Interaction between reactive and deliberative tasks for on-line decision-making

Michel Lemaître and **Gérard Verfaillie**
ONERA
2 av. Édouard Belin, BP 74025, F-31055 Toulouse Cédex 4, France
{Michel.Lemaitre,Gerard.Verfaillie}@onera.fr

Abstract

To get actual autonomous engines or systems, it is necessary to equip them with on-line decision-making mechanisms: computation of decisions that fit the current situation, performed in parallel with real execution. However, such a computation introduces a contradiction between requirements on the quality of the decision and on the time at which it will be delivered. This contradiction is all the stronger as decision-making may require intensive computing whose duration is not very well controlled. The solutions that are proposed in the literature to overcome this contradiction are often unsatisfactory, either from the point of view of the quality requirements, or from the one of the temporal requirements, most of the time from the one of the organisation and validation of the whole decision-making mechanism.

In this paper, we propose a generic schema for the interaction between reactive and deliberative tasks for on-line decision-making. The basis of this schema is a control of decision-making by reactive tasks, which may call for anytime deliberative tasks. We show an implementation of this schema using the synchronous *Esterel* language for the programming of the reactive tasks and the *Java* language for the programming of the deliberative ones. The proposed schema is illustrated on an example of on-line management of observation tasks performed by an Earth detection and observation satellite. We conclude with a synthesis of the advantages of the proposed approach.

On-line decision-making problematics

In many fields, but especially in the aeronautics and space one, one wants to have at one's disposal more and more autonomous engines or systems, able to manage their activities without any permanent control exercised by human operators. This is for example the case in the field of Earth watching and observation where one would like to have at one's disposal satellites able to perform systematic detection of ground phenomena, such as eruptions, fires, floods, pollutions, ... and to trigger autonomously higher resolution observations of the areas on which phenomena have been detected.

Such an autonomy requires that the engine or system be able to make decisions that fit the current situation: state of the engine or system, state of its environment, and state of the objectives it is currently assigned. Roughly speaking, one can say that there exist three main approaches to decision-making:

1. *off-line computation* of an *optimal policy*, associating with each possible situation an optimal decision in this situation;
2. *off-line learning* (possibly carried on on-line) of a *policy*, associating with each possible situation a decision in this situation too;
3. *on-line computation* of a *decision* or of a sequence of decisions that fit the current situation.

The first two approaches aim at building a policy, that is a function which associates a decision with each possible situation. They differ mainly about the means used to build this function: *model-based* reasoning for the first one, *simulation-based* learning for the second one. They differ also about the result they provide: possibly optimal for the first one, sub-optimal for the second one. However, both of them come up against the same difficulty: as soon as one wants to deal with a real application, the space of the possible situations becomes quickly huge; practical difficulties appear in terms of computation and particularly in terms of memory; the space that is necessary to memorize a policy becomes quickly incompatible with the memory that is available on an engine.

If everything cannot be off-line computed and memorized, on-line computation is necessary. Unfortunately, on-line computation introduces a contradiction between requirements in terms of decision *quality* and in terms of decision *delivery time*. One wants good quality decisions which do not jeopardize the engine or system and which satisfy as well as possible the mission objectives, but one wants also that these decisions be delivered in time. For example, in the case of an Earth observation satellite, if the decision-making software says that the right decision is to perform the observation of a given ground area z, but if it says that when the visibility of z is over, then this information has strictly no value.

This contradiction is all the stronger as decision-making requires often heavy computation tasks, which consist in solving complex problems (in the meaning of complexity theory): sometimes *polynomial*, but most of the time *NP-hard*, indeed *Pspace-hard*. To face this contradiction, various solutions have been proposed. Some of them give priority to *temporal* requirements:

1. this is the case when one uses pre-defined decision rules which fix immediately the decision to make according to some prominent features of the current situation;
2. this is also the case when one limits the degrees of freedom of the system in order to simplify the decision-making problem and to get a problem, for example polynomial, which can be certainly solved by meeting the temporal requirements;
3. this is finally the case when one uses limited search mechanisms, such as greedy search, local search, or limited tree search.

Their main drawback is a possible loss in terms of quality, to be sure that the temporal requirements be met. Others give priority to *quality* requirements:

1. this is the case when one keeps enough time for decision-making, even if the engine or system must be maintained in a waiting state until decision delivery (Muscettola *et al.* 1998);
2. this is still the case when one allows decision-making and execution to be interleaved, with a possible execution of some parts of the plan while others are still in hand (Lemai & Ingrand 2004).

