Structuring processes into abilities : an information-oriented architecture for autonomous robots

Arnaud Degroote and Simon Lacroix [1,2]

Abstract

This paper presents a framework to organize the various processes that endow a robot with autonomy. The main objectives are to allow the achievement of a variety of missions without an explicit writing of control schemes by the developer, and the possibility to augment the robot capacities without any major rewriting. The organization relies on the notion of *ability*, that encapsulates the means to produce various *information* within the complete system. The mechanisms that autonomously activate the various abilities are depicted, and illustrated in the case of an autonomous navigation mission.

Keywords. Decisional architecture, robot autonomy, supervision

1. Introduction

The robotics community has produced tremendous achievements in the wide spectrum of processes required by autonomous operations: perception, planning, control, learning, human-robot interactions... But it is the *assembly* of these processes that leads to autonomy. This assembly, often referred to as “decisional architecture”, is in charge of configuring, scheduling, triggering and monitoring the execution of the various processes. It should be designed in order to endow the robot with *(i)* the capacity to achieve a *variety* of high level missions, without manual configuration; and *(ii)* the capacity to cope with a variety of events which are not necessarily a priori known, in a mostly unpredictable world – these two capacities being essential characteristics of autonomy.

Related work. The most popular architectural paradigm for autonomous robots is probably the three layered architecture. In [5], E. Gatt argues that the consideration of the internal state naturally yields the definition of three layers: an intermediate layer is necessary to tie the functional layer, that has no or ephemeral internal state, with the decisional layer, a symbolic planner that strongly relies on a long lasting internal state. Much work has been devoted to this intermediate layer: in [5], it is called "sequencer", and is in charge of translating the symbolic plan into a sequence of elementary behaviours, conditionally to the current situation. In the LAAS architecture [1], the layer called “ex-

[1]CNRS; LAAS; 7 avenue du colonel Roche, F-31077 Toulouse, France
[2]Université de Toulouse ; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
$firstname.lastname@laas.fr$

ecutive" is slightly different, in that it is only responsible to *control* the proper execution of the sequence of behaviours. A dedicated component in the decision layer is in charge of decompositing of plans into an executable sequence, the "supervisor", based on the PRS language [6]. The *Remote Agent* system [3] combines the translation of plans into atomic tasks, execution control, event management and even resource management. In case of task execution failure, the EXEC layer can ask a recovery expert called MIR (and not the usual decisional layer). [7] presents PLEXIL, an approach to tie the functional and decisional layer that relies on a predictable and verifiable language to define a robust executive layer.

Even if there are some differences between these approaches, they all rely on the main idea that an intermediate layer is required to fill the gap between the functional and symbolic worlds. This leads to different representations of plans, models and information coexist in the different layers, that leads in turn to difficult diagnostics of plan failures, because the planner does not have relevant information about the failure causes, or inefficient plan executions, because the executive layer does not have a global view of the plan. A first step to solve these issues has been done by the CLARAty system [4]: even if there are still two different tools and representations for the decisional layer (CASPER) and the executive layer (TDL), the system has some way to reflect changes from one representation to another, and exploits heuristics to decide which subsystem will handle the faults. IDEA [11] defines a two-layer architecture: the problem is partitioned into several agents relying on the *same plan model*, each one composed of a planner and an execution layer. In this way, the planning and the execution phase are interleaved into a consistent way. Moreover, during execution the different agents are synchronised to maintain a consistency of the global plan. The T-REX architecture [10] takes after the IDEA approach, in that it uses a collection of agents, but it also introduces some systematic formulation for exchanging states between these agents. This offers more guarantees on the coherence of the global plan execution. Moreover, T-REX uses a systematic approach to drive re-planning, based on state estimation.

Another issue of a three-level architecture is its scalability: it can hardly handle robots with numerous abilities or the realization of a variety of missions. In such an architecture, the decisional and execution layers are two separate "monolithic" blocks, without a fine granularity for information representation and manipulation. Adding or removing functionalities to deal with a new robot or new kinds of missions often leads to heavy side effects in these layers, requiring a major rewrite. Increasing the functionalities results in increasing the deliberation time, making the robot less reactive to situation changes. Mc Gann *and al* [9] state that having one big plan and one execution layer is not scalable on the long run, and conclude that the problem needs to be portioned to be efficiently handled: the use of different planning agents, with different timing constraints, partially solves the scalability issue.

