ROAR: Resource Oriented Agent Architecture for the Autonomy of Robots

Arnaud Degroote and Simon Lacroix[⋆]

Abstract. This paper presents a multi-agent system 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 proposed architecture relies on a partition of the *decisional layer* in separate *resources*, each one managed by a specific agent. The architecture of *resource* agents and their interactions to guarantee a coherent system are depicted.

1 Introduction

Besides progresses in the robotic functional layer, whether on algorithms or in architecture with some well-understood component architecture like GeNoM or ROS, it is the *assembly* of theses components 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 in the roboticists community 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

Arnaud Degroote · Simon Lacroix
CNRS; LAAS; 7 avenue du colonel Roche, F-31077 Toulouse,
Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
e-mail: `{arnaud.degroote,simon.lacroix}@laas.fr`

[⋆] This work has been partially supported by the DGA founded Action project – action.onera.fr.

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 and plan. Several languages have been proposed in the literature, to ease the implementation of this layer, as PRS [6] for the LAAS architecture [1], or TDL [10] for the *Remote Agent* system [3].

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 that coexist in the different layers. This discrepancy of representations makes the diagnostics of plan failures difficult, because the planner does not have relevant information about the failure causes, and it hinders the efficiency of the 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 [9] and T-REX [8] define 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 consistently interleaved. Moreover, during execution the different agents are synchronised to maintain a consistency of the global plan. T-REX goes further by proposing some systematic formulation to synchronously exchange states between these agents.

Another issues of a three-level architecture are their lack of modularity (decisional and execution layers are two separate "monolithic" blocks, and changes in their model often leads to heavy side effects) and scalability (as the deliberation time increases exponentially with the number of robot functionalities). Mc Gann *and al* [7] 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. However, their partition is constructed by the programmer, based on the mission needs. If the nature of the mission changes, the whole partition must be reorganized: this kind of construction does not scale well over a large variety of missions, missing the objective of a versatile architecture.

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

A key feature of autonomy is the ability to properly react to unpredicted events or situations (though handling correctly *any* situation remains a challenge): such events should be asynchronously treated as they occur, and each agent must select the most suitable strategy, on basis of its model and its knowledge of this event.

Finally, the architecture must be *robust to failures*: in case of an agent failure due to a logic or programming error, or to a physical failure, the framework must pursue its operation if possible, using alternative strategies to handle the mission.

Overview. We propose in this paper the definition of ROAR[1], an architecture based on a partition scheme that aims at fulfilling these *composability*, *reactivity* and *robustness* requirements.

We follow the decomposition principle proposed in IDEA or T-REX , but contrary to these architectures in which the decomposition is defined according to a set of tasks, our proposal is to decompose the robot abilities into a set of separate *resources*. The term "resource" has to be understood in its most general sense here: a resource can be a physical resource, an information resource or a planification resource. Each resource is embedded within a separate agent, a *resource agent*, which is responsible of the consistency and the good use of the resource. The decomposition in resources still must be done by a domain's expert but depends only on the domain, and not a specific robot or functional layer.

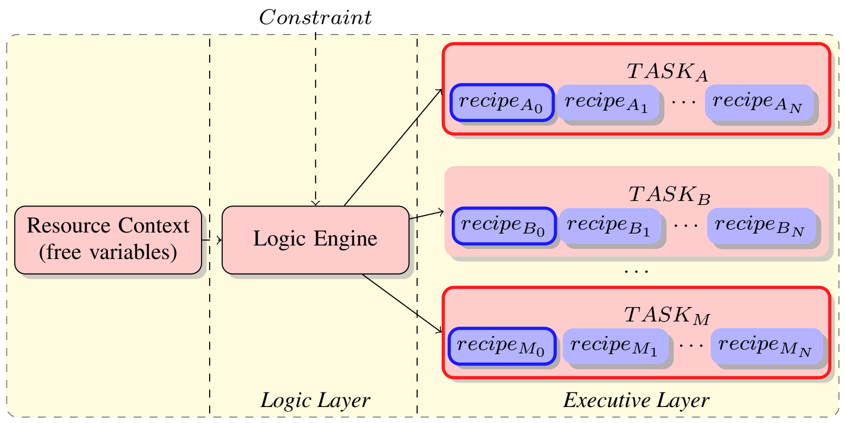
Resource agents do not expose directly the resource they embed, but a list of different points of control, which are called *free variables*. Other resource agents ask for specific behaviour of one resource by constraining these *free variables* (*i.e.* binding these variables with specific values). The constraints between agents form a dynamic directed acyclic graph, where vertices are the agents, and the edges are the constraints between agents at a time t. The ROAR framework is in charge of maintaining this graph, ensuring that the required relations are satisfied. For this purpose, each agent is endowed with a solver that locally enforces the constraints set on it. 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 agent: the system adapts the information graph to handle the problem at hand, and the different logic solvers locally schedule the access to the resources.

