedy State and (2)
known remedy for State is Action
then candidate for remedy of State is Action

Commit to action if the agent can establish that it is permitted according to the rules of the protocol.

if candidate for remedy of State is Action and (3)
decision status of Action is permitted and Action is safe
then decision status of Action is obligatory

Section 2

Any action that could be hazardous must be authorized [before it may be executed].

if candidate for remedy of State is Action and (4)
possible(Action causes NewState) and
NewState is not safe
then authorization of Action is obligatory

When an action has been authorized it is permitted.

if authorization of Action is obligatory and (5)
Action is authorized
then Action is permitted

Any action that has no hazardous consequences is permitted.

if candidate for remedy of State is Action and (6)
not(possible(Action causes NewState) and NewState is not safe)
then Action is permitted

Fig. 4 — A basic generalized safety protocol [Fox & Das 2000]

predicting all the states that can be reached from the current state. Fox and Das cite qualitative simulation and model-based prediction as candidate techniques for such predictions.

The modality *possible* may be formally summarized by the following schemas:

possible State *is equivalent to* not necessarily not State
necessarily State *is equivalent to* not possible not State

Informally, an agent may regard something as possible if it cannot show that it is impossible. Thus, rule (4) in **Fig. 4** can be informally interpreted as follows: if the agent cannot show that all states reachable from the current state by executing Action are safe states, then Action must be authorized (e.g., by a human operator or by another agent).

Fox and Das extend the simple protocol of **Fig. 4** by introducing further rules aimed at actively preventing dangerous situations, including those that could be produced as side effects of actions. They have formalized these notions in a logic for reasoning about safety, L_{safe}. The logic includes some interesting modalities (including deontic modalities) as summarized in **Table 1**.

It is difficult to say without further study whether the "active safety management" approach proposed by Fox and Das for medical care can indeed be extended to agent technologies used in other domains, such as space, but it is definitely an interesting direction to pursue.

Table 1 — Modalities of the safety logic L_{safe} [Fox & Das 2000]

safe	action □ or property □ is safe
authorized	action □ is authorized by a superior agent
preferred	action □ is preferred to action □
permitted	all obligatory preconditions of action □ are satisfied
obligatory	action □ (property □) is obligatory
[t1, t2]	action □ (property □) is true in the interval *t*1 to *t*2

5 Case Study 2 — Deep Space One

In this section, we summarize lessons learnt from the application of AI-based systems in the NASA's on-board controller for Deep Space One, as exposed in [Muscettola *et al.* 1998][Feather & Smith 2001].

5.1 Overview

NASA's "New Millennium" series of spacecraft is intended to evaluate promising new technologies and instruments. The first of these, Deep Space One (DS1), was launched in 1998. Spacecraft autonomy is one of several innovative technologies that DS1 demonstrated. The "Remote Agent" architecture selected as a technology experiment on DS1, is the first artificial intelligent-based autonomy architecture to reside in the flight processor of a spacecraft and control it for several days without ground intervention.

The challenge of developing remote agents for controlling space explorers was driven by four major properties of the spacecraft domain [Muscettola *et al.* 1998]:

- The spacecraft must carry out autonomous operations for long periods of time with no human intervention.
- Autonomous operations must guarantee success, given *tight deadlines* and *resource constraints*.
- Since spacecraft are expensive and often designed for unique missions, spacecraft operations require *high reliability*. Even with the use of highly reliable hardware, the harsh environment of space can still cause unexpected hardware failures. Flight software must compensate for such failures by repairing or reconfiguring the hardware, or switching to possibly degraded operation modes. Providing such a capability is complicated by the need of rapid failure responses to meet hard deadlines and conserve precious resources, and due to limited observability of spacecraft state.
- Spacecraft operation involves *concurrent activity* among a set of *tightly coupled subsystems*, since the subsystems operate as concurrent processes that must be coordinated to enable synergistic

interactions and to control negative ones. For example, while a camera is taking a picture, the attitude controller must hold the spacecraft at a specified attitude, and the main engine must be off since otherwise it would produce too much vibrations.

