Real-Time Execution Control for Autonomous Systems

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

Abstract

There is an increasing need for advanced autonomy in complex embedded real-time systems such as robots, satellites, or UAVs. Still, the growing complexity of the decision capabilities of these systems raises a major problem: how to prove that the system is not going to end in a dangerous state (for itself or for humans)? How to guarantee that the robot will not grab a sample on the ground with its arm, while moving (which could supposedly break the arm)? How to make sure that the satellite RCS jets are not fired when the camera lens protection is off? How do we make sure that a service robot for elderly people is not moving faster than $20cm.s^{-1}$?

This paper presents some recent developments of the LAAS architecture for autonomous systems. 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 system (through the functional level). It acts as a real-time "safety bag"[1], to make sure that the commands issued are consistent with the current state of the system and with a formal model of the acceptable states. To implement this component, we present a new approach and a new tool inspired by the model checking domain. We introduce a new language (ExᴼGen) to specify the model of acceptable and required states of the system (valid contexts for requests to functional modules and resources usage). This language is compiled offline in an OBDD (Ordered Binary Decision Diagram) like structure which is then used online to check the specified constraints in real-time. This tool is seamlessly integrated in the LAAS architecture and relies on the other tools used to build autonomous systems ($G^{en}{}_{o}M$, OpenPRS, etc). We have deployed it on a number of robotics platforms (ATRV and XR4000 robots). We show that such an approach allows us to improve the runtime dependability of the system at a minimal acceptable cost (compared to the possible loss of the complete system), but could also be extended to check more complex temporal properties of the system off line.

Keywords

Architecture for Autonomous Systems, OBDD, Dependability

1 Introduction

Advanced robots or satellites have an increasing need for a high level autonomy while performing in a hard real time environment. However, this raises a major non trivial problem: most complex autonomous systems, which operate with a minimal human intervention and in a highly non deterministic environment are hard to validate. Nevertheless, these systems must be safe and dependable (e.g. avoid non nominal and unknown system states), to avoid financial loss and/or to avoid disturbing/harming humans. These two requirements (high level autonomy and dependability) may appear contradictory or at least hard to satisfy together. Indeed, how can we be

[1] This term was first introduced in this paper [1] and is used by the Dependable Computing community to talk about systems which have an ultimate fault protection mechanism such as the one presented here.

sure that a high level autonomous system with rather strong decisional capabilities will not take a decision that could threaten the mission and/or the humans?

An initial response is to use a *safe* decisional component (e.g. a high level planner giving safe plans). However, in most autonomous systems the decisional component cannot deal with a complete system model. Indeed such a large model will render the planner unusable. Taking into account the model, the environment and the goals would lead to such a huge state space that no search would be able to extract a plan in a reasonable time. So we have to stick with a planning model which is "limited" to remain "tractable", and provide other means to check the system dependability (w.r.t. the decisional aspect). As a limit to the system planning abilities, we may rely on predefined human written plans or procedures which are in most cases hard to validate (except by test) and whose concurrent execution may lead to unexpected outcomes.

The LAAS[2] architecture, described in section 2, proposes a solution with an execution controller to address this problem. The execution controller role will be presented in section 3. While we introduce a solution in our architecture: the Request & Resource Checker (R^{2}C) in section 4. Section 5 focuses on the ExoGen tool used to automatically generate the R^{2}C from the system constraint specifications. We will conclude with some experimental results and on the perspectives offered by this work.

2 The LAAS Architecture

The LAAS architecture [2] 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 levels, with different temporal constraints and manipulating different data representations. From the top to the bottom, the levels are:

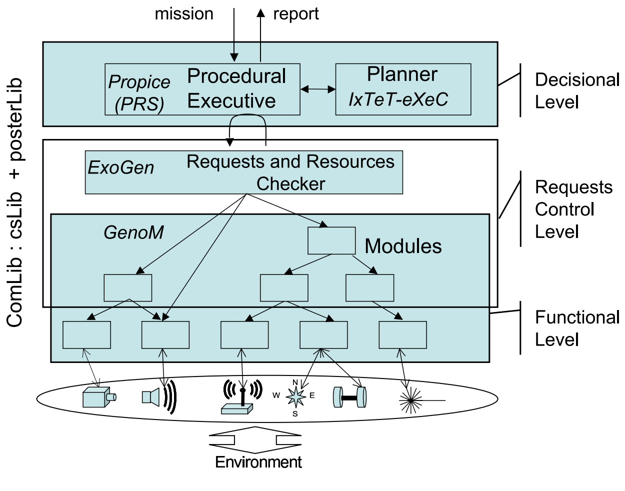

Figure 1: The LAAS Architecture.

- *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 [3], which is connected to the underlying level, to which it sends requests that will ultimately initiate actions and start treatments. It is responsible for

[2] LAAS: LAAS Architecture for Autonomous Systems

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 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 [4]. 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, OpenPRS 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).