Their main drawback is a possible loss of opportunities due to the wait for decision delivery. In addition, the waiting state may not be neutral. For example, it may consume energy and possibly jeopardize the engine or system.

Between both these extreme approaches, the so-called *anytime* algorithms (Zilberstein 1996), which guarantee the production of a first solution within a limited time and then its progressive improvement, seem to be at the basis of a sensible compromise between temporal and quality requirements. This is the basis we adopt.

However, most of the works about anytime algorithms focused on a very ambitious and certainly unrealistic objective: using the knowledge (assumed to be available) about the *quality profile* of the anytime algorithms (the way solution quality improves as a function of the computation time) and about the *utility function* (the way decision utility evolves as a function of its quality and of its delivery time) to control these algorithms in order to get the best compromise between reasoning and decision-making (Russel & Wefald 1991; Boddy & Dean 1994; Horvitz 1987; Hansen & Zilberstein 2001).

We adopt a more realistic point of view which does not assume any knowledge of the quality profiles and of the utility functions. We consider that temporal constraints are hard requirements in the form of *deadlines* and that reasoning must compulsorily meet these requirements. This does not imply that reasoning must be systematically instantaneous because, in many applications, deadlines are not systematically immediate. When, for example, one triggers a task with a given duration, one has at one's disposal this duration to perform computations allowing one to decide upon the following task to trigger.

We show how temporal constraints can be met via *reactive control tasks*, how maximum quality can be sought via *deliberative reasoning tasks* and what can be the exchanges between both these kinds of task inside the control software of an engine or system.

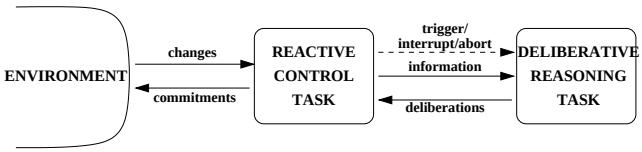

Figure 1: Global schema of the interactions between the environment, a reactive task and a deliberative one

Generic schema of interaction between reactive and deliberative tasks

The first principle of the proposed interaction schema is that the reactive tasks are at the *interface* between the environment and the deliberative ones. Deliberative tasks do not interact directly with the environment. They interact with it via reactive tasks. See Figure 1 for a representation of the interaction between the environment, a reactive task, and a deliberative one.

The second principle is that the reactive tasks are in charge (1) of the interaction with the environment at the rhythm it dictates and (2) of the control of the deliberative tasks. Concerning (1) the interaction with the environment, reactive tasks receive information about the way it evolves (what is referred to as *changes*) and emit action *commitments* following decisions. Concerning (2) the control of the deliberative tasks, reactive tasks trigger, interrupt, trigger again, or abort them; they provide deliberative tasks with relevant *information* about the problem to solve and receive *deliberations* resulting from the reasoning the latter perform. To sum up, reactive tasks can be seen as immersed in an environment made (1) of the physical environment (the physical system and the other components of the control system) and (2) of associated deliberative tasks (a reactive task can indeed control several deliberative ones).

The third principle is that, to meet the temporal requirements on the interaction with their environment (physical environment and deliberative tasks), the reactive tasks are designed so as to meet a *synchronous behavior* assumption: in case of any event coming from its environment, a synchronous reactive task is able to compute and to execute a reaction, then to update its internal memory before any other event arrival; in the abstract, this reaction and this updating can be seen as instantaneous.

The fourth principle is that the temporal requirements that are imposed by the reactive tasks on deliberative ones take the form of *deadlines*: to produce a deliberation by a given time. To meet these deadlines, deliberative tasks are designed so as to meet a *anytime behavior* assumption: when it is triggered, a deliberative task is able to produce quickly a first solution and then to improve on it; each time it produces a better solution, it extracts from this solution the part that is of interest for the associated reactive task (for example, the first decision if the deliberative task produces a plan made of a sequence of decisions) and sends it to the associated

reactive task (what is referred to as a *deliberation*).