These characteristics of the domain have led to the development of a Remote Agent (RA) architecture based on the principles of model-based programming, on-board deduction and search, and goal-directed, closed-loop control. **Fig. 5** shows the RA architecture, and its relationship to the flight software within which it is embedded. As regards the RA, the need for autonomous operations with tight resource constraints and hard deadlines dictated the need for a temporal Planner/Scheduler (PS), with an associated mission manager (MM), that manages resources and develops plans that achieve goals in a timely manner. The need for high reliability dictated the use of a reactive executive (EXEC) that provides robust plan execution and coordinates execution time activity, and a model-based mode identification and reconfiguration system (MIR) that enables rapid failure responses in spite of limited observability of spacecraft state. The need to handle concurrent activity impacted the representation formalism used: PS models the domain with concurrently evolving state variables, EXEC uses multiple threads to manage concurrency, and MIR models the spacecraft as a concurrent transition system (see [Muscettola *et al.* 1998] for a detailed description of each entity that constitutes the RA, as well as an extensive discussion of technical issues encountered while developing the RA).

As regards the RA's relationship to the flight software (**Fig. 5**), RA sends out commands to the real-time control system (RT). RT provides the primitive skills of the autonomous system, which take the form of discrete and continuous real-time estimation and control tasks, e.g., attitude determination and attitude control. RT responds to commands by changing the modes of control loops or states of the devices. Information about the status of RT control loops and hardware sensors is passed back to RA either directly or through a set of monitors. Other on-board systems, called planning experts, participate in the planning process by requesting new goals and answering questions from PS (e.g., questions about estimated duration of specified turns and resulting resource consumption).

Through the example of DS1, the process of working on a real mission with a real mission schedule provided valuable lessons about inserting this kind of technology into operational missions. Muscettola and his colleagues identified three key technology insertion lessons:

- *Human-centered operations*. While new classes of missions may require systems with highly autonomous capabilities, it is important that such systems also support operational modes in which humans exercise tight control over the system.
- *Validation and testing*. A major barrier to

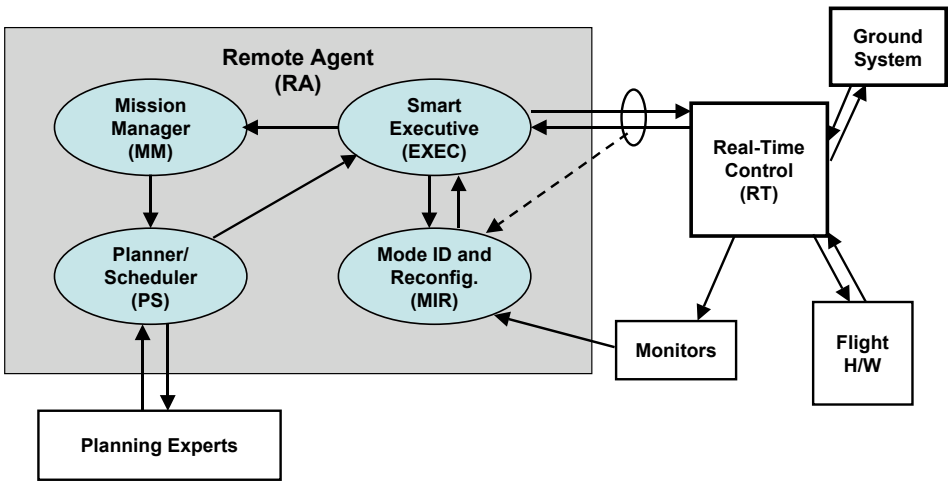

Fig. 5 — RA architecture embedded with flight software [Muscettola *et al.* 1998]

increasingly autonomous systems is concern about how to test them and validate that they will actually perform as desired. Architectural design choices that let spacecraft engineers focus on domain model, rather than on the problem-solving methods, can significantly help address this barrier.

- *Schedule impacts*. Putting an autonomous system on-board a spacecraft potentially has a major impact on the traditional flight software development schedule, as it can require knowledge normally codified during operations (after the system is built) to be encoded in the system early on. *Developing first things first* can alleviate this problem: focus first on the critical models at the level necessary to meet launch requirements; then progressively refine the models to provide increased performance and capabilities. The approach reduces the tendency to have detailed models of some components while major spacecraft capabilities are still unmanaged, and enables the model-based approach to fit into the risk management approach of the overall flight software project.
- *Model-based skunkworks*[1]. Ensuring coherence of mental models across a large software team can be inordinately time-consuming. This has motivated the development of a research paradigm in which all software is programmed in a unified modeling language by a small team supported by automated synthesis techniques and collaborative modeling environments.

Validation and testing issues are further discussed in the next section. To conclude on this short overview of the DS1 project, it is worth noting that the lesson regarding schedule impacts is in favor of the adoption of an evolutionary program strategy, as defined in the [MIL-STD-498].