3 Execution Control

The execution control has a critical role in the LAAS architecture. Considering that the interactions between the functional and the decisional levels are somewhat difficult to validate, the execution control acts as an intermediate component which controls online the system behavior and avoids inconsistent system states. Still, we need to specify what has to be taken into account to ensure that this component will effectively control the system behavior and keep the system in a nominal state.

During normal operation, the decisional level commands the functional level. The functional level is deployed using a strongly modular approach which helps developers to easily add new functionalities to the system. As a result, none of the modules has a real knowledge of other modules existence and behavior. So if each functional module can (and usually does) check its own status, behavior and its inputs/outputs, still it cannot make any assumption on the interactions it will have with other modules (a particular module may be responsible for running the robot at a particular speed provided by another module, but it has no idea what an acceptable speed is in this particular context). So bluntly said, the functional modules alone cannot provide inter-module conflict detection nor can they check for conflicts coming from the decisional level.

The execution control is the solution we propose to address this problem. It performs this conflict detection by enforcing some of the system constraints.

3.1 Execution control role

The development of the execution control level is based on a simple yet efficient general principle:

> If we are not able to validate neither the functional level interactions, nor the interactions between the decisional and functional levels, then we can insert a simple synchronous component which will monitor the system state evolution in real time and check that this evolution is conform to a formal model of acceptable states.

This component must satisfy the following requirements:

- *Observability* The component must have the ability to monitor and catch all events that may lead or participate to a system inconsistency. Indeed, the execution control requires this information to properly monitor the evolution of the system..
- *Controllability* The execution control must be able to control the system (i.e. block or deny commands) sufficiently to avoid inconsistent states. If this requirement is not respected then there is no way to avoid these states.
- *Synchronous and Bounded cycle time.* The component must act under a synchronous hypothesis (i.e. computation and communication take virtually no time). Apart from avoiding asynchronous formalism difficulties, this allows us to have a cleaner formal model of the system behavior and states transition. In practice, this implies that the system will run as a loop with a bounded cycling time, which will also offer some guarantees on the overall system reactivity.
- *Verifiability.* The execution control component has to offer a formalism and a representation that allow the developer to check if it safely controls the system behavior.

The execution control component acts has a “safety bag” [1]: it captures all the incoming events (request of services, end of services, ...) and, according to some specified constraints, enforces the resulting state validity. This may results in requests being rejected or killed.

3.2 A simple example

Let consider an autonomous robot which is able to move with a maximum speed of $2m.s^{-1}$ and which has three ways to localize:

- one based on stereo pixels tracking,
- one based on GPS.
- one based on odometry.

As shown on figure 2, the functional level is composed of 8 modules:

- *Wheel Control:* controls the wheel speed and orientation. The input of this module is the speeds (linear and angular) taken from the *Path Planning* module. It also produces the position data using odometry.
- *Pan/Tilt:* controls the camera pan/tilt position.
- *GPS :* gives data retrieved from GPS.This module produces two data: the position and the GPS reception level.
- *Camera:* is used to take images from the stereo bench.

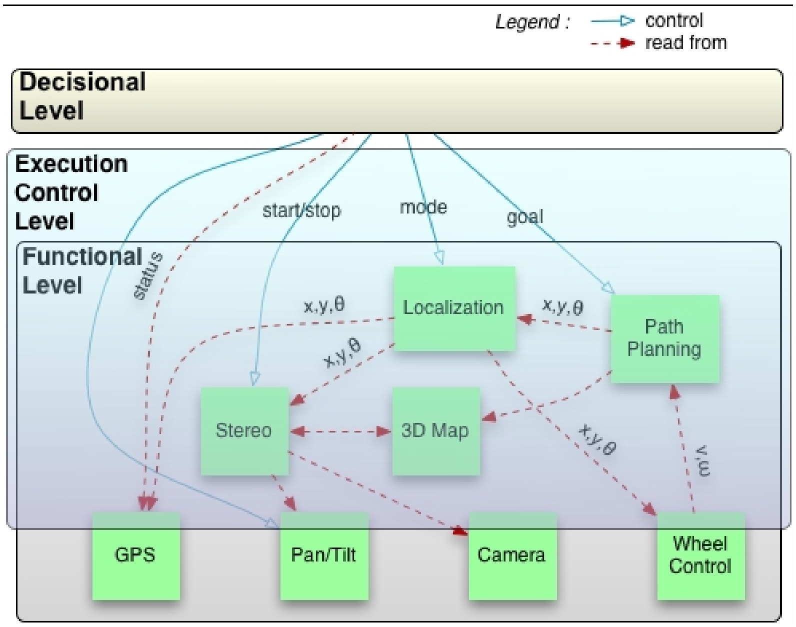

Figure 2: Example of a robot architecture

- *Stereo:* computes a stereo correlation image. It exports a local 3D representation of the environment and a position. This module may be started or stopped. Indeed this functionality is complex and uses a lot of CPU resource so it can be useful to stop it when not used.
- *3D Map:* manages a 3D map of the environment according to the data retrieved from *Stereo*.
- *Localization:* acts as a selector between the 3 modules producing a position. The decisional level controls the position mode which is being used.
- *Path Planning:* computes a path based on *3D Map* to reach a given goal position. It offers one service which takes the goal position as an argument. This service terminates when the robot has reached the goal or when an error occurs.