The partition proposed in [9] is defined and constructed by the programmer, based on the mission needs. If the nature of the mission changes, the whole partition must be accordingly reorganized: this kind of construction does not scale well over a large variety of missions, missing the objective of a versatile architecture for robots.

Objectives. The principle of *partitioning* the robot functionalities into a network of components is essential to simplify the overall system control. This partition must be carefully designed : in particular, it must allow the addition or removal of some components without breaking other parts of the framework. In other words, each component

and its interactions with the other components must be defined by an abstract formal description, thus yielding a *composability* property of the whole system.

We mentioned that autonomy implies the capability to properly react to *a priori* unknown events or situations (though handling correctly *any* situation remains a wishful thinking): for the architecture to choose the most suitable strategy to *adapt* to unknown situations, a formal internal model is required. Finally, we consider that a good architecture must be *(i) verifiable*, *i.e.* provide guarantees about the execution of each component and about the interaction between the different components, in particular avoiding deadlocks between different components, and *(ii) robust to failure*: in case of a component failure due to a logic or a programming error, the framework must pursue its operation, using alternative strategies to handle the mission.

Approach. We propose in this paper the definition of a partition scheme that fulfils these requirements. We follow the decomposition principle exploited in IDEA or T-REX . But contrarily to these architectures in which the decomposition is defined according to a set of tasks, we rely on the explicit definition of the various dynamic information that are present in the system: information on the world, information on the robot intentions (or plans), and information on the robot internal states[3]. The relations between the various information that we consider define a dynamic graph: the framework is in charge of maintaining this graph, ensuring that the different relations are enforced. This is achieved by constraint solvers associated to each information node: a solver tries to locally enforce the constraints set on the associated information. In case of impossibility, the failure goes back through the graph until it is solved by a defined policy – *e.g.* by a call to a planner or (in last resort) to a human operator. In this way, the framework can handle a variety of problems without changing the definition of each information node: the system adapts the information graph to handle the problem at hand, and the different constraint solvers locally schedule the access to information.

Outline. The next section provides an overview of the different components that define the architecture, and how they interact. It also illustrates the approach in the case of a navigation mission handled with several motion modes. Sections 3 and 4 respectively describe formally the components and the mechanisms through which they interact, and a discussion concludes the paper.

2. Overview

2.1. Abilities, tasks and recipes

Information is a core concept of our framework. The means to produce or update a given information are encapsulated into a component called *ability*. An ability does not expose directly the information it encapsulates: it exposes ways to retrieve it (in read-only mode), and a set of free variables that represent some states of the associated information, which can be constrained. For instance, such a constraint can be the date of the information provided by a sensor, or a wished robot state for the information provided by a planner (*i.e.* a plan).

[3]The static information, that mainly represent the various robot characteristics and capacities, are mostly used for planning and are not explicitly considered in the framework.

Abilities can exchange asynchronous messages, which define constraints on the abilities free variables. This communication uses a two-way channel, so that the sender is informed whether the receiver has handled or not the constraint: such messages implement the control flow among abilities. Another kind of communication implements the data flow: here a single-way asynchronous channel is used to transfer the information between abilities (communications are depicted more formally in section 4.2).

When an ability receives a new constraint, it tries to enforce it. For that purpose, each ability has a list of computational processes, the *tasks*, with given pre- and post-conditions. The ability chooses the right combination of tasks to handle the received constraints.

Recipes handle the various possible strategies to fulfill a task. They deal with basic recovery error and internal details for one task. Similarly to tasks, recipes define pre-conditions, which are used to select the one that is best suited to the current execution context.

The objective of the decomposition in *tasks* and *recipes* is twofold: *(i)* reduce the complexity to select the action to perform in an *ability*, and so reduce the time to make this choice, and *(ii)* provide multiple strategies to achieve a transition from one state to another state – in particular, one can stack different strategies to handle different robots capabilities. In other words, this decomposition improves the *reactivity* of the system, and its portability over different robotic platforms.

Figure 1 illustrates the relations between the three entities abilities, tasks and recipes. An ability may have one or more task running at each time, and each task, when running, executes one (and only one) recipe.

