An Execution Control System for Autonomous Robots

Félix Ingrand, Frederic Py
LAAS/CNRS,
7 Avenue du Colonel Roche, F-31077 Toulouse Cedex 04, France
{*felix,fpy*}*@laas.fr*

Abstract— **This paper presents some recent developments of the LAAS architecture for autonomous mobile robots. In particular, we specify the role of the Execution Control level of this architecture. This level has a fault protection role with respect to the commands issued by the decisional level, which are transmitted to the real system (through the functional level). We introduce a new approach and a new tool inspired from the model checking domain. We present a new language to specify the model of acceptable and required states of the system (valid contexts for requests to functional module and resources usage). This language is compiled in an OBDD (Ordered Binary Decision Diagram) like structure which is then used online to check the specified constraints in real-time. Such model checking approach could be extended to check off line more complex temporal properties of the system.**

I. Introduction

There is an increasing need for advanced autonomy in complex embedded real-time systems such as robots, satellites, or UAVs. The growing complexity of the decision capabilities of these systems raises a major problem: how to prove that the system is not going to engage in dangerous states? How to guarantee that the robot will not grab a sample with its arm, while moving (which could supposedly break the arm)? How to make sure that RCS jets on a satellite are not fired when the camera lens protection is off? etc. A partial response to this problem is to use a planner which only synthesizes valid and safe plans. Yet, high level planners do not (cannot) have a complete model representing the full extend of their actions. Some of these actions are refined by the supervisor/executive, therefore the particular sequence of commands sent to the physical system is not completely planned.

A solution to guarantee the safety properties is to integrate a system that formally controls the validity of the commands sent to the physical system and prevents it from entering in an inconsistent state. This controller must check the system consistency online during system execution without affecting the system basic functionalities, such as reaction time.

The LAAS[1] architecture, presented in section II, foresaw such a mechanism in its Execution Control Level, but for various reasons, the approach and tools proposed to fill this functionality were not used. Section III presents the Execution Control Level roles and requirements, with a state of the art and related works. Section IV gives an informal description of the proposed approach, the tool and the language we use for the Execution Control Level. In section V, we present some experimental results of the execution control system. We then conclude the paper and consider future works and research directions.

II. The LAAS Architecture

The LAAS architecture [1] was originally designed for autonomous mobile robots. This architecture remains fairly general and is supported by a consistently integrated set of tools and methodology, in order to properly design, easily integrate, test and validate a complex autonomous system.

As shown on figure 1, it has three hierarchical levels, with different temporal constraints and manipulating different data representations. From the top to the bottom, the levels are:

- *A decision level:* This higher level includes the deliberative capabilities such as, but not limited to: producing task plans, recognizing situations, faults detections, etc. It embeds at least a supervisor/executive [2], which is connected to the underlying level, to which it sends requests that will ultimately initiate actions and start treatments. It is responsible for supervising plans or procedures execution while being at the same time reactive to events from the underlying level and commands from the operator. Then according to particular applications it may integrate other more complex deliberation capabilities, which are called by the supervisor/executive when necessary. The temporal properties of the supervisor are such that one guarantees the reaction time of the supervisor (i.e. the time elapsed before it sees an

This paper has been published in the proceedings of the IEEE International Conference on Robotics and Automation, Washington D.C., May 11 - 15, 2002

List of authors in alphabetical order.

[1]LAAS Architecture for Autonomous System.

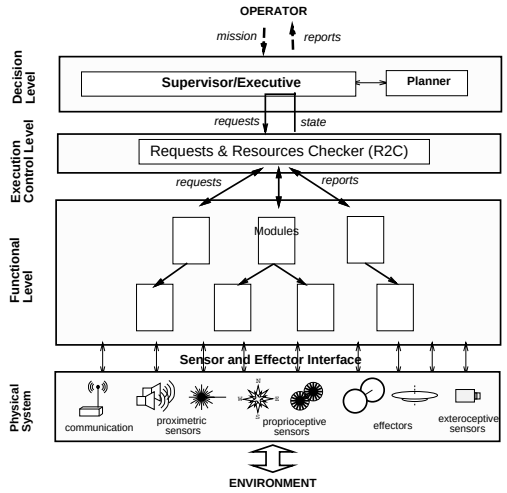

Fig. 1. The LAAS Architecture.

event), but not much can be said for other decisional components.

• *An execution control level:* Just below the decisional level, the Requests and Resources Checker (R2C) checks the requests sent from above to the functional level, as well as the resources usage. It is synchronous with the underlying functional modules, in the sense that it sees all the requests sent to them, and all the reports coming back from them. It acts as a filter which allows or disallows requests to pass, according to the current state of the system (which is built online from the past requests and past replies) and according to a formal model of allowed and forbidden states of the functional system. The temporal requirements of this level are hard real-time. This is the level on which this paper focuses.