The fifth last principle is that the deliberations that are successively emitted from deliberative tasks to reactive ones are only *advice* and that reactive tasks are the only responsible for the emission of commitments to the environment when deadlines occur. According to the current state of the environment, a reactive task may follow the last deliberation emitted by the associated deliberative task. It may choose among the deliberations that have been successively emitted or among the ones that have been emitted by several associated deliberative tasks. It may also follow none of them and emit any commitment it chooses. This is especially what happens when no deliberation is available or when all the available deliberations are inconsistent with the current state of the environment. The latter situation may occur when a deadline and a change from the environment occur at the same time.

These principles are not all original and many of them originate from previous studies such as (Adelantado & de Givry 1995). They have been also widely inspired by previous application studies about the autonomous control of Earth detection and observation satellites (Damiani, Verfaillie, & Charmeau 2005).

It is important to stress the difference between this architecture and the classical *three-layered control* architecture, widely used in the robot control community (see for example (Alami *et al.* 1998; Muscettola *et al.* 1998)). In a few words, deliberative planning is the core of a three-layered architecture: nothing can be executed if it has not been planned before. On the contrary, reactive tasks are the core of the architecture we propose: deliberative tasks can make proposals, but reactive tasks remain responsible for final decisions and can make decisions even when no proposal is available.

Implementation using Esterel and Java

Programming languages To implement these principles, we use the synchronous *Esterel* language for programming reactive tasks, the *Java* language for programming the deliberative ones and more generally for acting as a host language, and the so-called *task* mechanism offered by *Esterel* for connecting reactive and deliberative ones.

We chose the *Esterel* language (Berry & Gonthier 1992) for four reasons: (1) it is built on the *synchronous* assumption, (2) its *semantics* is clearly defined, (3) there exist associated *validation* tools based on *model checking*, and (4) it provides the user with a *task* mechanism which allows synchronous reactive tasks and external concurrent asynchronous ones, written in any host language such as *Java*, to be connected. To be more precise about the *synchronous* assumption, two complementary points of view can coexist:

1. from the abstract point of view of *Esterel* programming the control flow goes forward by means of a potentially infinite sequence of *reactions*, each one involving a finite sequence of instantaneous actions; so, according to this point of view, each reaction is *instantaneous* too; it takes no time; this abstraction makes reasoning about interaction with the environment and about concurrency much easier; this is however the responsibility of the *Esterel* compiler to check that each reaction entail a finite number of actions (no infinite loop);
2. of course, from a practical point of view, the processing of each reaction takes an amount of time which cannot be null; so, we have to check in some way that, taking into account the CPU capabilities, each reaction processing time has an upper bound which is lower that the lower bound on the time between two successive events from the environment; tools exist for this purpose; if this requirement is not met, usual solutions are to use a more powerful CPU, to redesign the reactive task to be less CPU consuming, to split the reaction work into several successive reactions, or to transfer some parts of the reactive work to a deliberative task with lower priority.

Figure 2: Main functional schema of the interactions between the environment, a reactive task and a deliberative one

The *Java* host language provides the *Esterel* code with support for external types, procedures and functions. Moreover, the *Esterel* code is compiled into the *Java* host language in the form of a compact function that implements each possible reaction of the reactive task. It is worth noting that, although *Esterel* programming makes heavily use of concurrency constructs via *logical threads*, all these threads are then flattened into a single *physical thread* by the compiler from *Esterel* to *Java*.

Main functional schema Figure 2 represents the main functional schema of the interaction between the environment, a reactive task, and a deliberative one. It shows the main *events* that are exchanged between tasks (those that appear above arrows) or between modules inside the reactive one (those that appear in the box associated with the reactive task). One must stress that, in the *Esterel* language, events can be *pure* signals (present or absent) or *valued* signals (carrying information of any type, when they are present).

Main events The reactive task may receive from the environment a *time* event informing it that time is getting on and an *evolution* event informing it of a change in the state of the environment. It may send to the environment a *commitment* event informing it of an action commitment following a decision. It may also trigger the execution of a deliberative task with information about the current state of the environment (*exec(?state)*) and receive from the deliberative task a *deliberation* event informing it of a better solution. The events exchanged inside the reactive task are *state* (informing of the updating of the state of the environment), *nextDeadline* (informing of the date of the next deadline), and *deadline* (informing of the occurrence of a deadline).