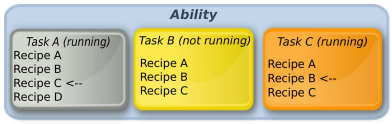


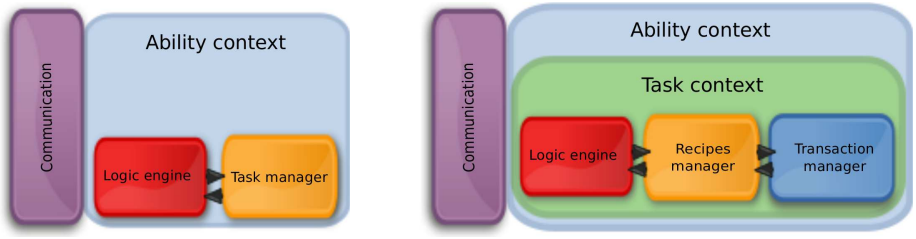
Figure 1. Abilities, tasks and recipes

2.2. Internal mechanisms

Our framework manages a network of abilities. Abilities exchange mainly some constraints on their free variables, through a *Communication* interface(figure 2(a)). When an ability receives such a constraint, it is directed to its own *Logic Engine*. Its *Logic Engine* is in charge of selecting the appropriate combination of tasks to execute, on the basis of the defined tasks for this ability, the currently running tasks, the current constraints, and the newly received constraints – those information being part of its *Ability Context*. Next, its *Task Manager* is responsible to monitor the execution of the tasks selected by its *Logic Engine*, checking in particular the tasks pre- and post-conditions. Task execution failures are handled through a dialogue between its *Task Manager* and its *Logic engine*, in order to find another combination of tasks that solves the constraints.

Tasks, when activated within a *Task Manager*, has a quite similar design. Each task has its own *Task context* which contains private variables for the task, to handle for example local errors – it can be empty. Each task also contains a *Logic Engine*,which selects the appropriate recipe (a single one), on the basis of its context, and a *Recipe Manager*,

which ensures the proper execution of the selected recipe. In case of failure or if changes occur in the task context, its *Logic Engine* chooses another recipe to execute. A correct transition is ensured by its own *Transaction Manager*, which records constraint on other abilities set by the recipe. This component is also responsible to lazily handle conjunctions or disjunctions of constraints. Figure 2(b) illustrates the interaction between these different blocks.



(a) Ability mechanisms (b) Tasks mechanisms

Figure 2. Internal mechanisms description

2.3. Supporting example

We consider an outdoor robot endowed with autonomous navigation capacities, adapted to the current context [8]:

- In rather flat or easy terrains, the navigation mode is built upon obstacle detection by a laser scanner. The mode is instantiated by a sensory-motor loop defined by four functional components *Sick*, *Segs*, *Avoid* and *Control*, that respectively gather the data from the laser scanner, fuse the data within a 2D segments-based binary obstacle map, determine instantaneous translation and rotation speeds to avoid obstacles, and drive the robot according to the defined speeds.
- In rather difficult terrains, the navigation mode is built upon 3D modelling of the terrain and path finding within this model. The associated loop is defined by the *Stereo*, *DTM*, *3DPath* and *Control* components, that respectively gather 3D data, fuse them into a digital terrain map, select an elementary trajectory corresponding to a pair of translation and rotation speeds (arc of a circle), and drive the robot accordingly.

Another component, *Nav*, is in charge of selecting the best suited navigation mode, on the basis of a dedicated *Traversability map* built upon stereovision data that assesses the terrain difficulty. This can be seen as a higher level control loop, defined by the assembly of the *Stereo*, *TravMap* and *Nav* components.

Localization is handled by a position manager component (*POM*), that gathers and fuses all the available localisation information (namely *Odometry*, visual odometry *VisOdom* and *SLAM* processes, the two latter relying on stereovision).

Finally, the destination can be either specified as coordinates, or a visually detected target, in which case the two following components that rely on stereovision are required: *detectTarget* and *trackTarget*. The related information is encapsulated in the ability *TargetPos*.

Figure 3 exhibits the various abilities defined by such a navigation solution. An example of the behaviour of our architecture is depicted in section 4.5.

