# A resource-agents robot control architecture: design and formalization

**Abstract.** This article introduces a robot control architecture that explicits the management hardware and information resources, and sketches a formal definition. The architecture is defined by a network of agents, that interact through the exchange of constraints on the resource they embed. The agents are composed of a logic layer, that selects and configure the tasks to achieve to satisfy the constraints set on the agent, and an executive layer, that is in charge of the proper achievement of the selected tasks. The inter-agent mechanisms that guarantee the good use of each resource, solve resource conflicts and handle execution errors are detailed. The article then introduces a formal description of the proposed architecture, that expresses the various logic and execution processes involved using the Concurrent Logic Framework. This formalization aims at establishing various properties, in order to assess a guaranteed behavior.

## 1 Introduction: on control architectures

The robotic community has produced numerous achievements in a wide spectrum of functionalities (perception, planning, control, learning ...), but yet autonomous *versatile* robots are not a tangible reality. Robot autonomy not only requires a collection of smart functionalities, but also a proper *assembly* of these functionalities. For this purpose, several frameworks have been proposed in the literature to encapsulate computation and communication processes, and it is now admitted that component-based frameworks provide good modularity and reusability properties (significant examples are GenoM [1], OROCOS [2] or ROS [3] – see *e.g.* [4] or [5] for a survey). But while they allow to assemble functionalities, these frameworks do not convey principles to *control* these functionalities, *i.e.* to configure, schedule, trigger, coordinate and monitor the associated processes. This is the role of a *control architecture* to structure and control the available functionalities, according to the context, in order to autonomously achieve missions [6, 7]. A control architecture should be designed in order to endow the robot with *(i)* the capacity to achieve a variety of high level missions, and *(ii)* the capacity to cope with a variety of events which are not necessarily a priori known, in a mostly unpredictable world. Besides these two general characteristics of autonomy, a control architecture must satisfy the following requirements:

- Reactivity and deliberateness. Robots evolve in dynamic environments and must react accordingly. Meanwhile, they must be able to reason on models to plan tasks, especially to achieve missions that involve complex behaviours, dealing with time constraints and payload management, etc.
- Modularity and composability. For versatile robots to be able to handle various missions, the architecture must be modular and composable: introducing new functions or new behaviours should not break the existing assembly, nor require any major rewriting of its definition. The enabling and disabling of functions at runtime should also be supported. Component-based development environments provide this composability property, but at the lower functional level: this property should be extended to the overall control architecture.
- Concurrency. Robots are highly concurrent systems, within which multiple tasks run in parallel, accessing and modifying a finite set of resources. They must be able to reason about their global state and the mission at hand to ensure a consistent and efficient behaviour, and in particular to avoid resource conflicts.
- Robustness. Most of the functionalities a robot is endowed with can fail. Whatever causes the failures, the robot should be able not only to detect and identify them, but also to recover them when possible.
- Validation. Robots are complex systems that interact with a complex environment. It is necessary to know the answer to some correctness and safety questions such as "will the robot *always* stop before hurting someone?". Nowadays, developers mostly rely on testing to assess such properties. But testing only increases the confidence on the robot behavior, and cannot *prove* any property: a formal validation is necessary for this purpose.

Finally, the architecture must ease the task of the robot programmer. In particular, preventing resource conflicts and defining error recovery procedures rapidly becomes cumbersome as the complexity of the system increases: providing means to handle this complexity is essential.

This article introduces ROAR (Resource Oriented Architecture for Robots), an architecture which aims at satisfying these requirements. ROAR proposes a control architecture decomposed into a network of agents, each one encapsulating one *resource*. The activities of an agent encompass both deliberation and execution. They are driven by constraints set on the agent, and ruled by a formalized logic engine. The overall behaviour of the system is defined by the interactions between the agents, which in particular allow to handle resource conflicts and to recover execution errors.

**Outline.** The article is decomposed in two parts. The first part describes in a precise way the ROAR architecture. Section 2 briefly reviews the main existing robot control architectures, and analyses to what extent they satisfy the above requirements. Section 3 presents the rationale of the ROAR architecture and its design principles. Section 4 depicts the design of a resource agent and its internal deliberation and execution mechanisms, and section 5 depicts how agents interact to lead to a global coherent behaviour, robust to errors and concurrent accesses to resources.

In a second part, we focus on the encoding of ROAR in a logical formalism. Section 6 briefly reviews encoding and validation proposals for robotics system, and the solutions to formalize concurrent systems. Section 7 recalls the basics of the concurrent logic framework, which is used in 8 to formalize the ROAR architecture. A discussion concludes the article.

# Part I
# A resource oriented robot control architecture

## 2 Related work

The first architecture paradigm for the control of autonomous robots consisted in splitting the activities within three sub-systems according to the sense-plan-act scheme (SPA architecture) [8]. The sensing system derives environment models from the perceived data, the planner generates a plan to achieve the goal, the plan execution is handled by the execution system, and the overall loop is iterated until the goal is reached. But the model generation and planning are hard problems, and consequently the loop can become very slow, limiting the possibility to behave in an uncertain and dynamic world – as argued in [9]. Two different paradigms then emerged, often referred to as "reactive architectures" and "layered architectures". Reactive architectures reject the classic AI paradigm, and suggest that autonomy arises from the interaction of elementary behaviours – one of the most known being the subsumption architecture [10]. Following these precepts, different multi-agents systems appeared, focusing on the interactions between agents [11, 12]. The second paradigm is rather centred on symbolic planning. It introduces an intermediate layer between high-level planning and reactive behaviours ("3-layer architectures"), that handles the realisation of the plans, while preserving the capacity to react to the current situation [9, 13, 14]. These architectures raise two issues: first, each layer uses different information representations, which leads to inefficiency execution or error recovery. Second, due to their monolithic approach, each layer becomes less and less efficient when the complexity of the robot grows. More recently, "2-layer architectures" have been introduced [15, 16] to tackle these issues, using a set of agents defined along a uniform representation between executive and planning processes. But the modularity of these approaches is limited by their requirement on a strict order between agents.

## 3 Approach: a graph of resource agents

The principle of *partitioning* the robot functionalities into a set of components is essential to simplify the overall system control and keep reactivity. To define the partition, one must address the following issues:

- how to decompose the system into components, while preserving the modularity and allowing parallelism?
- how to organize the components: what information are exchanged among them, how conflicts and errors are handled?

In ROAR , the components are agents that encapsulate one *resource* and its associated logic. Formal rules describe the evolution of the system, the interactions between agents, and the means to handle resource conflicts and error recovery.

**From actions to resources.** There are three different ways to define an action of the robot. The simplest case is an elementary action, that has a direct implementation in the functional layer. The second case is a sequence of elementary actions (possibly with conditionals), that are simply executed step by step. The last case is probably the most frequent: an action is a plan defined by a specialized planner. The generated plan can be a set of actions (recursively defined) or a special construction which is executed by a specific component of the functional layer (*e.g.* a trajectory follower). Sequences and plans can be described as a tree, where leaves are elementary actions or sequence of actions, and nodes are supervision processes of small sense-plan-act loops. On board a robot, several actions can be executed in parallel, leading to a forest of actions, with possible overlaps between the different trees, *i.e.* concurrent accesses to some subsystems.

Actions are naturally the key elements to consider within a control architecture, which purpose is to select, trigger and control them according the current context and the mission at hand. But the usual SPA loop is often implicitly understood sequentially in the sense-plan-act order, whereas we view it the other way around: actions needs to be planned or decided, and plans or decisions require information, the information on the environment being provided by perception. According to this, we envision the activities of the robot as *constraints propagations on resources*. The term "resource" is understood in its most general sense: a resource can be a physical resource, an information resource or a planification resource. For example, to reach a given position, a mobile robot must ensure that it has an environment model (which can require data acquisition), on which it regularly plans a trajectory (the plan is an information), and finally it exploits the locomotion motors (a physical resource) to achieve the trajectory. The overall `GoTo(x,y)` action can also be viewed as a constraint on the robot position to satisfy.