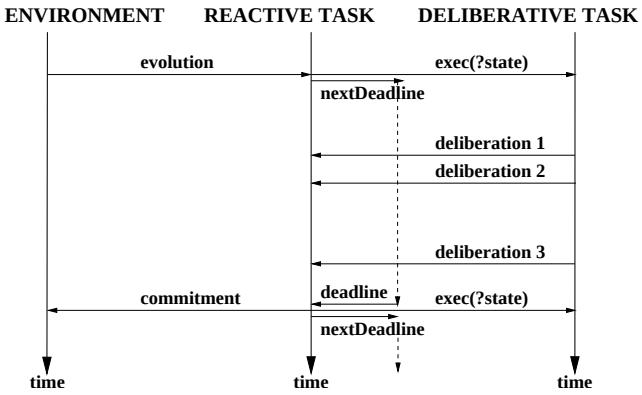

Figure 3: Execution example

Impact of an *evolution* event At each occurrence of an *evolution* event, the reactive task computes the date of the next deadline, updates the state of the environment, and triggers or triggers again the deliberative task on the basis of the new state, everything *instantaneously*, that is before any other event occurrence.

In many applications, there exist *natural* deadlines associated with the dates available for action triggering, either at fixed times, or at the end of the current action when actions are executed in sequence. When such natural deadlines do not exist, it is always possible to consider *artificial* deadlines by any fixed time. One must stress that the deadline may be immediate. In such a case, no deliberative task is triggered.

Impact of a *deadline* event At each occurrence of a *deadline* event, the reactive task computes, always *instantaneously*, a decision as a function of the current state of the environment and of the successive deliberations possibly received from the deliberative task. This decision may be empty, implying no action commitment. If it is not empty, the associated *commitment* event is emitted to the environment.

Setting that the reactive task computes a decision as a function of the state of the environment and of the successive deliberations implies that it is able (1) to check *consistency* between a deliberation and the environment state (for example, the engine or system will be jeopardized neither immediately, nor in the future) and (2) to produce a *consistent decision* when no deliberation is available or all the available deliberations are inconsistent. This requires that the reactive task have at its disposal not only a model of *consistency* between a deliberation and the environment state (model used in a checking mode), but also some *rules* able to produce a consistent *default decision* whatever the environment state is (rules which have been automatically or manually produced off-line).

Execution example Figure 3 shows a typical execution example. The way time gets on is represented from top to bottom, as in *UML sequence diagrams*. An *evolution* event

```
1 every evolution do
2    loop
3       exec deliberative(?state)
4          return deliberation ;
5    end loop
6 end every
```

Figure 4: Excerpt from the Esterel code

triggers the updating of the environment state, the computing of a deadline, the emission of a *nextDeadline* event, and the triggering of the deliberative task. Before the occurrence of the *deadline* event, the deliberative task emits three successive deliberations. Then, the *deadline* event triggers the computing of a decision, which is not empty in this case, the emission of a *commitment* event, the updating of the environment state, the computing of a new deadline, the emission of a new *nextDeadline* event, and again the triggering of the deliberative task.

An excerpt from the Esterel code Figure 4 shows an excerpt from the *Esterel* code responsible for driving a deliberative task. The actual code is a bit more involved, but the one given here is quite representative.

This small excerpt is made of a single *every* instruction and its body. Line 1, the program waits for the presence of an *evolution* event. If such an event is present, the program enters the body of the *every* instruction (lines 2 to 5). Later, each time an *evolution* event is present, the current execution of the body is aborted, including the deliberative task which is launched inside, and the body is entered again.

Let us examine now the body of the *every* instruction. The program enters a *loop* whose body (lines 3 and 4) consists of a single *exec* instruction. This instruction triggers a thread which will start or continue the deliberative task, with the state of the environment (*?state*) as an argument.

If the deliberative task terminates normally, it makes a *deliberation* event present. Then, the deliberative task is triggered again, as the loop commands it, in search for a possibly better deliberation result.

But, if an *evolution* event is present before the normal termination of the deliberative task, then the body of the *every* instruction is aborted, as explained above. This results in the abortion of the deliberative task. The body of the *every* instruction is entered again immediately, triggering again the deliberative task. This way, an *evolution* event is taken into account immediately each time it is present.