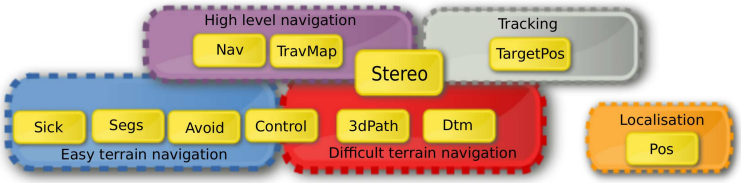


Figure 3. Abilities defined for an adaptive navigation system (yellow blocks, full line). The big blocks (dotted lines) are not explicited in the framework, and are showed for a better understanding: they exhibit in particular the possible resource conflicts on control and stereo information.

3. Formal description

Section 2 provided an overview of how information and actions are organised in the framework. We now enter into a more categorical description of each component of the architecture.

3.1. Ability description

Abilities are the main brick of the architecture. An ability is an information-centered structure, it maps one specific information. Each ability is defined by (refer to the example listing 1):

1. Its identifier, denoted by N_A (lines 1 and 12)
2. Its context: it is decomposed in three different parts *(i)* the readable variables S_r, which expose the information around which the ability is defined (line 16), *(ii)* the writable variables S_w, which are used to control the behaviour of the ability (lines 14-15), and *(iii)* the internal variables S_I, not exposed to other abilities (line 17).
3. The tasks: it is composed of two parts *(i)* a set of task identifiers denoted by S_{Ta}, (line 20) and *(ii)* a set of relations between these tasks, in particular incompatible tasks S_{rt} (lines 21-23)
4. The programming environment: it is composed of two parts *(i)* a set of exported types S_T (in sense of computer science type, lines 6-7) and *(ii)* a set of functions which can manipulate these types, and their logic relations, denoted by S_F (line 10). This programming environment is used in the recipe implementation and the tasks pre / post-condition definition.

Listing 1: Description of the abilities “pos” and “dtm”

```
1  Pos = ability {
2  Context
3  { currentPosition } { } {}
4
5  Type
6  { Distance = newtype double;
```

```
7     Position = struct { double x, y, yaw;}}
8
9   Function
10  { computeDistance :: Position -> Position -> Distance }
11 }
12 Dtm = ability {
13   Context
14   { Pos::Position lastMerged
15     bool isEmpty }
16   { Port map }
17   { Pos::Distance threshold }
18
19   Tasks
20   { clear, merge, panoramicView }
21   { clear <> merge
22     clear <> panoramicView
23     merge <> panoramicView }
24 }
```

3.2. Task description

A task represents a computational process, *i.e.* a process which takes some input, does some computation on the input and gives back an output – or some error. It can be represented categorically as a tuple `<S, I, St, F, Succ>`, where `S` is a start point, `I` an input channel, `St` a stop point, `F` a failure point, and `Succ` a success point. Both `F` and `Succ` can have an associated output channel. Figure 4 illustrates the task execution model.

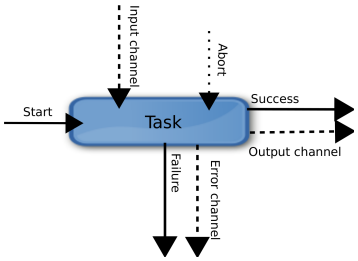


Figure 4. Computation model for a task

Listing 2 illustrates how a task is described. A task has a identifier N_T referenced by the ability that describes it (line 4 and 10), and is described by pre-condition *Pre* (lines 5 and 11) and post-condition *Post* (lines 6 and 12-13). These are logic formulas, constructed on the basis of predicates and with the classical Boole logical operators OR, AND and NOT. Predicates are based on equality or comparison of members of the ability context, or function calls defined by the ability programming environment. Finally, each task has a set of recipe identifiers, which can handle the task execution in different cases (lines 7, 14 and 21), denoted by S_{Rec}.