This decomposition upon resources has several advantages over a classical task-oriented decomposition. First, it maps more directly the functional layer, which is mostly data-driven. Second, the concurrency problem is related to resources, not to tasks: expliciting resources eases the reasoning about them. Finally, defining relations between resources is more declarative and generic than explicitly defining the steps necessary to provide the resource: for instance, if a 3D-points cloud is required, one does not need to specify whether it is produced by a lidar, stereovision or a Kinect sensor.

**Resource agents.** In ROAR , a robot is modelled as a set of resources, each one being encapsulated by a single agent. These agents form a graph through the exchange of messages that encode constraints. The definition of these constraints must be as expressive as possible for the developer, while allowing to reason about their implications: we propose to represent them using logic formulae. In presence of multiple concurrent constraints on the same resource agent, the agent can decide with classical logic if it can enforce them, or if it needs to reject some. This guarantees the correct access to the resource in a deterministic way, and keeps the system openness: whenever an agent is added or activated, if it tries to access to an already used resource, its request will be rejected (with some informative context) and the system will continue to operate.

Once an agent has decided to enforce a constraint, it must find a way to execute it, considering its internal state and its environment. A situation is defined as a set of logic propositions, and the execution recipes are parametrized by a situation. The agent evaluates the situation to decide which method it will select to enforce the constraint. The situation description yields a good expressiveness for the devel-

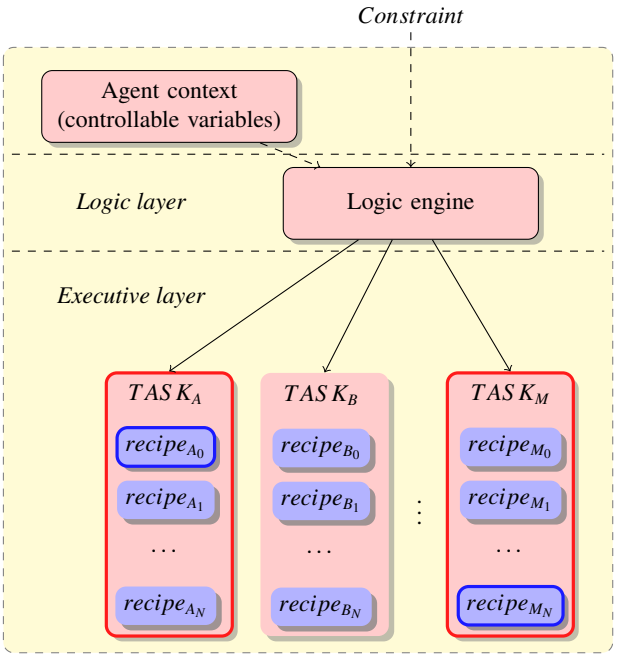


**Figure 1.** Overall structure of a resource agent. The thick lined tasks are the one currently selected by the logic layer, the thick lined recipes are the ones selected by the execution layer – several tasks can be activated in parallel, whereas only one recipe is active for each active task.

oper, it is syntactically coherent with the constraints formulation, and it allows a deterministic behaviour of the agent.

**The ROAR framework.** To sum up, the robot is modelled as a dynamic graph, where node are agents encapsulating one resource, and edges are the constraints between these agents at 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 and controls concurrent accesses. Any problem (concurrency issue, failure in the functional layer, unexpected situation) is asynchronously treated as it occurs: if it cannot be handled by the involved agent, the failure goes back through the graph until it is solved by a fixed policy, or in the last resort by a human operator – if any.

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 constraint solvers locally schedule the access to the resources. ROAR allows a coherent reasoning on resources, and considerably eases the definition of supervision schemes.

## 4 A ROAR agent

This section depicts the inner architecture of a resource agent: it introduces the syntax used to describe the agents and the associated algorithms.

### 4.1 Overview

A ROAR agent is composed of three parts (figure 1): a *context* which is exposed to other agents, and that in particular specifies the constraints the agent can enforce, a *logic layer* that selects the appropriate tasks to enforce these constraints, and an *executive layer* that is in charge of selecting a recipe to achieve each of the selected tasks.

Constraints are received through messages emitted by other agents. Upon reception of a new constraint, the logic layer assesses whether the constraint can be enforced or not: if yes, it selects the tasks to activate to enforce it. The logic layer ensures the coherency of the internal agent state. This state is a logic state: it is defined as a set of propositions, and a task is a transition between two logic states. The task specification is not exposed to other agents, so it can be modified without impacting them. Moreover, it only relies on resources and logic formulae, and does not depend on details of the functional layer: it can therefore be reused in different robot contexts.

Tasks describe the possible transitions between two logic states, not the way to effectively achieve these transitions: this is handled by the executive layer, which selects an appropriate *recipe* for each task to achieve. Developers can implement multiples recipes to handle one task, to allow different behaviour depending on the agent state, the available agents, ... Note that at this level robot or middleware specific functions can be used, in which case recipes may not be directly transferable between different platforms.

### 4.2 Agent description

The ROAR framework exposes basically the default data types `bool`, `string`, `int` and `double`, and equality and comparison propositions. Each agent can extend the interface, by proposing new product type (similarly to the `struct` construction in `C` – for example, one agent can expose the type `point`, which is composed of three `double`). Moreover, it can expose some functions which manipulate such types and some associated logical rules. For example, one can declare the function `distance` which takes in arguments two points, and returns a `double` and states the symmetry rules for `distance`:

$$\forall A, B : point\ distance(A, B) == distance(B, A)$$

Each agent expresses its context, *i.e.* a set of typed variables which represents its state. The context is divided in three different sets, each

one corresponding to a different access policy. The controllable variables $\{C_v\}$ are readable and writable by any other agent, and define the primary interface to control an agent. $\{R_v\}$ represents the set of variables readable by any agents, and describes the main information (or reference to) encapsulated by the agent. Last, $\{P_v\}$ defines the internal states of the agent, and so other agents cannot access it.

## 4.3 Logic layer

A task is defined as a transition between two logic states. It does not have any implementation, but only declares requirements and effects. From a developer perspective, a task can also be viewed as an interface with a certain contract, as in [17], *i.e.* some pre- and post-conditions. Contrarily to a classic notion of contract, they have here an active role, as they are used to decide if a constraint can be handled, and how.

To decide if a task $T$ can handle a specific constraint, the logic layer tries to solve the following Horn clause:

$$Post_1^T \wedge Post_2^T \wedge \cdots \wedge Post_i^T \wedge R_1 \wedge \cdots \wedge R_i \rightarrow C$$

where $C$ is the constraint to handle, $Post_i^T$ represents the $i^{th}$ post-conditions of the task $T$ (containing no free variables, nor universal quantification), and $R_k$ represents the rules associated to the domain of the constraint (for example, if a task contains a reference to a point, the rules associated to distance will be used). If the clause can be directly proved, it means that the task $T$ matches the constraint. In this case, the system checks the pre-conditions of task $T$, adds all failed pre-conditions to the list of constraint to handle, and recursively tries to find a solution to these constraints using a parallel first-depth search. If there is no direct solution, but a solution under certain hypotheses, these hypotheses are added to the list of constraint to handle. Moreover, at each step, the agent removes from its search space the selected task, and the tasks incompatible with it[1]. In this way, the process is guarantee to terminate, as the number of tasks is finite, and decreases at each step.