• *A functional level:* It includes all the basic built-in robot action and perception capabilities. These processing functions and control loops (image processing, motion control, ...) are encapsulated into controllable communicating modules [3]. Each module provides a number of services and treatments available through requests sent to it. Upon completion or abnormal termination, reports (with status) are sent back to the requester. Note that modules are fully controlled from the decisional level through the R2C. Modules also maintain so called "posters"; data produced by the modules, such as the current position and speed (from the locomotion module) or current trajectory (from the motion planning module) which can be seen by other modules and the levels above. The temporal requirements of the modules depend on the type of treatments they perform. Modules running servo loop (which have to be ran at precise rate and interval without any lag) will have a higher temporal requirement than a motion planner, or a localization algorithm.

This architecture naturally relies on several representations, programming paradigms and processing approaches meeting the precise requirements specified for each level. We developed proper tools to meet these specifications and to implement each level of the architecture: IxTeT a temporal planner, Propice a procedural system for tasks refinement and supervision/executive, and GenoM for the specification and integration of modules at that level. These various tools share the same namespace (i.e. the name of the modules, requests, arguments and posters).

This paper focuses on the Execution Control Level. Until recently, this level was implemented using the KHEOPS system, but for various reasons (language, complexity, etc) we moved to a newer approach/tool: the Requests and Resources Checker (R2C) and the ExoGen tool used to implement it.

III. Execution Control Level

A. Role and Requirements

The main role of the Execution Control Level and its main component the R2C is a fault protection role. Faults are inevitable, even more with complex decisional system partially based on informal methods and tools. Yet to be able to use such advanced decisional tools, one needs to design systems which in the worse cases prevent the system from engaging into disastrous situations. Thus the execution control level has a "simple", yet critical role in the architecture:

• As the interface between the decisional and the functional level, it ensures that all the requests passed to the functional level remain consistent with respect to a model of desirable or undesirable states of the system, i.e. interactions between the functional modules. For example, it is the R2C role to make sure that a request to move the robot is not issued while a picture is being taken.
• It manages the resources of the system and guarantees that any requests leading to an overconsumption or inconsistent use of resources is properly handled.
• It acts synchronously with the functional level to ensure a consistent view of the state of functional modules.
• It acts in guaranteed real-time. No request to the functional level should be delayed more than one R2C cycle before being processed.

This critical role requires the use of formal tools to validate it. Moreover for this tool to be used by the engineers developing complex autonomous systems, one needs to provide a user-friendly specification language.

B. State of the Art in Execution Control

Many of the concerns raised in the previous section are not new, and some robotics architectures address them in one way or another.

Indeed, some of the requirements presented above were clearly fulfilled by a previous version of the LAAS execution control layer based on KHEOPS [4]. KHEOPS is a tool for checking a set of propositional rules in real-time. A KHEOPS program is thus a set of production rules ($condition(s) \rightarrow action(s)$), from which a decision tree is built. The main advantage of such representation is the guaranty of a maximum evaluation time (corresponding to the decision DAG depth). However, the KHEOPS language is not adapted to resources checking and appears to be quite cumbersome to use.

Another interesting approach to prove various formal properties of robotics system is the ORCCAD system[5]. This development environment, based on the ESTEREL [6] language provides some extensions to specify robots "tasks" and "procedures". However, this approach does not address architecture with advanced decisional level such as planners.

In [7], the author presents another work related to synchronous language which has some similarities with the work presented here. The objective is also to develop an execution control system with formal checking tools and a user-friendly language. This system represents requests at some abstraction level (no direct representation of arguments nor returned values). This development environment gives the possibility to validate the resulting automata via model-checking techniques (with SIGALI, a SIGNAL extension).

In [8], the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description (preconditions, postconditions and deadlines of tasks). It can then deduce the corresponding timed automaton to control on-line the system, with respect to these constraints. This automaton can be formally validated with model checking techniques.

In [9] the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language) to SMV a symbolic model checker language. Compared to our approach, this system seems to be more designed for the high level specification of the decisional level, while our approach focuses on the online checking of the outcomes of the decisional level.

IV. R2C and the ExoGen Tool

In this section we give a description of the R2C, the main component of the LAAS Execution Control Level. The internal model of the R2C is built using the ExoGen tool which largely uses the GenoM semi formal descriptions of the underlying functional modules [3] and its namespace.