Open options This generic schema of interaction between reactive and deliberative tasks has been actually implemented using *Esterel* and *Java*, everything being compiled to produce a *Java* code. One must stress that it keeps many options open:

- a reactive task may trigger several deliberative tasks dealing with various problems or with the same problem using various models or algorithms;

- whereas it is important that the deliberative task be able to produce successive deliberations, it is not really compulsory that the quality of these deliberations monotonically increase; it suffices that the reactive task have at its disposal rules allowing it to choose between several deliberations; if, for example, the successive deliberations result from reasonings performed on temporal horizons of increasing length, a sensible rule consists in choosing the deliberation that results from reasoning on the longest horizon, even if it is known that there is no guarantee that such a deliberation be the best one (Pearl 1983);
- the deliberative task may be systematically aborted and triggered again each time an *evolution* or *commitment* event occurs; it may be merely interrupted and triggered again on the basis of the new state; in the latter case, it may trigger a completely new search, or try and reuse most of the results produced by previous searches; see (Verfaillie & Jussien 2005) for a survey of the existing reuse techniques in a dynamic setting;
- when an *evolution* event occurs, the reactive task may perform a quick analysis of this evolution to decide whether or not the deliberative task must be aborted or interrupted according to the potential impact of this evolution, in order to avoid useless abortions or interruptions;
- the deliberative task may ignore the deadline or be informed of it and use this information to choose a model or an algorithm that fits *a priori* the deadline; see (Lobjois & Lemaître 1998; Lemaître & Verfaillie 2001) for examples of techniques allowing a model or an algorithm to be selected as a function of the deadline.

Example of management of observation tasks

The implemented generic schema has been experimented on the problem which inspired it: the management of the observations performed by an Earth detection and observation satellite (Damiani, Verfaillie, & Charmeau 2005).

Application The satellite is assumed to be able to detect ground phenomena such as forest fires or volcanic eruptions (*detection* performed in front of the satellite thanks to a wide swath instrument). In case of detection, it triggers automatically an alarm to the ground and an internal request for a higher resolution observation of the ground area on which the phenomenon has been detected (*observation* performed thanks a narrow swath instrument when the satellite flies over the area, about one minute after the detection). The difficulty is that the satellite has at its disposal only one observation instrument. This may result in conflicts between observation requests. In general, it will be impossible to satisfy all of them, at least during one satellite revolution. The problem is to satisfy as many as possible of them and preferably the most important ones.

The choice of the observations to perform is assumed to be made by an observation management module. This is this module we programmed, with its reactive features as well as its deliberative ones, using the implemented generic schema.

Optimisation problem For this experiment, we limited ourselves to the binary conflicts between observation requests (we ignored conflicts resulting from the common use of on-board energy and mass memory). A fixed execution window is associated with each candidate observation. Two observations are incompatible if and only if their execution windows overlap. Moreover, we assumed that a gain is associated with each observation request and that the gain associated with a sequence of observations is merely the sum of the gains associated with the observations in the sequence (actually, one will prefer often to reason in terms of priority).

In such conditions, the objective is at any time to build a sequence of observations of maximum associated gain on the basis of the pending observation requests over a given temporal horizon: known, not yet achieved, and still achievable observations.

Optimisation algorithm Still for this experiment, we chose an iterated stochastic greedy algorithm, inspired from the *Heuristic-Biased Stochastic Sampling* (HBSS (Bresina 1996)) and *Value-Biased Stochastic Sampling* (VBSS (Cicirello & Smith 2005)) algorithms. This algorithm consists in performing a *sequence of stochastic greedy searches*. Each greedy search performs itself a sequence of *stochastic choices* which are *biased* by a *heuristics*. Satisfaction of the physical constraints is checked before each choice. During a search, choices are never undone. Stochasticity allows the same choices not to be systematically made search after search, whereas the heuristic bias prompts each search to explore the neighborhood of the purely heuristic solution, resulting globally in a good trade-off between *diversification* and *intensification* of the search.

The main advantages of such an algorithm are that it is very easy to implement and is naturally *anytime*: each greedy search produces a consistent sequence of observations; a consistent sequence of observations is thus available from the first search, that is very quickly; improvement is achieved each time a greedy search produces a sequence whose associated gain is higher than the best gain produced so far. It main drawbacks are that it offers no guarantee of optimality and that its efficiency strongly depends on the chosen heuristic and bias functions.

The heuristics we used in this experiment is a usual knapsack heuristics, which associates with each observation request a weight equal to its gain divided by its duration. Observations are inserted one after each other, when insertion is possible, in a decreasing weight order. To get a stochastic search, these weights are merely modified by multiplying them by a noise factor.