Listing 2: Description of three tasks “clear”, “merge”, “panoramicView” associated to the “dtm” ability (“panoramicView” is applied when the map is empty and scans the environment around the robot).

```
1   Dtm = ability { ...
2
3   clear = task {
4         pre = {{isEmpty == false}}
```

```
5         post = {{isEmpty == true}}
6         recipes = {clearMap}
7   }
8
9   merge = task {
10        pre = {{isEmpty == false}}
11        post ={{isEmpty == false}
12              { Pos::computeDistance(lastMerged, Pos::currentPosition) < threshold)}}
13        recipes = {nominalMerge, noImageInformation}
14        }
15
16  panoramicView = task {
17        pre = {{isEmpty == true}}
18        post = {{ isEmpty == false}}
19              { Pos::computeDistance(lastMerged, Pos::currentPosition) < threshold)}}
20        recipes = {nominalView, noImageInformation}
21  }
```

3.3. Recipe description

A recipe is an implementation of a task. There are three important parts to describe a recipe *(i)* its identifier N_R, previously referenced by the associated task S_{Rec}, *(ii)* its context call, *i.e.* its associated pre- *Pre* and post-conditions *Post*, built on the same model than tasks one, and *(iii)* its body or the real implementation B. The implementation is constructed on top of a specific language, which specification is out of the scope of this paper.

4. Internal mechanisms

4.1. Framework runtime

Each ability is implemented as a separate process and is registered to the *framework runtime*. The use of separate processes mitigate the failure (crash or infinite loop) of an ability as it does not impact the other abilities. The *runtime* has three main purposes : *(i)* tell how to communicate with another ability (*i.e* a nameserver function), *(ii)* give the list of currently usable activities to an ability, in order to choose a strategy and *(iii)* monitor the different abilities, using a ping protocol. When the *runtime* notices an ability failure (*i.e* the ability does not answer to ping) , it broadcasts the information to all other abilities which can react in consequence.

4.2. Communications

Communication between abilities are essential in the framework. The basic constructs are based on asynchronous constructs *send* and *receive*, similarly to the Erlang concepts [2] – that shows in particular that these two primitives are sufficient to build any communication protocol, synchronous or asynchronous.

Building on theses concepts, we introduce a higher level construct *send_constraint*. This function sends a constraint message with *send*, waits asynchronously for an acknowledgement from the receiver or fails after a timeout. After receiving the acknowledgement, the system expects two possible answers : *(i)* an OK message, which states that the constraint has been enforced by the receiver, and *(ii)* a FAILURE message, with additional information about the failure: error in the logic engine, conflict with another constraint on the system with the information about offending constraint, ...

On top of this function, we define a synchronous construct *make*, which asks and waits for an answer about the enforcement of one constraint by an ability.

To read the information associated to a remote variable, we use a proxy representation for this variable in the current ability. The proxy variable is initially fed using an asynchronous request/reply, then, the system uses a publisher/subscriber interface to update it.

4.3. Logic Engines

The Logic Engine is the component in charge of adapting the behaviour of an ability with respect to its current environment and the constraints it tries to enforce. The logic engine for an ability is only called in two different cases: *(i)* when receiving a new constraint, or *(ii)* when a task fails, in which case the logic engine tries to find another task combination to enforce the constraint.

To match a constraint with a pre or post-condition, a unification process is used along with a comparison with a set of logic rules. The set of logic rules is selected depending on the type of data manipulated (this is similar to *concept* notion in C++0x or *typeclass* notion in Haskell). Currently, we introduce the following variable types :

- *Equalable* : the only predicate available on variables of this kind is $=$ (and naturally $\neq$).
- *Comparable* : this type of variables can be compared with $<$ (and other derivatives). A *Comparable* type is of course *Equalable*, and can be compared to a range of values. Date or duration are two variables of this type.
- *Set* : one can test the membership to a set, and all the classical operations on a set (union, intersection, etc...). A basic example is the abilities type.

For functions, only the geometric type is considered for now. Into this family, there are various relations between the functions. For example, if one has the two functions *isInCircle (Center, Radius)* and *isInSquare (Center, Size)*, *isInSquare* implies *isInCircle* if $CenterSquare = CenterCircle$ and $Radius >= Size \times \frac{\sqrt{2}}{2}$). One can define rules for each specific domain with similar mechanisms.

Using this mechanism to match a rule with pre and post-conditions, classical search and backtrack processes are used to select an ordered set of tasks to execute. At each step i, the engine searches for all the combination of tasks which post-condition match the new constraint, removing the ones contradicting the currently running ones (according to S_{rt} specification). The combination of tasks S_i is sorted using three criteria: *(i)*: number of pre-condition not fulfilled *(ii)* number of tasks involved *(iii)* number of tasks not runnable from S_{rt}. The algorithm stops when all pre-conditions are fulfilled. If not, the algorithm continues with the pre-condition not fulfilled of the first element of S_i. It backtracks if S_i is empty, removing the first entry from S_{i-1}, and searching with the new first element from S_{i-1}. The result of the algorithm is an ordered set of tasks, where tasks from a same level i can be run in parallel.