A. Overview

The R2C (see figure 2) is designed to support a safe execution of the system. It contains a database representing the current state of the functional level (i.e. running instances of requests, resources levels, and history of requests) and – according to these information and the model checker – calculates appropriate actions to keep the system safe.

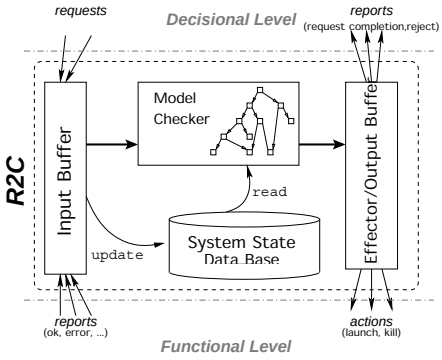

Fig. 2. R2C general view.

The possible R2C actions are: to launch a request; to kill an existing request; to reject a request (and report it) and to report a request completion.

B. A real example

In this section we illustrate ExoGen usage and the model we implemented on our XR4000 Nomadics. This example is mostly related to the poster localization function used on this robot[2]. The figure 3 represents dependencies between modules described below:

- `xr4000:` This module groups all the basic functionalities of the xr4000 nomadics. We can find here requests for locomotion, US and IR control.
- `LRF:` This module controls the LRF and calculates segments for localization and obstacles detection.
- `platine:` It controls the pan end tilt cameras positions.
- `camera:` Manages cameras and images acquisition.
- `nd`,`band:` Two modules to control the navigation and obstacle avoidance.
- `segloc:` Localizes the robot according to laser segments and odometric robot position.
- `locpost:` This module controls the visual localization with posters.

[2]Our XR4000 has various means to localize such as, LRF segments matching in a previously acquired map, or localization using posters on the wall, etc.

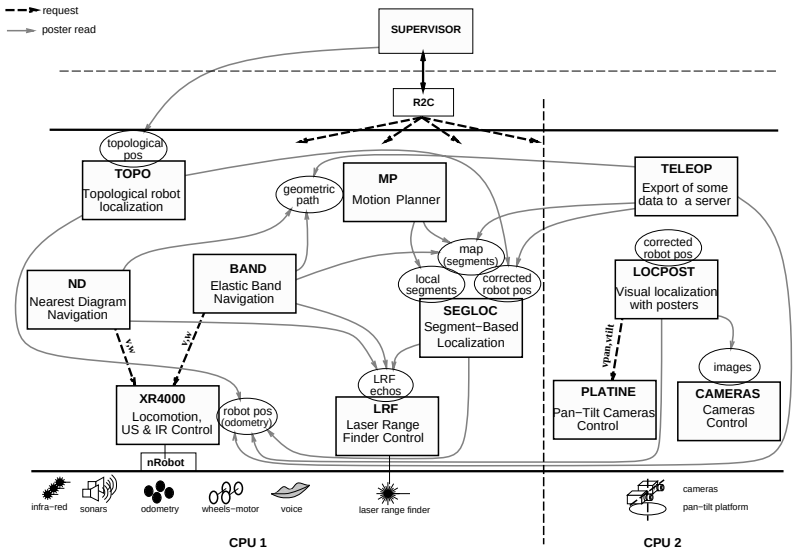

Fig. 3. Diligent Architecture

• `teleop:` Grab some robot information and send it to a server periodically. It is used to control our robots via a web interface.
• `topo:` Calculates the topological position of the robot used by the supervisor(Example: the robot has to move more slowly in a room than in a corridor).
• `MP:` The Motion Planning module computes the best trajectory for the robot to reach a particular position (taking into account known obstables).

To fully present this example we need to describe some requests of the involved modules:

• `xr4000_GotoRelative(?relative_pos)` This request of the `xr4000` module is used to make basic moves. `?relative_pos` is a two dimensional position relative to the current robot position.
• `band_Move(?pos), nd_Goto(?pos), nd_ExecTraj(?pos)` Are three requests to move the robot using various avoidance mechanism (Elastic Band in the `band` module, and Near Diagram approach for `nd`) to an absolute position `?pos`. They are all incompatible with one another and use `xr4000_GotoRelative` at the lowest level.
• `locPost_ActivePosterSearch(?poster)` is a request of the `locPost` module, called to enable robot localization using posters on the wall. When it runs, the robot tries to find the poster `?poster` and correct its position according to the localization.
• `platine_CmdPosTilt(?pos), platine_CmdPosPan(?pos)` Are requests from the `platine` modules to set the pan/tilt camera platine position to `?pos`.

Note that these seven requests from five different modules are withdrawn from a much larger set of modules and requests just to give example of the R2C usage.