Events and reactions In this application, an *evolution* event is triggered by the detection of one or several ground phenomena which gives rise to new pending observation requests.

For this experiment, we assumed that observations can be triggered at the last time (we ignored the movement of the sight mirror necessary to have the right ground area in view). This implies that the deadlines are merely the starting dates

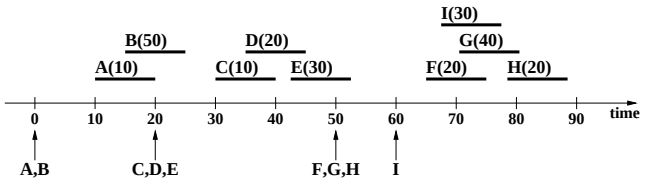

Figure 5: Example of scenario

of the observation execution windows. The next deadline is thus the minimum date among the starting dates of the execution windows associated with the pending observation requests.

A deliberation involves only the next observation to perform according the optimisation algorithm. A *deliberation* event is thus emitted from the first greedy search, containing the first observation of the produced sequence. Later on, a *deliberation* event is emitted each time a greedy search produces a sequence of observations which improves on the total gain and modifies the previous deliberation (change in the first observation of the produced sequence).

When a *deadline* event occurs, the policy consists in following the last deliberation, if at least one deliberation is available: if the observation associated with the deadline and the one associated with the last deliberation coincide, then the latter is triggered; else it is left; if no deliberation is available, the observation associated with the deadline is triggered by default. In every case, the set of pending requests is modified, because the observation associated with the deadline is now impossible and, if it is triggered, all the observations that are in conflict with it become impossible too.

Scenario example Figure 5 shows an example of scenario involving nine observation requests from A to I. The time at which each of them arrived following detection appears below the time line: A and B arrived at time 0; C, D, and E arrived at time 20 ... Their execution windows appear above the time line. All of them have the same duration (10). The associated gains appear in parenthesis: the gain associated with A is 10, the one associated with B is 50 ... One can observe for example that A and B are incompatible and that D is incompatible with C and E.

Execution example On this micro-scenario, optimisation problems are very small and the stochastic greedy search is able to produce an optimal solution after only some greedy searches, sometimes from the first one. After the arrival of observations A and B at time 0, the first deliberation is B (B is the next observation to perform). Then, after the arrival of observations C, D, and E at time 20, the first deliberation is C. After the triggering of C, the first one is E. Then, after the arrival of observations F, G, and H at time 50, the first deliberation is G. After the arrival of observation I at time 60, the first deliberation remains G and some stochastic greedy searches are necessary to find the optimal one I. Finally, after the triggering of I, the first deliberation is H.

Finally, the sequence of performed observations is $[B, C, E, I, H]$ resulting in a gain of 140, the optimum on this scenario. It must be stressed that, if we would have used a mere greedy search with the same heuristics, the sequence would have been $[B, C, E, G]$, resulting in a gain of 130, 10 units smaller. With mere decision rules, things would have been worse: with a rule stating that an observation must be performed if it is possible, the sequence would have been $[A, C, E, F, H]$ resulting in a gain of 90, 50 units below the optimum; with a slightly more intelligent rule stating that an observation must be performed if it is possible and not in direct conflict with a more important one, the sequence would have been $[B, E, G]$ resulting in a gain of 120, still 20 units below the optimum. This shows the potential gain resulting from the use of on-line optimisation tools.

Conclusion

If we try now a synthesis of the proposed approach, its first advantage is that it allows actually *on-line decision-making* as a function of the current situation: changes from the environment, whatever they are, positive or negative, are immediately taken into account by both the reactive and deliberative tasks.

The second advantage is that all the *time* which is available between a change and the next deadline is *used* by the deliberative task to try and produce the best possible deliberation: if the deadline is distant and if the problem to solve is not too complex, an optimal or quasi-optimal deliberation will be possibly produced; otherwise, deliberation quality will possibly suffer on account of lack of time.

The third advantage is that the reactive task offers *guarantees* in terms of decision: deliberations are only advice whose consistency with the current situation is checked before commitment; even if no deliberation is available, rules allow a default decision fitting the current situation to be produced. In fact, the engine or system can work, maybe worse, but still correctly, without any deliberative task.

The fourth advantage is a clear *distinction* between the respective roles of the reactive and deliberative tasks and a clear definition of their interactions. This contrasts with numerous implementations of on-line decision-making systems where reactive and deliberative activities are mixed up into a unique code.