2 Resource Agents

Each agent only exposes a set of *free variables* to specify modifications on the state of the resource it embeds, and is structured along a 2-tier approach (figure 1): upon reception of a new constraint, the *logic layer* states if it can be enforced on the basis of the current *agent context*. If yes, it selects a set of tasks to achieve the transition from one logic state to another one. For each task, the *execution layer* selects a recipe to achieve it on the basis of current *task context*.

Agent Logic layer. On reception of a new constraint message, a resource agent needs to decide if the requested constraint is compatible with the current logic state of the resource. This can be done with a Finite State Machine for simple resources

[1] ROAR stands for "Resource Oriented Agent architecture for Robots".



$Constraint$
Resource Context (free variables)
Logic Engine
$TASK_A$
$recipe_{A_0}$ $recipe_{A_1}$ $\cdots$ $recipe_{A_N}$
$TASK_B$
$recipe_{B_0}$ $recipe_{B_1}$ $\cdots$ $recipe_{B_N}$
$\cdots$
$TASK_M$
$recipe_{M_0}$ $recipe_{M_1}$ $\cdots$ $recipe_{M_N}$
Logic Layer
Executive Layer

Fig. 1 Global mechanism for a resource agent. The surrounded boxes are the tasks / recipes currently selected by the logic layer/execution layer.

with few *free variables*, but the complexity of the state machine exponentially grows with the number of variables, making it difficult for the developer to guarantee its consistency or to update it after modifications of the *free variables* interface. The programming language community has proposed several languages to solve this kind of problem according to a *logic paradigm*. We propose to use such a paradigm to declare and decide if, in the current context, a constraint is enforceable. The part of the system which handles this is called *Logic Engine*.

The logic layer of an agent is the set of the resource specific legal transitions, described by their name and their pre- and post-conditions. The post-conditions are predicates that are true at the end of the transition, and the pre-conditions are predicates which must be true before the beginning of the transition. Considering the current facts and the required constraints, the *Logic Engine* chooses the combination of transitions to fulfill the constraints or reject them if it can not find any transition. This transition between two states is called a *task*.

Since the notion of resource is robot-agnostic, the notion of transition between two states for a resource is also robot-agnostic. This layer is therefore generic, and reusable for different robots. This layer is also essential to guarantee the consistency of the resource – as long as the developer does not make errors in its specification.

Agent Execution layer. *Tasks* describe the possible transitions between two logic states, they do not describe how to handle them: their execution is handled by some *recipes*. The objective of the decomposition in *tasks* and *recipes* is twofold: *(i)* reduce the complexity to select the action to perform in an agent, 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.

Recipes are described as a set of pre- and post-condition, similarly to *tasks*, and a body, which contains the real description of the new behaviour of the agent. The body is mainly constructed on top of two constructions, *make* and *ensure* which asynchronously send a constraint message, respectively for discrete and continuous constraints. For conjunction or disjunction of predicates, they dispatch in parallel the constraints to the different agents, and will evaluate lazily the result. The layer evaluates the pre-conditions of each recipe, and execute the most adequate one (the one with the largest number of satisfied pre-conditions).

Error handling. Errors or unexpected events often occur during robotic missions, and it is essential to correctly handle them to successfully achieve the autonomous mission. In the three layer architectures, there are several strategies to handle errors:

- exception handlers, as proposed by Simmons in [10], are piece of application code responsible to handle a specific error. Our framework proposes this approach too. As said previously, recipes are selected on the basis of their pre-conditions. To handle specific error E, we just need to add some recipes with pre-condition check for this specific error.
- the plan has computed several strategies for one task. In ROAR , each task can be handled by multiples recipes. If one recipe fails because of specific agent A, the system can choose another recipe, not using A, to achieve the task.
- the planner is able to repair its plan. In our architecture, it means that the agent can try to choose another combination of tasks to handle a constraint.