5.2 Dependability Techniques

In this section, we first concentrate on *robustness* issues, which mainly relate to EXEC and MIR (see **Fig. 5**), and then, on *validation and testing* issues through the PS example.

5.2.1 Robustness

EXEC is a robust event-driven and goal-oriented multi-threaded execution system that coordinates the activity of the other flight software modules, both internal and external to RA. It is built on the Execution Support language (ESL) [Gat 1996], which provides control structures such as loops, parallel activity, synchronization, error handling, and property locks. These language features are used to implement robust schedule execution, hierarchical task decomposition, context-dependent method selection, routine reconfiguration management, and event-driven responses.

One main aspect of EXEC's behavior is *robust plan execution*. EXEC must successfully execute plans in the presence of uncertainty and failures. Such a robustness is achieved by [Muscettola *et al.* 1998]:

- Executing flexible plans by running multiple parallel threads and using fast constraint propagation algorithms in EXEC to exploit plan flexibility.
- Choosing a high level of abstraction for planned activities so as to delegate as many detailed activity decisions as possible to the procedural executive.
- Handling execution failures using a combination of robust procedures and deductive repair planning.

Also, when EXEC is notified by MIR of degraded capabilities of the hardware and control system, it keeps track of such degradation when commanding future planning cycles. Such failures are recognized by MIR trough a combination of monitoring and diagnosis (see below). For example, one fault mode in DS1 is for one of the thrusters to be stuck shut. The attitude control software has redundant control modes to enable it to

[1] The term "skunkworks" was first introduced by Lockheed Martin to denote their engineering, technical, consulting, and advisory services with respect to designing, building, equipping and testing commercial and military aircraft, and related equipment. It has since been adopted in a variety of forms by many organizations as a way to quickly develop solutions by bypassing some of the time-consuming bureaucracy and allowing the team to make ad hoc decisions.

maintain control following the loss of any single thruster, but an effect of this is that turns take longer to complete. When EXEC is notified of this permanent change by MIR, it passes health information back to PS.

MIR is provided by the Livingstone system [Williams & Nayak 1996], which is a discrete model-based controller that sits at the nexus between the high level feed-forward reasoning of classical planning and scheduling systems, and the low level feedback control of continuous adaptive methods (see **Fig. 6**). It is a discrete controller in the sense that it constantly attempts to put the spacecraft hardware and software into a configuration that achieves a set point, called a configuration goal, using a *sensing component*, called mode identification (MI), and a *commanding component*, called mode reconfiguration (MR). It is model-based in the sense that it uses a single declarative, compositional spacecraft model for both MI and MR. In the RA architecture, MR is used primarily to assist EXEC in generating recovery procedures, in response to failures identified by MI

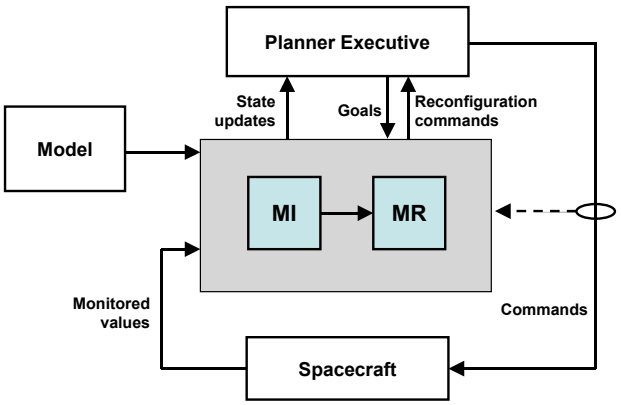

Fig. 6 — Livingstone architecture diagram
(adapted from [Muscettola *et al.* 1998])

MI provides the capability to track changes in the spacecraft's configurations due to executive commands and component failures. It uses the spacecraft model and executive commands to predict the next nominal configuration. It then compares the sensor values predicted by this configuration against the actual values being monitored on the spacecraft. Discrepancies between predicted and monitored values signal a failure. MI isolates the error and diagnoses its cause — thus identifying the actual spacecraft configuration, using algorithms adapted from model-based diagnosis.

When the current configuration ceases to satisfy the active configuration goals, MR capability can identify a least cost set of control procedures that, when invoked, take the spacecraft into a new configuration that satisfies the goals. It can support a variety of functions, including: mode configuration, recovery, standby and safing.