The Logic Engine used in each task to select the recipe to apply uses a similar matching mechanism. However, there is no backtrack processing, it just searches the recipe matching the current environment. If there are no available strategy for this situation,

the task fails, and the constraint has to be enforced in another way, using if possible another task combination. If no other combination is possible, the ability can not fulfill the constraint.

4.4. Transaction Manager

The last important element of the architecture is the transaction manager. As stated in section 2.2, it has two main purposes : *(i)* handle correctly disjunction and conjunction of conditions and *(ii)* handle correctly transitions from one recipe to another.

The *make* construction tries to enforce a constraint on a remote ability, in a synchronous way. It is extended to an arbitrary complex combination of constraints, in particular to handle a conjunction or a disjunction lazily: if one clause fails in the conjunction case, or is satisfied in the disjunction case, the other clauses can be aborted, as their results will have no further impact.

The extended *make* is implemented with a call to the transaction manager. The transaction manager references the operation and uses the *send_constraint* to send the individual constraint. On reception of an answer, it tries to reduce the logic expression, aborting the clauses that are not anymore necessary. If the expression is reduced to $True$ or $False$, the transaction manager gives the answer to the recipe, with additional information in case of failure.

The other purpose of the transaction manager is to handle transitions from one recipe to another recipe. In particular, one recipe does not know what recipe previously runs for one task, and so does not know which constraints has been requested. It can lead to dangling constraints on the system. To deal with this issue, the *Transaction Manager* registers all the constraints requested by the recipe. One easy solution is to clear all the registered constraints, when switching. However, in some case, it is not a really efficient solution because the two recipes are really similar, and so can benefit from the previously requested constraints (and it is often the case when two recipes are the incarnation of the same strategy, but deals with different error case). In this case, the best solution is to not clear anything: when the recipe asks for an already registered constraint, the transaction manager does not send anything, and just waits for answer to its previous request.

To achieve this, each recipe is classified into a family. In the case of a transition between two recipes of the same family, the transaction table is not cleared. In other case, it is cleared to have a clean environment.

4.5. Illustration

This part illustrates the behaviour of the framework for a specific task, the navigation on difficult terrain, as presented in section 2.3. For this purpose, we use the recipes described in listing 3. The construction *ensure* is a continuous *make*, in that it continues to enforce the constraint as long as it is not aborted.

Listing 3: two recipes for navigation on difficult terrain

```
3DPath_reach_goal = recipe {
        Pre = {{no_failure == true}}
        Body = {
                ensure ( pos::computeDistance(Dtm::lastMerged, Pos::currentPosition) < threshold)
```

```
                        && 3DPath::goal == currentGoal
                        && Control::tracker == 3DPath::plan)
                wait(pos::computeDistance(goal, Pos::currentPosition) < goalThreshold))
        }
}

3DPath_handle_failure = recipe {
        Pre = {{ 3DPathFailure == true }}
        Body = {
        /* Get some new information */
        make ( pos::computeDistance(Dtm::lastMerged, pos::currentPostion) == 0.0) }
        /* Restart the classic processing */
        ensure ( pos::computeDistance(Dtm::lastMerged, Pos::currentPosition) < threshold)
                 && 3DPath::goal == currentGoal
                 && Control::tracker == 3DPath::plan)
        wait(pos::computeDistance(goal, Pos::currentPosition) < goalThreshold))
}
```

The *3DPath_reach_goal* recipe is applied to reach a goal. The construction *ensure* leads to a call to the *Transaction Manager* with the associated conjunction of constraints, which asks in parallel the three abilities *Dtm*, *3DPath* and *Control* to enforce their respective constraint. Let us check what happens in *Dtm* (see listing 2 for task description): depending on whether the map is empty or not, the *Dtm* logic engine launches the *panoramicView* or the *merge* task, because both fill the constraint. The choice has no consequence, as the only important thing is that the map is filled on a regular basis.