Once the agent finds a complete solution, it commits its task tree in the agent context, *i.e.* it hides the tasks required to fulfill the constraints (and the tasks incompatible with them) in the agent context. This ensures that other constraints cannot trigger a task incompatible with the ones used to handle the current constraint. If the agent fails to commit the task tree (when some of the required tasks happen to be not available anymore), the process restarts, using the new list of tasks.

## 4.4 Executive layer

A *recipe* describes a way to achieve a task. While tasks can be viewed as an interface with a precise contract, recipes can be viewed as a specialisation of this interface, with a possibly stricter contract (*i.e.* more pre-conditions). It is defined by a situation (or pre-conditions) *i.e.* the valid states in which the agent can execute it, and a body (or implementation) *i.e.* a sequence of (possibly parallels) actions which describe what to do to achieve the task.

**Recipe syntax.** Two important primitives allow to exchange constraints between agents: *make* and *ensure*, respectively with synchronous and asynchronous behaviours. Informally, *make <predicate>* sends the *predicate* constraint to a remote agent, and waits for its answer (success or failure), while *ensure* returns directly the constraint identifier and assumes that the constraints holds until it is aborted ("manually" with the *abort* primitive, or automatically when the recipe ends). The recipe fails if one of its *ensure* primitive fails. The *Wait <predicate>* primitive allows to block the recipe execution until the predicate becomes true. Last, *let* permits to introduce a new local variable in the scope, while *set* allows to modify the *agent context*.

**Recipe selection.** Several recipes can be associated to a given task, to allow the system to adapt at best to the current situation. The actual recipe to activate must therefore be selected within the ones that are acceptable.

A recipe $r$ is an acceptable choice if:

$$domain(r) \subset available\_agents$$

$$all\ preconditions\ of\ r\ are\ evaluated\ to\ true$$

Of course, if there are no acceptable recipe, the task fails to execute. But if there is more than one acceptable recipe, the one to activate can for instance be the one that satisfies the largest number of preconditions. Other heuristics can be defined to achieve this selection, *e.g.* selecting the recipe that implies the smallest number of other agents, or exploiting cost functions to select the "most efficient" recipe. The definition of this selection process and associated heuristics or rules is an important entry point that allows the programmer to intervene in the overall robot control scheme.

# 5 Interactions between agents

Agents asynchronously interact through messages that specify constraints to enforce and messages that inform about the status of the constraint enforcement. We depict here the three mechanisms that define the agent interactions, to ensure that propagated constraints are properly enforced, to recover errors and to handle concurrency issues.

## 5.1 Constraint handling

An agent handles constraints according to the state machine shown figure 2. Constraint requests go through different states, each state corresponding to a specific message. When receiving a new constraint request, the agent computes a task tree, and then executes it. The request enters in the *running* state only when a recipe corresponding to the constraint handling is executed (and not one of its internal preconditions). This recipe can success, and so lead to *success* state, or fail, and so enter in the *temporary failure*. From this state, the agent can try to find a local solution to the failure; if it finds one, the request enters back in the *running* state, otherwise, it enters in the *failure* state – the recovery process is depicted in section 5.2. Messages can change the state of the constraint request handling: the abort message (from *abort*, or automatically sent at the end of the recipe) changes the state to *aborted*. Last, the request handling can enter in the *paused* state if the agent receives a pause message, and goes back to *running* state through the continue message.

The *paused* state is important to guarantee the consistency of the system. Consider a classic SPA loop composed of three concurrent constraint (*sensing_ctr*, *plan_ctr* and *exec_ctr*). If the satisfaction of the *plan_ctr* temporary fails, the agent needs to ensure that an old (maybe invalid) plan is not being executed: in reaction of a temporary failure of a constraint $C_i$, and agent pauses all the constraints

[1] Two tasks are incompatible if their post-conditions leads to inconsistency.

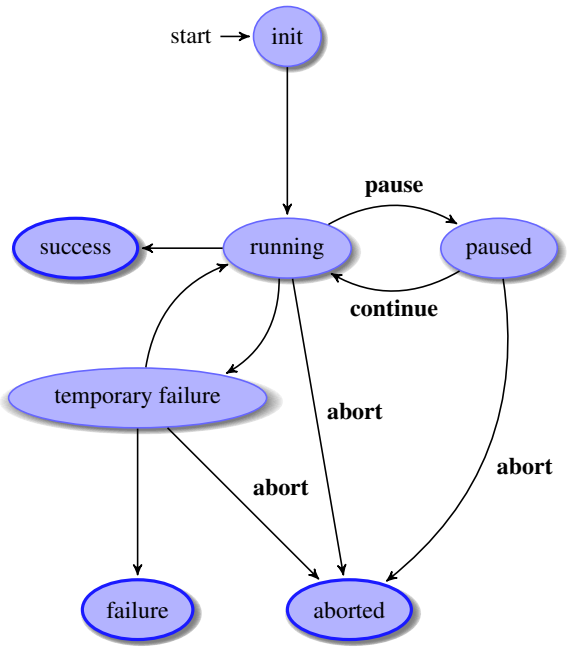


**Figure 2.** State machine to process the constraint requests

$C_j$ which depends on $C_i$ to ensure the safety of the robot. Figure 3 exhibits the message exchanged between agents in such a situation.

From an execution point of view, the paused state for a recipe is equivalent to the following steps:

1. interrupt the current synchronous primitive
2. forward a pause message to all current constraints
3. wait for a continue message
4. upon reception of a continue message, restart the interrupted synchronous primitive.

## 5.2 Error handling

Unexpected events (or errors) often occur during robotic missions, and it is essential to properly handle them in order to autonomously achieve the mission.

**Local errors.** Some errors can be predicted, when the developer knows the limitations of the functional layer or how to handle certain non-nominal situations. The developer can encode this knowledge using dedicated recipes whose preconditions check the known situations (this is nearly equivalent to exception handlers, as proposed by Simmons [18]). For this purpose, the primitive *last_error?* can be used.

But in the general case, the developer can hardly provide an explicit management for any possible error. In unpredicted failure cases, the agent tries to select another recipe which does not contain the faulty condition. Let the *constraint_domain* of a recipe the set of constraints it emits. The definition for a valid recipe at a time $t$ is then extended as :

$$domain(r) \subset available\_agents$$

$$constraint\_domain(r) \cap error\_context = \emptyset$$

$$all\ preconditions\ of\ r\ are\ evaluated\ to\ true$$

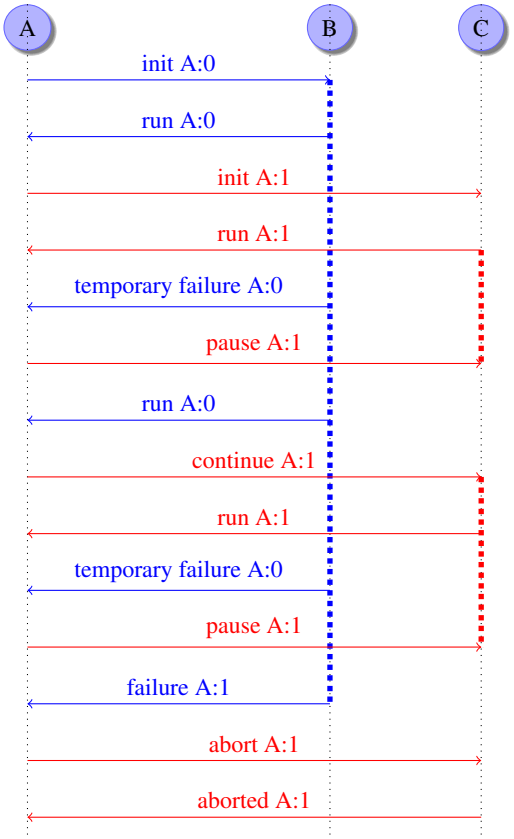


**Figure 3.** Exchanged messages between 3 agents in case of partial failure. Here the satisfaction of constraint *A:1* depends on the satisfaction of *A:0*: a temporary failure in the satisfaction of *A:0* pauses the satisfaction of *A:1*.