Here are some constraints examples for these requests execution:
• one cannot call `xr4000_GotoRelative` while a moving request execute (i.e. if `band_Move`, `nd_Goto` or `nd_ExecTraj` is active),
• only one moving request can be executed at the same time,
• the platine cannot be moved by an external request during a poster search,
• the robot cannot move during a poster search,

All these constraints can be expressed with the ExoGen tool described below.

C. Presentation of ExoGen

This section presents the ExoGen tools and its language used to build the main components of the R2C.

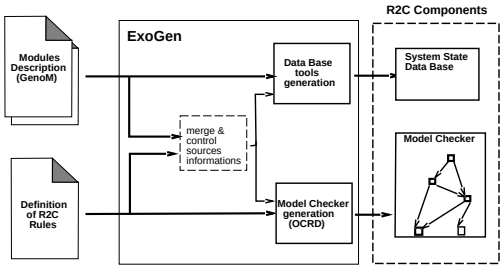

Fig. 4. ExoGen development cycle.

An ExoGen program consists of a set of requests and resources usage descriptions. For each request of the functional level – as defined by GenoM – one may define the preconditions it has to verify to be allowed

for execution. Those preconditions can be defined on the arguments values of the requests themselves, past requests (i.e. running requests) and states of the current system (which results from previously completed requests). Moreover when applicable, one has to specify the resources used by a particular request call.

The ExoGen language has been specifically designed to easily represent those descriptions. For a particular application, the ExoGen program contains hundreds of such preconditions (which need to be verified or maintained), as well as the resources usage.

We shall now describe the ExoGen language features.

C.1 Request Launching Context

Contexts are used to describe states that are either required or forbidden to launch a request. Thus we have contexts to *prevent* request execution (`fail`) and contexts *required* for request execution (`check`). Moreover, these contexts can be checked *before* launching (`precond:`), and *while* the request is running (`maintain:`)

The contexts are conjunctions of predicates. We have three predicates:
Active(request(?arg)[with cstr$^+$]) is true when an instance of *request* satisfying *cstr* is currently running.
Last_Done(request(?arg):?ret[with cstr$^+$]) is true when the last correctly terminated instance of *request* satisfies *cstr*.
Resource tests example: $BatLevel < 10$.

The constraints can be of the following types: range of a variable, comparison of a variable with a constant value. They can be defined over the arguments and, for the `Last_Done` predicate, the results of the requests.

Example: To express the fact that we cannot call `xr4000_GotoRelative` during a moving request execution one writes:

```
request xr4000_GotoRelative(?_) {
  fail {
   maintain:
     Active({band_Move(?_) | nd_Goto(?_) |
             nd_ExecTraj(?_)});
  }}
```

C.2 Mutual exclusion between requests

ExoGen includes a control structure `mutex` to easily describe mutual exclusion between requests. In a `mutex` we declare requests that must be excluded (i.e. only one of these requests can run at the same time) and the contexts when this mutual exclusion is active. Similarly to request launching contexts (see IV-C.1), mutual exclusion contexts can be checked *before* (`precond:`) or *during* (`maintain:`) requests execution.

Example: Thus, the fact that we must have only one moving request at the same time can be described with:

```
mutex {band_Move(?_), nd_Goto(?_), nd_ExecTraj(?_)} {}
```

This will be understood by ExoGen as:

```
request band_Move(?_) {
fail {
  Active({nd_Goto(?_) | nd_ExecTraj(?_)});
}}
request nd_Goto(?_) {
fail {
  Active({band_Move(?_) | nd_ExecTraj(?_)});
}}
...
```

C.3 Resources

We distinguish two types of resource:
sharable: The resource (such as battery power) is borrowed during request execution and released at the end of execution.
depletable: The resource (such as battery load) is consumed/produced by request execution[3].

To describe a resource, we have to declare its type, its name and to indicate its location in the functional module description (most likely in a GenoM poster). Example:

```
depletable batLevel: auto(Battery_State.level);
sharable batPower: auto(Battery_State.power);
```

Resources usage is declared with: `use`(*value*) and `produce`(*value*). For instance to describe the battery usage of the `xr4000_GotoRelative` request we write:

```
request xr4000_GotoRelative(?_) {
  uses {
    batLevel: use(1.2);
    batPower: use(3);}
  [...]}
```

D. Internal Model Structures

The ExoGen compiler produces an internal structure similar to an OBDD[4] [10], named OCRD for Ordered Constrained Rule Diagram. A description of how it is built can be found in [11]. This data structure offers real-time guarantees (as we can give a maximum traversal time) and can formally validate some temporal properties via a model checking approach.