5.2.2 Validation and testing

As regards validation and testing issues, the main lessons highlighted in [Muscettola *et al.* 1998] concern the Planner/Scheduler component. Indeed, the authors note that "while AI planning research has so far concentrated on problem-solving performance, in mission-critical applications it is *validation* of the problem-solving system that takes a much more prominent role".

The fact that systems like the RA promise complete autonomy over a much wider variety of complex situations than was previously possible, makes their validation harder than traditional systems. Fortunately, the use of a declarative approach dictates a clean separation between modeling and problem heuristics within PS. This strict separation of concerns between models and heuristics allowed non-AI specialists to inspect the model and understand the knowledge embedded in the system without having to be experts in AI problem solving methods. This should ensure that the system and mission engineers can focus on guaranteeing that requirements are met, and not on the details on how the reasoning engines manipulate the models in order to produce solutions efficiently. Hence, Muscettola and his colleagues conclude that "inspectable representational techniques and tools to automatically analyze models and synthesize problem solving heuristics are important research areas that will widen the applicability of AI techniques to real-world applications".

Concerning verification and validation of AI planning systems, a recent paper concentrates on an important issue related to planner testing, that is, the development of an automated generator of planner test oracles [Feather &

Smith 2001]. The role of such oracles is to automatically determine whether or not the plans produced by a planner in response to a test suite are correct. Since a sound test suite for a planner should require hundreds of test cases, determining plan correctness is a time and knowledge intensive process. Analyzing these by hand would have been prohibitively expensive and error-prone. Hence, some kind of automated test oracle is clearly needed. In their paper, Feather and Smith describe a progression from two successive pilot studies to the development and use of a planner test oracle for DS1. The results clearly show both the feasibility and the necessity of automated planner test oracles for actual spacecraft's autonomous planner (here, exemplified by the DS1 planner). In fact, without an automated test oracle, it would have been impossible to validate the DS1 planner in a cost-effective manner. Using a large test suite with hundreds of cases allowed the detection of a total of 84 defects related to violations of high-level requirements (70), and syntax errors in the domain model and plan (14). Additionally, the oracle provided information on which of the planner constraints had been exercised in the plan (coverage analysis). This was useful information for assessing how well the test cases exercised the model. Emphasis was also placed on producing "verbose results", that is, results reporting more than "OK" when a plan passed the checks (in that case, the justification of why a temporal constraint was satisfied is also given).

The "oracle" problem is a well-known fundamental issue of software testing. In our opinion, Feather and Smith's contribution to the development of automated test oracles for planner testing is of utmost interest: due to the complexity of AI planning systems, such automated oracles are a prerequisite to the feasibility of any testing process.

6 Conclusions

AI-based autonomous systems pose some significant dependability challenges. They are a relatively new trend in real-world applications and there have been few studies aimed specifically at defining appropriate dependability techniques. However, several tentative conclusions may be drawn from the initial study presented in this paper:

- The problem of verifying and validating knowledge-independent components of an AI-based system (e.g., inference mechanisms) is similar to that of classical software engineering.
- Separate knowledge representation is one key aspect that makes verification and validation of AI-based systems different to that of classical software engineering (cf. hazard 1, Section 3). Checking the consistency and completeness of the knowledge representation has thus received deserved attention (cf. Sections 2.3.1 and 4.1). Note, however, that several authors underline the advantages, from a product dependability viewpoint, of having domain-specific knowledge represented separately from procedural mechanisms making use of it, since it may be more readily checked by domain experts. Moreover, inference mechanisms based on logic may allow formal proof of correctness properties (cf. Section 2.2 and hazard 2, Section 3).
- Learning systems, whose function emerges from training examples or during operation, prove to be quite robust in practice. Nevertheless, they are less amenable to dependability and safety arguments than those whose knowledge and inference mechanisms are determined *a priori* by the designer (cf. Section 2.2 and hazards 1 and 2, Section 3).
- The most significant challenge in the use of AI-based techniques for autonomy is that of unanticipated and complex situations in which the system is nevertheless expected to act sensibly (cf. hazards 3 and 4, Section 3). There are only two apparent (complementary) ways to address this challenge:
  - Use extensive simulation testing to increase statistical confidence that the autonomous system will behave as expected. For really extensive simulation testing, some form of automated oracle should definitely be envisaged (cf. Section 5.2.2).
  - Use on-line dependability techniques, such as the safety-bag or safety supervisor approach to ensure that catastrophic failures are avoided, which implies some form of graceful

degradation (cf. Section 4.2) (see, e.g., the "executive layer" of the autonomy architecture presented in [Alami *et al.* 1998]). The generalization of the safety bag concept towards "active safety management" is also an interesting direction for future research.