The agent selects one of this valid recipe according to the specified selection heuristic, and then executes it. In case a new failure occur, the *error_context* is filled accordingly. It is clear that every time an error occurs, the number of valid recipes decreases: this process eventually terminates, which means that the agent cannot handle the task at hand. The next solution for the agent is to find another set of tasks to handle the constraint. For this purpose, the system reuses the method described in 4.3, after having removed the faulty tasks from the set of available tasks.

**System level errors.** When an agent cannot locally recover a failure with the above mechanism, the failure goes up in the agent graph, with the full *error_context*. Each agent tries to fix the error locally, using the complete *error_context* to take its decision, *i.e.* the concatenation of its local *error_context* and the *error_context* of the agents in the failure chain. In this way, agents can efficiently decide an alternative solution: the history of the current constraint handling deters them to select paths leading to failures in the overall graph.

## 5.3 Managing concurrency

The *logic engine* of an agent rejects any new constraint $C_{new}$ conflicting with the constraints it is currently enforcing using algorithm described in 4.3. However, this algorithm can not differentiate whether $C_{new}$ is rejected because the agent has no mean to enforce it, or because one of the required task is temporally masked. The second case is a concurrency issue on a resource: an important feature of ROAR is the ability to autonomously handle such issues.

This is handled by searching a solution of the system described in 4.3 for each task, and not only for each available tasks. Here, one only check whether there is one task that can handle the constraint on not: if yes, there is a concurrency issue, otherwise it means that the agent is not able to handle $C_{new}$.

The *calling context* of a constraint $C$ is defined as the ordered stack of constraints whose enforcement generated a call to the constraint $C$ (it is equivalent to the notion of call stack in classic imperative programming).

When the agent detects a concurrency issue with the constraint $C_{new}$, the agent retrieves the calling context associated to the conflicting constraint $C_i$ returned by the algorithm[2] , and sends it back to the calling agent that emitted $C_{new}$. The calling agent then searches a solution, by asking successively to each agent that emitted a constraint of the calling context if it can find an alternate solution to solve the constraint $C_{i+1}$ without using the conflicting constraint $C_i$. For this purpose, each involved agent first evaluates if there are recipes associated to the current tasks for which $C_i$ does not appear in their *context_domain*. If not, it searches for an alternate executable tasks set that does not imply the enforcement of $C_i$. If it finds a solution to enforce $C_{i+1}$ without emitting $C_i$, it updates its internal configuration to activate the new recipes (or tasks), and sends OK to the calling agent, which the tries again to enforce $C_{new}$. Otherwise, it sends a not-OK, and the calling agent checks the agent associated to the next entry of the *calling context*, and so forth. If the *calling context* can not be cleared, the concurrency issue cannot be solved and is treated as a standard error, using the error recovery process depicted above.

# Part II
# A model for the ROAR architecture

## 6 Related work

Robot control architectures aim at providing solutions to interleave decisional, control and functional processes in order to en sure adaptiveness and robustness – and so does ROAR . Validating the architecture to guarantee the proper behaviour of the robot has seldom been considered: we review here the main existing contributions.

### 6.1 Robot control architecture validation

In [19], the author proposes to use a process algebra to specify and analysis its control language called *Robot Schemas*. The Orccad architecture [20] and the associated language MAESTRO [21] propose an encoding of the controller in the synchronous language ESTEREL, which allows to use Petri net tools to prove properties. In [22], the authors propose a translation from the robotic language TDL [18] to a specification language SMV, and then apply model checking techniques. But this translation is partial and so deduction on the system is limited. Moreover, the TDL is just a part of the NASA Remote Agent architecture [23], and so the model does not capture the whole architecture. In [24], the LAAS architecture is extended with the R2C layer: it contains a logic model representing wished constraints and allows to verify that commands sent from the control layer matches the model. While this approach ensures the satisfaction of constraints on the robot and prevents incorrect uses, it does not help the developer to write correctly the control layer, nor to fix it. More recently, [25] exploits the BIP formalism [26] to produce a formal model of the functional layer. Constraints can be added on the generated model, and then (partially) validated by formal tools like D-Finder [27]. At run-time, the BIP engine detects inconsistent transitions and rejects them. But this approach has the same drawbacks than R2C from the point of view of the control layer developer.

[2] In the general case, the described method returns a *list* of conflicting constraints $\{C_i\}$, and the process is actually applied to the associated *list* of calling contexts.

### 6.2 Logic execution models

One of the hard point about validating a control architecture is the interleaving of deliberation and (concurrent) execution, which usually rely on different models. Concurrent executions has been studied through different formal tools like Petri nets, or processes algebras (in particular $\pi$-calculus [28]) while deliberation is mostly a logic process. Girard introduced in [29] the linear logic, a new sub-structural logic which has had an important impact on the study of concurrent systems. [30] proposes to extend the Curry-Howard correspondence for concurrent systems, considering formulas as processes, and modelling dynamic behaviours by proof-search. Following this correspondence, [31] encodes the $\pi$-calculus and [32] proposes ACL, a concurrent linear logic language, which allows to represent messages and processes at the same (logic) level. More recently, [33] introduces CLF , a concurrent logic framework, using a different representation: concurrent computations are represented as monadic objects, separating in this way "classic computation" from concurrent ones. Last, Lollimon [34] extends the operational semantic of CLF to provide a new concurrent linear logic programming language. In particular, it interprets differently proof search inside and outside the monad constructor: outside, it uses a classic backward-chaining, with backtracking, while inside, it uses forward chaining and committed choice which naturally models concurrency.

## 7 Background

CLF is a logic framework, *i.e.* a formal meta-language specifically designed to represent and reason about programming languages, logics, or any formalisms that can be described as deductive systems. The paradigm of such a framework is to define the evaluation of a language as some signatures in the framework type theory, and then deduction on the language can be subsumed to a type inhibition problem in the framework type theory. The type theory of CLF is based on linear logic, a sub-structural logic in which the use of contraction and weakening rules are carefully controlled. In particular, linear logic introduces the linear implication $\multimap$ which consumes operands on its left to produce operands on its right, which allows to efficiently represent state evolutions. The exponential connective ! (often called "of course" operator) expresses that a resource can be used as much as desired, and so makes possible to encode the intuitionist implication $\rightarrow$. Andreoli [35] splits the different connectives of linear logic in two categories: asynchronous and synchronous (which respectively correspond to determinism and non-determinism in the proof search). This separation appears clearly in CLF with the monad which encapsulate synchronous primitives, allowing to preserve the semantic of LLF outside the monad. The type constructor theory of CLF can be described as :

$$
\begin{array}{lll}
A, B ::= \Pi x : A.B \mid A \multimap B \mid A \,\&\, B \mid \top \mid \{S\} \mid P & & \text{asynchronous} \\
P ::= a \mid P\ N & & \text{Atomic} \\
S ::= S_1 \otimes S_2 \mid 1 \mid \exists x : A.S \mid A & & \text{synchronous}
\end{array}
$$

Asynchronous types can be respectively defined by the dependant function type $\Pi x : A.B$, the linear function type $A \multimap B$, the additive product $A \,\&\, B$, the additive unit type $\top$, an atomic type $P$ or a synchronous type encapsulated in the monad constructor $\{S\}$. Atomic types are the constants, and the type level dependent application $P\ N$, where $N$ is an object. Last, the synchronous type includes the product type $S_1 \otimes S_2$, the unit multiplicative type 1, and the dependant pair type $\exists x : A.S$. Moreover, we can use syntactic sugar like $A \to B$ (the intuitionist function type) for $\Pi x : A.B$ if $B$ does not contain any free occurrence of $x$ and $!A$ for $\exists x : A.1$. A more complete description of the CLF framework is available in [36].