V. Experimental results

We implemented an R2C on our XR4000 Nomadics: Diligent (see Figure 5). In its current configuration, Diligent has the architecture presented on Figure 3. There are currently only few rules but the results are quite encouraging.

With an ExoGen declaration containing six binary `mutexes`, one ternary `mutex` and one `fail` context we build the resulting OCRD in 67 seconds[5] – including

[3]Our model remains pessimistic, as the consumption is done at the launch of the request and the production at the completion of the request.
[4]OBDD: Ordered Binary Decision Diagram.
[5]The machine used is a Sunblade 100 with 512 Mb of memory.

Fig. 5. Diligent

40 seconds of optimization. The resulting OCRD has a maximum depth of 17 and the maximum node traversal number before a controllable one is 13. Therefore, at best, the resulting R2C will make 13 tests before doing any action and the maximum delay of execution of the R2C corresponds to the traversal of 17 nodes (i.e. 17 tests/actions). After simplification and optimization of the diagram, the graph contains 552 nodes. Note that, similarly to OBDDs, the resulting graph size (given by the number of node) depends of the predicate order. So we added an optimization algorithm based on the sifting method (see [12]) adapted to the OCRD data structure.

On our Nomadics XR4000, the average traversal time of the decision diagram in the R2C is 100 microseconds.

VI. Conclusion and Future Works

We have briefly presented the LAAS architecture, its components and its integrated tools. Then the presentation focuses on the Execution Control Level of this architecture and its main component the Requests and Resources Checker (R2C). This layer of the LAAS architecture has a critical role with respect to safety and faults protection. It must guarantee that the functional modules which "act" on the real physical system are properly controlled and do not engage in "dangerous" situations with respect to the "commands" coming from the decisional level.

Thus the R2C and its associated tool ExoGen propose a language in which the user can specify, using the GenoM namespace, the contexts in which a particular request (with constraints on its arguments) can be executed.

Our approach using OCRD is similar to OBDD, thus it eases the formal verification of the generated model. Moreover, it offers some real-time properties such as the guaranty of the maximum time taken to check new requests. Another interesting property inherited from OBDD is the reduction of input formula to a canonical form. The generated DAG is relatively compact (depending on predicates ordering method) and is unique. The counterpart of this reduction property is that the completeness checking of a declaration[6] is harder than with a complete representation.

Our objective in the future is to experiment formal verification tools based on model-checking techniques applied to OCRD and execution control problem. Another considered extension is to add temporal informations to the description of the R2C, to allow more complex (Reachability of a state from a particular state, etc).

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming*, vol. 17, no. 4, pp. 315–337, April 1998.

[2] F.F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots," in *IEEE International Conference on Robotics and Automation*, Mineapolis, USA, 1996.

[3] S. Fleury, M. Herrb, and R. Chatila, "Design of a modular architecture for autonomous robot," in *IEEE International Conference on Robotics and Automation*, Atlanta, USA, 1994.

[4] A. D. de Medeiros, R. Chatilla, and S. Fleury, "Specification and Validation of a Control Architecture for Autonomous Mobile Robots," in *IROS*. 1996, pp. 162–169, IEEE.

[5] B. Espiau, K. Kapellos, and M. Jourdan, "Formal verification in robotics: Why and how," in *The International Foundation for Robotics Research, editor, The Seventh International Symposium of Robotics Research*, Munich, Germany, October 1995, pp. 201 – 213, Cambridge Press.

[6] F. Boussinot and R. de Simone, "The ESTEREL Language," *Proceeding of the IEEE*, pp. 1293–1304, September 1991.

[7] E. Rutten, "A framework for using discrete control synthesis in safe robotic programming and teleoperation," *IEEE International Conference Robotics & Automation*, pp. 4104–4109, May 2001.

[8] R. P. Goldman and D. J. Musliner, "Using Model Checking to Plan Hard Real-Time Controllers," in *Proc. AIPS Workshop on Model-Theoretic Approaches to Planning*, April 2000.

[9] R. Simmons, C. Pecheur, and G. Srinivasan, "Towards automatic verification of autonomous systems," in *IEEE/RSJ International conference on Intelligent Robots & Systems*, 2000.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computing*, vol. 98, no. 2, pp. 142–170, 1992.

[11] F. Ingrand and F. Py, "Online execution control checking for autonomous systems," in *International Conference on Intelligent Autonomous Systems*, Marina del Rey USA, March 2002.

[12] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams.," in *IEEE/ACM ICCAD'93*, 1993, pp. 42–47.

[6] That all states are specified.