- Although autonomous systems are required to operate for extensive periods of time without human intervention, it is important that autonomous systems also support human intervention when necessary (cf. Section 5.1).
- When humans and AI-based systems are to interact synergistically, new human factor risks may be introduced (cf. Section 3, hazards 5, 6 and 7).
- Autonomous operation can significantly impact software development in that domain-specific knowledge needs to be encoded early on (cf. Section 5.1). An evolutionary program development strategy, such as that defined in [MIL-STD-498], should facilitate a progressive refinement approach in which critical autonomous system capabilities may be addressed first.

Acknowledgements

This work was financed by the European Space Agency, under contract ESTEC 14898/01/NL/JA. The authors wish to thank Jean-Paul Blanquart of Astrium, Félix Ingrand of LAAS-CNRS, and Jean-Clair Poncet of Axlog, for the constructive inputs and comments.

References

[Alami *et al.* 1998] R. Alami, R. Chatila, S. Fleury, M. Ghallab and F. Ingrand, "An Architecture for Autonomy", *International Journal of Robotic Research*, 17 (4), pp.315-37, April 1998.

[Boden 1989] M. C. Boden, *Benefits and Risks of Knowledge-Based Systems,* Oxford University Press, 1989.

[Feather & Smith 2001] M.S. Feather and B. Smith, "Automatic Generation of Test Oracles — From Pilot Studies to Application", *Automatic Software Engineering*, Kluwer Academic Publishers, 8, pp.31-61, 2001.

[Fox & Das 2000] J. Fox and S. Das, *Safe and Sound - Artificial Intelligence in Hazardous Applications,* AIAA Press / The MIT Press, 2000.

[Fox & Long 2001] M. Fox and D. Long, *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*, University of Durham, UK, July 4 2001 http://www.dur.ac.uk/d.p.long/pddl2.ps.gz

[Gat 1996] E. Gat, "ESL: A language for Supporting Robust Plan Execution in Embedded Autonomous Agents", in *Proc. AAAI Fall Symposium on Plan Execution*, L. Pryor (Ed.), 1996.

[Johnson *et al.*, 1999] Timothy L. Johnson, Robert Koneck and Stephen F. Bush, "Improving UAV Mission Success Rate through Software Enabled Control Design", *IEEE Aerospace Conference*, ISBN 0-7803-5846-5, IEEE, August 1999.

[Klein 1991] P. Klein, "The Safety Bag Expert System in the Electronic Railway Interlocking System ELEKTRA", *Expert Systems with Applications*, 3 (4), pp.499-560, 1991.

[Lécubin *et al.* 2001] N. Lécubin, J. C. Poncet, D. Powell and P. Thévenod-Fosse, SPAAS: Software Product Assurance for Autonomy on-board Spacecraft. Lessons Learnt from Autonomous Non-Space Applications (Deliverable TN1), Report N°01267, LAAS-CNRS, July 2001. ftp://ftp.estec.esa.nl/pub/tos-qq/qqs/SPAAS/StudyOutputs/SPAAS_TN1_1_0.pdf

[Marcos *et al.* 1995] M. Marcos, S. Moisan, A.P. del Bopil, *"Verification and Validation of Knowledge-Based Program Supervision"*, ECC COMET program, INRIA 1994.

[McDermott 1998] D. McDermott, *PDDL - The Planning Domain Definition Language*, Yale University, Technical Report, N°CVC TR-98-003/DCS TR-1165, October 1998 ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz

[MIL-STD-498] (ST08) Software Development and Documentation, Department of Defense, USA.

[Muscettola *et al.* 1998] N. Muscettota, P. Pandurag Nayak, Barney Pell and B.C. Williams, "Remote Agent: to Boldly go where no AI System Has Gone Before", *Artificial Intelligence*, Elsevier, 103, pp.5-47, 1998.

[Robertson & Fox 2000] D. Robertson and J. Fox, *Industrial Use of Safety-Related Expert Systems*, Health & Safety Executive, UK, Contract Research Report, N°296/2000, 2000.

[Williams & Nayak 1996] B.C. Williams and P.P. Nayak, "A Model-Based Approach to Reactive Self-Configuring Systems", in *AAAI-96,* Portland, OR, AAAI Press, Cambridge, MA, 1996, pp. 971-978.