## 8 Formalization

Using CLF , we propose in this part an encoding of the ROAR operational semantic. First, we propose some generic computation patterns, like the notion of sequential execution, references. After that, we describe respectively how it is possible to encode the algorithm used to select tasks, then, how we can represent the selection and the execution of sequences.

### 8.1 Generic computation patterns

Following the proposal of [37], we choose a destination-passing style representation. We start with the types $exp\ T$ which represents an expression returning a $T$; $val\ T$ which represents a value of types $T$; and $dest\ T$, a destination of type $T$. The operational semantic is based on two family types, $eval\ (E\ D)$ which evaluates $E$ with destination $D$, and $return\ (V\ D)$ which returns the value $V$ to destination $D$. Informally, without special effects, if we have a linear assumption $eval\ (E\ D)$, then there is a computation $return\ (V\ D)$ if and only if the evaluation of $E$ yields $V$.

Let's first introduce some generic computation patterns. The first one is the sequential execution $llet\ E_1\ (\lambda x.E_2 x)$ which must first evaluate $E_1$ and then use its result to compute $E_2$. It can be encoded as follows:

$$
\begin{aligned}
&eval\_llet : eval\ (llet\ E_1\ (\lambda x.E_2 x)\ D)\\
&\quad \multimap \{\exists d_1 : dest\ T.eval\ (E_1\ d_1)\\
&\quad\quad \otimes (\Pi V_1 : val\ T.return\ (V_1\ d_1) \multimap \{eval\ ((E_2\ V_1)\ D)\})\}.
\end{aligned}
$$

This encoding presents an interesting behaviour. First, it creates a fresh destination $d_1$ and then create two processes using the $\otimes$ operator. The first one evaluate $E_1$ in $d_1$, and the second waits until there is a corresponding *return* assumption in the logic store. When $E_1$ is finally evaluated, the $return\ (V_1\ d_1)$ is consumed, and starts a new process which which evaluate $E_2$, considering the previously computed value $V_1$.

Each agent contains variables representing its state (internal or not): these variables are represented by references. Informally, $newref\ E_1$ creates a new reference with the result of the expression $E_1$, $assign\ (E_1\ E_2)$ modifies the reference represented by the computation of $E_1$ by the result of the computation of $E_2$, and $deref\ E_1$ returns the value stored in the reference represented by $E_1$. *cell* is used to generate a reference from a destination, and $contains\ (C\ V)$ means that the cell $C$ contains the value $V$. These three propositions are formalized as follows:

$$
\begin{aligned}
&eval\_newref : eval\ ((\text{newref}\ E_1)\ D)\\
&\quad \multimap \{\exists d_1 : dest\ T.eval\ (E_1\ d_1)\\
&\quad\quad \otimes (\Pi V_1 : val\ T.return\ (V_1\ d_1)\\
&\quad\quad\quad \multimap \{\exists c : dest\ T.contains\ (c\ V_1) \otimes return\ ((cell\ c)\ D)\})\}.\\
&eval\_assign : eval\ ((assign\ (E_1\ E_2))\ D)\\
&\quad \multimap \{\exists d_1 : dest(ref\ T).eval\ (E_1\ d_1)\\
&\quad\quad \otimes (\Pi C_1 : dest\ T.return\ ((cell\ C_1)\ d1)\\
&\quad\quad\quad \multimap \{\exists d_2 : dest\ T.eval\ (E_2\ d_2)\\
&\quad\quad\quad\quad \otimes (\Pi V_2 : val\ T.return\ (V_2\ d_2)\\
&\quad\quad\quad\quad\quad \multimap \Pi V_1 : val\ T.contains\ (C_1\ V_1)\\
&\quad\quad\quad\quad\quad\quad \multimap contains\ (C_1\ V_2) \otimes return\ ((void')\ D)\})\})\}.\\
&eval\_deref : eval\ ((deref\ E_1)\ D)\\
&\quad \multimap \{\exists d_1 : dest\ T.eval\ (E_1\ d_1)\\
&\quad\quad \otimes (\Pi C_1 : dest\ T.return\ ((cell\ C_1)\ d1)\\
&\quad\quad\quad \multimap \Pi V_1 : val\ T.contains\ (C_1\ V_1)\\
&\quad\quad\quad\quad \multimap contains\ (C_1\ V_1) \otimes return\ V_1\ D\})\}.
\end{aligned}
$$

It is interesting here to note the usage of a linear function, which allows to easily represent the imperative behaviour, by consuming (and possibly) recreating linear hypotheses.

Agents exchanges constraints through discrete messages. There are two kinds of messages: $msg\ T$, which represents a message encapsulating a value of type $T$, and channel $chan\ T$, which denotes a way to exchange $msg\ T$. $writeMsg\ (E_1\ E_2)$ takes an expression $E_1$ representing a channel, an expression $E_2$ representing a message, and adds the necessary predicate to the linear context. $readMsg\ E_1$ takes an expression $E_1$ representing a channel and returns when the agent can read a message on this channel. We now presents how these behaviours can be translated in CLF : here, the linear function ensures that messages are consumed when used, and so will not be treated twice.

$$
\begin{aligned}
&eval\_writeMsg' : eval\ ((writeMsg'\ (Ch\ M))\ D)\\
&\quad \multimap \{(msg\ Ch\ M) \otimes return\ ((void')\ D)\}\\
&eval\_writeMsg : eval\ ((writeMsg\ (E_1\ E_2))\ D)\\
&\quad \multimap \{eval\ (llet\ E_1\ (\lambda Ch\\
&\quad\quad\quad llet\ E_2\ (\lambda V.(writeMsg'\ (Ch\ V))))))\ D\}.\\
&eval\_readMsg' : eval\ ((readMsg'\ Ch)\ D)\\
&\quad \multimap \{\Pi V : val\ T.\ msg\ Ch\ V\\
&\quad\quad \multimap \{return\ (V\ D)\}\}.\\
&eval\_readMsg : eval\ ((readMsg\ E_1)\ D)\\
&\quad \multimap \{eval\ ((llet\ E_1\ (\lambda Ch.(readMsg'\ Ch))\ D)\}.
\end{aligned}
$$

Representing natural numbers is required for several algorithms. Here, we use a classical logic interpretation, starting by the type $z$, representing zero, and the type family $s\ nat$, which represent the successor of a natural. For example, 3 can be represented as $s\ (s\ (s\ z))$. Lists can be represented similarly, using a default type *nil*, and the constructor *cons*. The following shows how it is possible to evaluate the less or equal test on natural numbers:

$$
\begin{aligned}
&\mathit{eval\_less\_eq} : \mathit{eval}\ ((\mathit{less\_eq}\ M\ N)\ D)\\
&\quad \multimap \{\exists d_1 : \mathit{dest\ nat}.\mathit{eval}\ (M\ d_1)\\
&\quad\ \otimes (\mathit{return}\ (z\ d_1) \multimap \{\mathit{return}\ (\mathit{True}\ D)\}\\
&\quad\quad \& (\Pi V_1 : \mathit{val\ nat}.\ \mathit{return}\ ((s\ V_1)\ d_1)\\
&\quad\quad\ \multimap \{\exists d_2 : \mathit{dest\ nat}.\ \mathit{eval}\ (N\ d_2)\\
&\quad\quad\ \otimes (\mathit{return}\ (z\ d_2) \multimap \mathit{return}\ (\mathit{False}\ D)\\
&\quad\quad\quad \& (\Pi V_2 : \mathit{val\ nat}.\ \mathit{return}\ ((s\ V_2)\ d_2)\\
&\quad\quad\quad\ \multimap \{\mathit{eval}\ ((\mathit{less\_eq}\ V_1\ V_2)\ D)\})\})\})\}.
\end{aligned}
$$