All these solutions are local to an agent, and are the preferred method to handle a problem. However, if the agent cannot find a solution by itself, the failure goes up in the agent graph, with a full *error context* (*i.e.* the list of agents that fails, with the associated constraints set on them during the failure). At each step, the system tries to find an alternate strategy using the methodology described previously. When executing this new strategy, the *error context* is passed to each agent, so they can decide of their local strategy, knowing the history of the global task. This improves the behaviour of the whole system, avoiding to use a path which leads to failure.

3 Discussion

We have presented the design of a framework to control the runtime configuration of complex robotic systems. Even if the decomposition in several layers in each agent is similar to the one presented in three-level layer architecture, the whole system is decomposed in several agents: in this way, the system is more reliable (no single point of failure), and scales better, as computations are split by each agent and not handled by the complete system (reducing the complexity at the expense of the optimum). It is modular, and the linked model for executive and planning allows better error handling. In comparison of the T-REX architecture, the use of an asynchronous model yields more reactivity, and it eases network transparency, *i.e.* the seamless use of agent on remote machines or robots. The strict decomposition in resources also make the system *more composable*.

Various language based solutions to control autonomous robots have been proposed, *e.g.* PRS [6], via TDL [10], a C++ extension with (parallel) task semantic. All these languages can express some high-level tasks, but we think that they lack the possibility to precisely deal with resource conflict. PRS may be the better alternative with respect to this, but lacks modularity and robustness. Our work is grounded on some robust and concurrent language like Erlang [2] to provide these features, and provides some methodology to ensure the composability of the framework.

The overall implementation of the framework is still partial, but is however already integrated within two of our robots. We now focus to make the implementation more robust. Our compiler generates some high-level C++ from the specification language. One of the motivation to targeting a high level language instead of a custom runtime is, apart from development time, the possibility to easily integrate with third party libraries without the need to define dedicated interfaces.

Ongoing work include putting efforts on making it applicable to multi-robots scenarios. Another of our goals is to make the system valid by design, *i.e.* to have some guarantees about execution of each agent and about their interaction, in particular avoiding deadlocks situations between different agents.

References

1. Alami, R., Chatila, R., Fleury, S., Ghallab, M., Ingrand, F.: An architecture for autonomy. The International Journal of Robotics Research 17 (1998)
2. Armstrong, J., Virding, R., Wikstrom, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. (1996)
3. Bernard, D., Dorais, G., Fry, C., Gamble Jr., E., Kanefsky, B., Kurien, J., Millar, W., Muscettola, N., Pandurang Nayak, P., Pell, B., Rajan, K., Rouquette, N.: Design of the Remote Agent experiment for spacecraft autonomy. In: IEEE Aerospace Conference (1998)
4. Estlin, T., Volpe, R., Nesnas, I., Mutz, D., Fisher, F., Engelhardt, B., Chien, S.: Decision-making in a robotic architecture for autonomy. In: Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation in Space (2001)
5. Gat, E.: On three-layer architectures. In: Artificial Intelligence and Mobile Robots, pp. 195–210. AAAI Press, Menlo Park (1997)
6. Ingrand, F.F., Chatila, R., Alami, R., Robert, F.: PRS: A high level supervision and control language for autonomous mobile robots. In: IEEE International Conference on Robotics and Automation, Mineapolis (1996)
7. McGann, C., Py, F., Rajan, K., Olaya, A.G.: Integrated Planning and Execution for Robotic Exploration. In: International Workshop on Hybrid Control of Autonomous Systems (2009)
8. McGann, C., Py, F., Rajan, K., Thomas, H., Henthorn, R., Mcewen, R.: A Deliberative Architecture for AUV Control. In: IEEE International Conf. on Robotics and Automation (2008)
9. Muscettola, N., Dorais, G., Levinson, C., Plaunt, C.: IDEA: Planning at the Core of Autonomous Reactive Agents. In: International NASA Workshop on Planning and Scheduling for Space (2002)
10. Simmons, R., Apfelbaum, D.: A task description language for robot control. In: Proceedings of the Conference on Intelligent Robots and Systems, IROS (1998)