The fifth one results from the use of a *synchronous language* like *Esterel* whose semantics is well-defined. Provided that is has been checked that the implementation meets properly the synchronous assumption, we get a program whose execution is not ambiguous. This contrasts with the previous implementation of the example of management of observations by an Earth detection and observation satellite, which used the *JADE* multi-agent platform (*Java Agent DEvelopment framework*) (Damiani, Verfaillie, & Charmeau 2005) and with which execution anomalies due to time management were frequent.

The sixth and last one is that still the use of *synchronous languages* paves the way to a *validation* of the control software. It is well known that the formal validation of a control software involving on-line decision-making mechanisms sets difficult problems, not satisfactorily solved yet.

With the proposed schema, the reactive tasks could be validated using formal methods such as *model checking* (Clarke, Grumberg, & Peled 1999) and the presence of the deliberative tasks could be taken into account via the models used by the reactive tasks to check the *consistency* of the deliberations they receive from deliberative ones.

Acknowledgements

This work has been done thanks to the CNES-ONERA-LAAS *AGATA* project (*Autonomy Generic Architecture : Tests and Applications*; see http://www.agata.fr), whose aim is to develop technical tools allowing space system autonomy to be improved.

References

[Adelantado & de Givry 1995] Adelantado, M., and de Givry, S. 1995. Reactive/Anytime Agents: Towards Intelligent Agents with Real-Time Performance. In *IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*.

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

[Berry & Gonthier 1992] Berry, G., and Gonthier, G. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19(2):87–152.

[Boddy & Dean 1994] Boddy, M., and Dean, T. 1994. Deliberation Scheduling for Problem Solving in Time-Constrained Environments. *Artificial Intelligence* 67(2):245–285.

[Bresina 1996] Bresina, J. 1996. Heuristic-Biased Stochastic Sampling. In *Proc. of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 271–278.

[Cicirello & Smith 2005] Cicirello, V., and Smith, S. 2005. Enhancing Stochastic Search Performance by Value-Biased Randomization of Heuristics. *Journal of Heuristics* 11(1):5–34.

[Clarke, Grumberg, & Peled 1999] Clarke, E.; Grumberg, O.; and Peled, D. 1999. *Model Checking*. MIT Press.

[Damiani, Verfaillie, & Charmeau 2005] Damiani, S.; Verfaillie, G.; and Charmeau, M.-C. 2005. Cooperating Onboard and On the ground Decision Modules for the Management of an Earth Watching Constellation. In *Proc. of the 8th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (i-SAIRAS-05)*.

[Hansen & Zilberstein 2001] Hansen, E., and Zilberstein, S. 2001. Monitoring and Control of Anytime Algorithms: A Dynamic Programming Approach. *Artificial Intelligence* 126:139–157.

[Horvitz 1987] Horvitz, E. 1987. Reasoning about Beliefs and Actions under Computational Resource Constraints. In *Proc. of the 3rd International Conference on Uncertainty in Artificial Intelligence (UAI-87)*, 301–324.

[Lemai & Ingrand 2004] Lemai, S., and Ingrand, F. 2004. Interleaving Temporal Planning and Execution in Robotics Domains. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI-04)*.

[Lemaître & Verfaillie 2001] Lemaître, M., and Verfaillie, G. 2001. Learning the Temporal Monitoring of an On-line Constraint Optimization Algorithm. In *Proc. of the CP-01 Workshop on "On-line Combinatorial Problem Solving and Constraint Programming"*, 15–24.

[Lobjois & Lemaître 1998] Lobjois, L., and Lemaître, M. 1998. Branch and Bound Algorithm Selection by Performance Prediction. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 353–358.

[Muscettola *et al.* 1998] Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–48.

[Pearl 1983] Pearl, J. 1983. On the Nature of Pathology in Game Searching. *Artificial Intelligence* 20:427–453.

[Russel & Wefald 1991] Russel, S., and Wefald, E. 1991. *Do the Right Thing*. MIT Press.

[Verfaillie & Jussien 2005] Verfaillie, G., and Jussien, N. 2005. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints* 10(3):253–281.

[Zilberstein 1996] Zilberstein, S. 1996. Using Anytime Algorithms in Intelligent Systems. *AI Magazine* 17(3):73–83.