## 8.2 Logic layer formalization

Now, we propose to encode the task selection process, described in section 4.3. To simulate the timing bound, we use a reference to a natural, and remove one at each step of the proof. If it reaches zero, the agent can not compute the proof in the limit, and returns $\mathit{False}$. The following fragment needs to be added to each rule, but for clarity and brevity, we only describe it once.

$$
\begin{aligned}
&\mathit{eval\_limit} : \mathit{eval}\ ((\mathit{limit}\ R)\ D)\\
&\quad \multimap \{\exists d_1 : \mathit{nat}.\ \mathit{eval}\ (R\ d_1)\\
&\quad\ \otimes (\mathit{return}\ (z\ d_1) \multimap \{\mathit{return}\ (\mathit{False}\ D)\}\\
&\quad\ \& (\Pi n\ :\ \mathit{nat}.\mathit{return}\ ((s\ n)\ d_1)\\
&\quad\quad \multimap \{\exists d_2 : \mathit{dest\ void}.\mathit{eval}\ ((\mathit{assign}\ (R\ n))\ d_2)\\
&\quad\quad\ \otimes \mathit{return}\ (\mathit{True}\ D)\})\})\}.\}.
\end{aligned}
$$

The only notable point is this encoding is the use of the additive product & to represent the branch notion in the algorithm.

To encode the proof search, we introduce three new family types: *thm* which represent a theorem, *hyp* that represents the forward chaining part, and *prove* which represents the backward chaining process. In the following encoding, we use the intuitionist function operator ← instead of the linear function operator (and the 'of course' operator ! inside the monad) , as a proof does not consume any resource, but only use facts. ← or → are used to respectively emphasize the backward chaining and forward chaining semantic. Beyond that, the encoding is quite simple, just transcribing classic deduction rules into the proper CLF connectives.

$$
\begin{aligned}
\mathit{hfalse} &: \mathit{hyp}\ \mathit{false} \to \{\Pi C : \mathit{thm} :!C\}\\
\mathit{hand} &: \mathit{hyp}(\mathit{and}\ A\ B) \to \{!A \otimes !B\}\\
\mathit{hymp} &: \mathit{hyp}\ A \to \mathit{hyp}\ (\mathit{imp}\ A\ B) \to \{!B\}\\
\mathit{hall} &: \mathit{hyp}(\mathit{all}\ A) \to \{\Pi x : \mathit{thm}.!(\mathit{hyp}\ A\ !x)\}\\
\mathit{hsome} &: \mathit{hyp}(\mathit{some}\ A) \to \{\exists x : \mathit{thm}.!(\mathit{hyp}\ A\ !x)\}\\
\mathit{proveand} &: \mathit{prove}\ (\mathit{and}\ A\ B) \leftarrow \mathit{prove}\ A\ \&\ \mathit{prove}\ B\\
\mathit{proveimp} &: \mathit{prove}\ (\mathit{imp}\ A\ B) \leftarrow (\mathit{hyp}\ A \leftarrow \mathit{prove}\ B)\\
\mathit{proveall} &: \mathit{prove}\ (\mathit{all}\ A) \leftarrow (\Pi x : \mathit{thm}.\mathit{prove}\ (A\ !x))\\
\mathit{provesome} &: \mathit{prove}\ (\mathit{some}\ A) \leftarrow \{\exists x : \mathit{thm}.\mathit{prove}\ (A\ !x)\}\\
\mathit{provetrue} &: \mathit{prove}\ \mathit{True}.\\
\mathit{proh} &: \mathit{proveA} \leftarrow \{!(\mathit{hypA})\}.
\end{aligned}
$$

## 8.3 Executive layer formalization

The section 4.4 describes two important mechanisms in the execution process: the selection of a recipe, and its execution.

This selection process is encoded as follows in CLF . First, *pre* is a recursive function which evaluates if all pre-conditions are satisfied and the number of pre-conditions (it returns a pair composed of a Boolean and a natural number *N*). Then, *select* is a recursive function, which iterates over the list of recipes *L*, and checks if the current recipe is better than the currently selected recipe *R*. The recursion is started with the full list of recipes, a nil recipe, and *N* equal to 0.

$$
\begin{aligned}
&\mathit{eval\_pre} : \mathit{eval}\ ((\mathit{pre}\ R\ N)\ D)\\
&\quad \multimap \{\exists d_1 : \mathit{dest}(\mathit{list}\ T).\mathit{eval}\ (R\ d_1)\\
&\quad\ \otimes (\mathit{return}\ (\mathit{nil}\ d_1) \multimap \mathit{return}\ ((\mathit{false}, N)\ D)\\
&\quad\quad \& (\mathit{return}\ ((\mathit{cons}\ E_1\ E_2)\ d_1)\\
&\quad\quad\ \multimap \{\exists d_2 : \mathit{dest}(\mathit{bool}).\mathit{eval}\ (E_1\ d_2)\\
&\quad\quad\ \otimes ((\mathit{return}\ (\mathit{False}\ d_2)\\
&\quad\quad\quad \multimap \{\mathit{eval}\ ((\mathit{pre}\ E\_2\ s(N))\ D)\}\\
&\quad\quad\ ) \& (\mathit{return}\ (\mathit{True}\ d_2)\\
&\quad\quad\quad \multimap \mathit{return}\ ((\mathit{False}, N)\ D))\})\})\}.
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval\_select} : \mathit{eval}\ ((\mathit{select}\ L\ R\ N)\ D)\\
&\quad \multimap \{\exists d_1 : \mathit{dest}(\mathit{list}\ T).\mathit{eval}\ (L\ d_1)\\
&\quad\ \otimes (\mathit{return}\ (\mathit{nil}\ d_1) \multimap \mathit{return}\ (R\ D)\\
&\quad\quad \& (\mathit{return}\ ((\mathit{cons}\ E_1\ E_2)\ d_1)\\
&\quad\quad\ \multimap \{\exists d_2 : \mathit{dest}(U).\mathit{eval}\ (E_1\ d_2)\\
&\quad\quad\ \otimes ((\mathit{return}\ ((\mathit{False}, \_)\ d_2)\\
&\quad\quad\quad \multimap \{\mathit{eval}\ ((\mathit{select}\ E_2\ R\ N))\ D)\}\\
&\quad\quad\ ) \& (\mathit{return}\ ((\mathit{True}, K)\ d_2)\\
&\quad\quad\quad \multimap \{\exists d_3 : \mathit{dest}(\mathit{bool}).\\
&\quad\quad\quad\quad \mathit{eval}\ ((\mathit{less\_eq}\ K\ N)\ d_3)\\
&\quad\quad\quad \otimes ((\mathit{return}\ (\mathit{True}\ d_3)\\
&\quad\quad\quad\quad \multimap \{\mathit{eval}\ ((\mathit{select}\ E_2\ R\ N)\ D)\})\\
&\quad\quad\quad\quad \& (\mathit{return}\ (\mathit{False}\ d_3)\\
&\quad\quad\quad\quad \multimap \{\mathit{eval}\ ((\mathit{select}\ E_2\ E_1\ K)\ D)\})\})\})\})\}.
\end{aligned}
$$