The processes pursue, and *3DPath* fails to plan a way to go to goal. The *Transaction Manager* receives a failure and tries to reduce the expression: since it is a pure conjunction, the result of the full expression is false. So it aborts the two remaining constraints, and marks in the task context that 3DPath failed: the *Logic Engine* decides to execute the *3DPath_handling_failure* recipe.

Now the *Dtm* component fails. Unfortunately, no more strategy are available to handle this case, so the task fails. A possible solution is to compute a new global plan using *Nav* planner, and then to try to enforce the new computed path...

It is possible to write such a process control script by hand. But it is hardly tractable *(i)* to write it correctly, as the complexity explodes with the combinatorial of the variable number, and *(ii)* to maintain and extend it to deal with new robot capabilities. On the contrary, the proposed framework is declarative and composable: one does not need to know the internal status of a task, nor the exact sequence of tasks that must be executed. The programmer only describes the relations between the information, and the different part of the framework enforces, as showed, the proper behavior of the system.

5. Discussion

We have presented the design of a framework to control complex robotic systems, that exhibits several important features. First, the decomposition of the system in abilities centered on information is not domain-specific: if the mission of a robot changes, there is no need to rewrite the whole system, but only to introduce new information or new relations between the information, without breaking the existing ones. Second, the strict interface of each ability and the use of logic engines allow a safe exploitation of each information: in a general way, the developer does not need to take care about the internals, but to just declare the required information for the ability, and the system provides it if possible. Moreover, logic engines allow the system to handle various situations and

resource conflicts, without explicit handling by the developer. Finally, the framework is quite robust with respect to component failures. Using one process by ability, the failure of an ability does not lead to a complete failure, but to a reconfiguration of the framework, considering the offending one as "not-available".

Even though the proposed concepts have been illustrated using a rover navigation case, the decomposition into abilities is relevant for other robotic missions. In particular, the fact that the decomposition is centered on information makes it relevant for *interactions* :

- In human / robot interactions, the robot must be aware of the human situation, actions or will. These information are required for various human / robot tasks, to plan the robot motions considering human activities, to plan physical interaction to exchange objects... This requires a fine granularity in information decomposition, and a good handling of conflicting information needs: we believe the proposed framework is well suited for these requirements.
- In multi-robot systems, the decomposition into abilities can be a sound basis to allocate tasks among the robots (*e.g.* within a market based approach). A straightforward extension of the framework would be to endow a robot with the knowledge of the others' abilities, thus yielding the establishment of cooperating schemes in a rather transparent way.

Ongoing work include a more robust implementation of the current framework, and putting efforts on making it more verifiable and applicable to multi-robots scenarios.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17, 1998.

[2] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. Concurrent programming in erlang - second edition, 1996.

[3] D.E. et al. Bernard. Design of the Remote Agent experiment for spacecraft autonomy. In *IEEE Aerospace Conference*, 1998.

[4] T. Estlin, R. Volpe, I. Nesnas, D. Mutz, F. Fisher, B. Engelhardt, and S. Chien. Decision-making in a robotic architecture for autonomy. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 2001.

[5] Erann Gat. On three-layer architectures. In *Artificial intelligence and mobile robots*, pages 195–210. AAAI Press, 1997.

[6] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *In IEEE International Conference on Robotics and Automation, Mineapolis*, 1996.

[7] A. Jonsson, Vandi Verma, Corina Pasareanu, and Michael Iatauro. Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations. In *AIAA Space Conference*, 2006.

[8] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila. Autonomous rover navigation on unknown terrains: Functions and integration. *International Journal of Robotics Research*, 21(10-11):917–942, Oct-Nov. 2002.

[9] C McGann, F. Py, K. Rajan, and A. G. Olaya. Integrated Planning and Execution for Robotic Exploration. In *International Workshop on Hybrid Control of Autonomous Systems*, 2009.

[10] C McGann, F. Py, K. Rajan, Hans Thomas, Richard Henthorn, and Rob Mcewen. A Deliberative Architecture for AUV Control. In *IEEE International Conference on Robotics and Automation*, 2008.

[11] N. Muscettola, G.A. Dorais, C.F.R. Levinson, and C. Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. In *International NASA Workshop on Planning and Scheduling for Space*, 2002.