As described previously, a recipe is a sequence based on a few different primitives. We now formalize each of these primitives. Then, the whole recipe is only the chaining of these different primitives, using the *llet* operator. **let** and **set** are quite easy to encode with the use of the reference notion previously introduced. We associate to each recipe a special channel, which indicates to each operation if it must continue (True) or stop (False). The encoding of **wait** shows its usage, and some others interesting features: first, it is recursive, and second, it uses the & CLF operator to express that the framework must execute only one branch. It is also important to note that we add again the recipe status after its use, so others operations can see it in the linear context. The encoding of **make** is relatively easy too: it sends the message composed of the constraint and the private channel to answer, and then creates two processes: the first one waits for an answer on this private channel, while the other waits for a interruption message from the recipe(*has_abort*): in this case, it just writes "Abort" on the private channel, which leads to the termination of the **make** primitive. The encoding of **ensure** is quite similar, but it returns directly after sending the message, and creates two others processes: one which checks for abort message as for *make*, and one which checks messages from remote agents (*answer*). The constraint identifier is represented as a private channel, and so **abort** is trivially encodable: it just writes "Abort" on this channel.

$$
\begin{aligned}
&eval\_let : eval\ ((let\ E_1\ E_2)\ D)\\
&\quad \multimap \{eval\ (llet\ (newref\ E_1)\ (\lambda x.\ (assign\ (x\ E_2)\ D)\}.\\
&eval\_set : eval\ ((set\ E_1\ E_2)\ D)\\
&\quad \multimap \{eval\ (assign\ (E_1\ E_2))\ D)\}.\\
&eval\_has\_abort : eval\ ((has\_abort\ Ch_1\ Ch)\ D)\\
&\quad \multimap \{(msg\ Ch\ False)\\
&\quad\quad \multimap \{\exists d_1 : dest(void).\\
&\quad\quad\quad eval\ ((writeMsg\ (Abort\ Ch_1))\ d_1)\\
&\quad\quad\quad \otimes (msg\ Ch\ False)\})\}.\\
&eval\_make : eval\ ((make\ E_1\ C\ Ch)\ D)\\
&\quad \multimap \{\exists Ch_1 : Chan\ S.\ \exists d_1 :\ dest(void).\\
&\quad\quad eval\ (writeMsg\ (E1\ (C, Ch_1))\ d_1)\\
&\quad\quad\quad \otimes (\Pi V1 : void.return\ (V_1\ d_1)\\
&\quad\quad\quad \multimap \{eval\ ((readMsg\ Ch_1)\ D)\\
&\quad\quad\quad\quad \otimes (\exists d_2 : dest(void).\\
&\quad\quad\quad\quad\quad eval\ ((has\_abort\ Ch_1\ Ch)\ d_2)\})\}\\
&eval\_answer : eval\ ((answer\ Ch_1\ Ch)\ D)\\
&\quad \multimap \{\exists d_1 : destT.eval\ (readMsg\ Ch_1\ d_1)\\
&\quad\quad \otimes (return\ (True\ d_1)\\
&\quad\quad\quad \multimap \{eval\ ((answer\ Ch_1\ Ch)\ D)\})\\
&\quad\quad \&\ return\ (False\ d_1)\\
&\quad\quad\quad \multimap \{eval\ ((writeMsg\ (Abort\ Ch))\ D)\}\\
&\quad\quad \&\ return\ (Abort\ d_1)\\
&\quad\quad\quad \multimap \{eval\ (writeMsg\ (Abort\ Ch)\ D)\}\}.\\
&eval\_ensure : eval\ ((ensure\ E_1\ C\ Ch)\ D)\\
&\quad \multimap \{\exists Ch_1 : Chan\ S.\ \exists d_1 :\ dest(void).\\
&\quad\quad eval\ (writeMsg\ (E1\ (C, Ch_1))\ d_1)\\
&\quad\quad \otimes (\Pi V1 : void.return\ (V_1\ d_1)\\
&\quad\quad \multimap \{(\exists d_2 : dest(void).\\
&\quad\quad\quad\quad eval\ ((ensureAnswer\ Ch_1\ Ch)\ d_2))\\
&\quad\quad\quad \otimes (\exists d_3 : dest(void).\\
&\quad\quad\quad\quad eval\ ((has\_abort\ Ch_1\ Ch)\ d_3))\\
&\quad\quad\quad \otimes return\ (Ch_1\ D)\})\}.\\
&eval\_wait : eval\ ((wait\ E_1\ Ch)\ D)\\
&\quad \multimap \{\exists d_1 : dest(Bool).\ eval\ (E_1\ d_1)\\
&\quad\quad \otimes ((return\ (True\ d_1) \multimap \{return\ (True\ D)\})\\
&\quad\quad\quad \&\ (return\ (False\ d_1) \multimap\\
&\quad\quad\quad\quad ((msg\ Ch\ True) \multimap\\
&\quad\quad\quad\quad\quad \{(msg\ Ch\ True)\\
&\quad\quad\quad\quad\quad\quad \otimes eval\ ((wait\ E_1\ Ch)\ D)\}\\
&\quad\quad\quad\quad \&\ (msg\ Ch\ False) \multimap\\
&\quad\quad\quad\quad\quad \{(msg\ Ch\ False)\\
&\quad\quad\quad\quad\quad\quad \otimes return\ (False\ D)\}))\}.\\
&eval\_abort : eval\ ((abort\ E_1)\ D)\\
&\quad \multimap \{eval\ (writeMsg\ (E_1\ Abort))\ D)\}.
\end{aligned}
$$

## 8.4 Summary

This section reviewed the different constructions of the ROAR , and their associated semantic. Then the behaviour of each construction is encoded using CLF . By using a destination-passing style representation, it is easy to compose the behaviours, and so to provide a full model of the control architecture.

# 9 Conclusion

**Summary.** We have presented the design of a framework to control the run-time configuration of complex robotic systems: it defines a control architecture as a network of agents managing individual resources. This decomposition in resource allows better granularity and composability than task-based decomposition, it makes the system more dependable (no single point of failure), and scales and reacts well, as computations are handled by agents which reason locally. The error representation and the internal agent model allow the system to react properly to unexpected events, and by reasoning on resources, the system can detect concurrent access situations and automatically propose reconfiguration to handle them. This work has been inspired by the architecture T-REX, but also by different programming paradigm, such as concurrent languages (like Erlang [41], and its hierarchy of supervision processes), contract programming languages [17] like Eiffel and its concurrent version Scoop [42]).

ROAR has been implemented and is exploited on an everyday basis to control the navigation processes of our research robot Mana. A very valuable feature from which we have already benefited is that the integration of new functional modules does not call for any thorough review of the overall architecture: new agents are simply defined and seamlessly added to the network of existing agents.

We also demonstrated that it is possible to encode ROAR into a rich enough logic framework that represents both the deliberative and executive parts of the architecture. Such a formalisation is useful for several reasons. First, it allows a better comprehension of the framework by using a clear and formal semantic. Second, we think that other control architecture frameworks can be formalized using the same techniques: this logic representation could then be used as a common language to compare architecture. Last, and probably most importantly, this logic representation makes it possible to formally reason about the control framework, and so to automatically (or semi-automatically) prove some properties.

**Future work.** Up to now, the agent specification is manually encoded into CLF , using the CELF tool [46] to verify that the encoding is correct. Ongoing work includes the automatic generation of the CLF specification from the language description of ROAR . Then, one can work on the establishment of proofs of some properties, such as the existence of non achievable states or the detection of livelocks.

Another perspective is to extend ROAR to the control of multi-robot systems. A readily extension is to endow the robots with the knowledge of the resources of the others, thus yielding the establishment of cooperating schemes in a rather transparent way. A second possible extension is to integrate a resource allocation algorithm (*e.g.* within a market based approach) to ROAR .

# References


[1] S. Fleury, M. Herrb, and R. Chatila, "GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture," in *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, pp. 842 –849 vol.2, Sept. 1997.

[2] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation, Seoul (Korea)*, pp. 2523–2528, 2001.

[3] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[4] A. Shakhimardanov, J. Paulus, N. Hochgeschwender, M. Reckhaus, and G. K. Kraetzschma, "Best Practice in Robotics: Best Practice Assessment of Software Technologies for Robotics," tech. rep., Bonn-Rhein-Sieg University, 2010.

[5] D. Brugali and A. Shakhimardanov, "Component-Based Robotic Engineering (Part II)," *IEEE Robotics and Automation Magazine*, vol. 17, pp. 100 –112, march 2010.

[6] E. Coste-Manière and R. Simmons, "Architecture, the backbone of robotic systems," in *IEEE International Conference on Robotics and Automation, San Francisco, CA (USA)*, 2000.

[7] E. Coste-Manière and B. Espiau, eds., *Special Issue on Integrated Architectures for Robot Control and Programming*, vol. 17(4) of *International Journal of Robotics Research*. Sage, April 1998.

[8] N. J. Nilsson, *Principles of artificial intelligence*. Morgan Kaufmann, San Francisco, CA, USA, 1980.

[9] E. Gat, "Integrating reaction and planning in a heterogeneous asynchronous architecture for mobile robot navigation," *SIGART Bull.*, vol. 2, pp. 70–74, July 1991.

[10] R. A. Brooks, "Elephants don't play chess," *Robotics and Autonomous Systems*, vol. 6, pp. 3–15, 1990.

[11] P. Giorgini, M. Kolp, and J. Mylopoulos, "Multi-agent architectures as organizational structures," *Autonomous Agents and Multi-Agent Systems*, vol. 13, p. 2006, 2001.

[12] B. Innocenti, B. López, and J. Salvi, "A multi-agent architecture with cooperative fuzzy control for a mobile robot," *Robotics and Autonomous Systems*, vol. 55, no. 12, pp. 881 – 891, 2007.

[13] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, pp. 315–337, April 1998.

[14] D. Bernard, G. Dorais, C. Fry, E. G. Jr., B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. P. Nayak, B. Pell, K. Rajan, and N. Rouquette, "Design of the Remote Agent experiment for spacecraft autonomy," in *IEEE Aerospace Conference*, 1998.

[15] N. Muscettola, G. Dorais, C. Levinson, and C. Plaunt, "IDEA: Planning at the Core of Autonomous Reactive Agents," in *International NASA Workshop on Planning and Scheduling for Space*, 2002.

[16] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. Mcewen, "A Deliberative Architecture for AUV Control," in *IEEE International Conference on Robotics and Automation, Pasadena (USA)*, 2008.

[17] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, pp. 40–51, October 1992.

[18] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *IEEE/RSJ Conference on Intelligent Robots and Systems, Victoria, B.C (Canada)*, 1998.

[19] D. Lyons, "Representing and analyzing action plans as networks of concurrent processes," *Robotics and Automation, IEEE Transactions on*, vol. 9, pp. 241–256, Jun 1993.

[20] J. Borrelly, É. Coste-Maniere, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro, "The orccad architecture," *The International Journal of Robotics Research*, vol. 17, no. 4, p. 338, 1998.

[21] E. Coste-Maniere and N. Turro, "The maestro language and its environment: Specification, validation and control of robotic missions," in *Intelligent Robots and Systems, 1997. IROS'97., Proceedings of the 1997 IEEE/RSJ International Conference on*, vol. 2, pp. 836–841, IEEE, 1997.

[22] R. Simmons, C. Pecheur, and G. Srinivasan, "Towards automatic verification of autonomous systems," in *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, vol. 2, pp. 1410–1415, IEEE, 2000.

[23] N. Muscettola, P. Nayak, B. Pell, and B. Williams, "Remote agent: To boldly go where no ai system has gone before," *Artificial Intelligence*, vol. 103, no. 1-2, pp. 5–47, 1998.

[24] F. Ingrand and F. Py, "An execution control system for autonomous robots," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 2, pp. 1333 –1338, 2002.

[25] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A verifiable and correct-by-construction controller for robot functional levels," *Journal of Software Engineering for Robotics*, vol. 1, no. 2, pp. 1–19, 2011.

[26] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-Time Components in BIP," in *Proceedings Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pp. 3–12, IEEE Computer Society Press, 2006.

[27] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis, "D-finder: A tool for compositional deadlock detection and verification," in *Computer Aided Verification*, pp. 614–619, Springer, 2009.

[28] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, Parts I and II," *Journal of Information and Computation*, vol. 100, pp. 1–40 and 41–77, 1992.

[29] J. Girard, "Linear logic," *Theoretical computer science*, vol. 50, no. 1, pp. 1–101, 1987.

[30] N. Martí-Oliet and J. Meseguer, "From petri nets to linear logic," in *Category Theory and Computer Science*, (London, UK), pp. 313–340, Springer-Verlag, 1989.

[31] G. Bellin and P. Scott, "On the $\pi$-calculus and linear logic," *Theoretical Computer Science*, vol. 135, no. 1, pp. 11 – 65, 1994.

[32] N. Kobayashi and A. Yonezawa, "Acl - a concurrent linear logic programming paradigm," in *Proceedings of the 1993 International Logic Programming Symposium*, pp. 279–294, 1993.

[33] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker, "A concurrent logical framework: The propositional fragment," *Types for Proofs and Programs*, pp. 355–377, 2004.

[34] P. López, F. Pfenning, J. Polakow, and K. Watkins, "Monadic concurrent linear logic programming," in *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '05, (New York, NY, USA), pp. 35–46, ACM, 2005.

[35] J. Andreoli, "Logic programming with focusing proofs in linear logic," *Journal of Logic and Computation*, vol. 2, no. 3, pp. 297–347, 1992.

[36] I. Cervesato, K. Watkins, F. Pfenning, and D. Walker, "A concurrent logical framework i: Judgments and properties," tech. rep., Carnegie Mellon University, 2003.

[37] I. Cervesato, F. Pfenning, D. Walker, and K. Watkins, "A concurrent logical framework ii: Examples and applications," tech. rep., Carnegie Mellon University, 2003.

[38] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A high level supervision and control language for autonomous mobile robots," in *In IEEE International Conference on Robotics and Automation, Mineapolis*, 1996.

[39] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with haskell," *Practical Aspects of Declarative Languages*, pp. 91–105, 1998.

[40] S. Joyeux, R. Alami, S. Lacroix, and R. Philippsen, "A plan manager for multi-robot systems," *The International Journal of Robotics Research*, vol. 28, no. 2, pp. 220–240, 2009.

[41] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, *Concurrent Programming in ERLANG - Second Edition*. Prentice Hall, 1996.

[42] B. Morandi, S. S. Bauer, and B. Meyer, "Scoop - a contract-based concurrent object-oriented programming model," in *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008* (P. Müller, ed.), vol. 6029 of *Lecture Notes in Computer Science*, pp. 41–90, Springer, 2008.

[43] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part i," *I and II. Information and Computation*, vol. 100, 1989.

[44] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, pp. 1–102, 1987.

[45] R. Firby, *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, Department of Computer Science, 1989.

[46] A. Schack-Nielsen and C. Schürmann, "Celf–a logical framework for deductive and concurrent systems (system description)," *Automated Reasoning*, pp. 320–326, 2008.