The system is controlled by the decisional level. This level controls the pan/tilt camera position, starts or stops the *Stereo* service, changes the mode of the *Localization* module and may give a goal position to *Path Planning*. It also reads the GPS reception level.

Here are the constraints we want to enforce:

- The localization must be valid. We do not want to take the position from *Stereo* when it is stopped or from *GPS* when reception level is not sufficient.
- If the localization is not valid, we are not able to execute path planning. Thus we should reject requests to move to a new goal.

In this example the role of the execution control level is to guarantee the satisfaction of these constraints.

3.3 Similar works

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 [5]. 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 a representation is the guaranty of a maximum evaluation time (corresponding to the decision DAG[3] 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[6]. This development environment, based on the ESTEREL [7] 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 [8, 9], the authors present 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 [10], 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 the system on-line, with respect to these constraints. This automaton can be formally validated with model checking techniques.

In [11] 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 at the decisional level.

Another approach to consider is IDEA presented in [12]. It relies on two main ideas: (1) most components can be seen as agents which share a common virtual machine defining their reactive planning behavior (planning here has to be taken in a wide sense);(2) all these agents share parts of a global temporal model which specifies the internal "behavior" of the agent, as well as the communication between agents. The time-lines representation of constraints supporting this architecture seems to be a good model for a formal validation of the system.

4 The Request & Resource Checker

We present now our execution control component. This component called the Request & Resource Checker (R^{2}C, see figure 3), acts as follows:

- *Events capture:* The R^{2}C captures all the events that may change the system state.
- *Update of system state database:* According to these events the R^{2}C updates its system state representation.
- *State Checking:* It checks the requested state validity. If it is not valid, then it deduces actions to do to keep the system in a safe state.
- *Actions launching:* It executes deduced actions and sends reports to concerned components. These reports are informations about R^{2}C deductions (rejection, ...) or external events (end of service, error, ...).

The main component of R^{2}C is the state checker. This component encodes the constraints of the system. To specify these constraints we have defined a language named ExᴼGen.

[3] DAG: Directed Acyclic Graph.

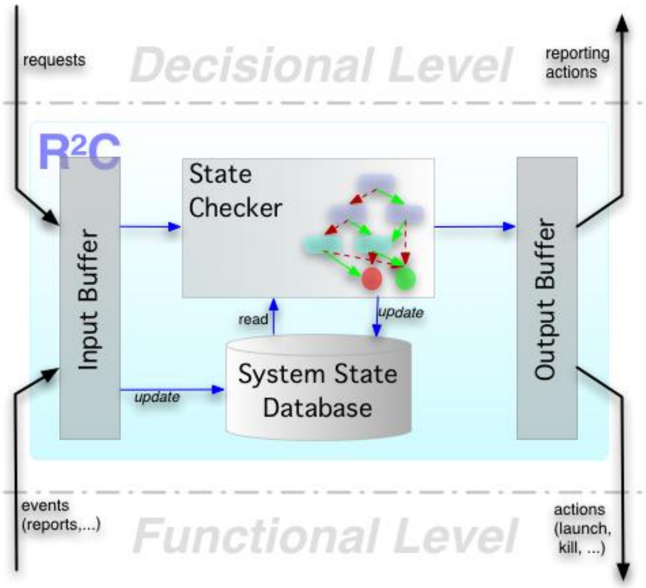

Figure 3: The Request & Resource Checker

5 Controller Synthesis

ExoGen is a compiler which generates the R^{2}C corresponding to a set of constraints specified for one particular system. These constraints are expressed thanks to a set of predicates:

- *Service past:* It tests if the last service instance correctly terminated satisfies the constraints given for its inputs and returned values.
- *Precedence relation between services:* Tests if the last instance of one service has finished before the last instance of another one.
- *Resource value test:* It checks if a resource value is in one particular domain. Generally resources are linked to one external data, for example battery level may be exported by one module, but we can also specify "virtual" resources. A virtual resource may be the cameras with a capacity of 1. We distinguish two kinds of resources:
  1. shared resources: they correspond to resources which are used during service execution and replenished at the end of this service.
  2. depletable resources: these resources are just consumed except for special services replenishing them explicitly.
- *Instance currently running:* It tests if one service instance is currently running. Instance arguments have to satisfy a specified constraint.

These predicates are used to describe contexts that block or are necessary for the execution of a particular service. Users can also specify if these contexts may be checked as a precondition or as an invariant.

Following is an example of an ExoGen specification for the *goto* service of the *PathPlanning* module :

```
request PathPlanning_goto(?goal) {
  fail {
  maintain:
    past(Localization_set(?mode) with ?mode==STEREO) &&
      ( !past(Stereo_start()) || [Stereo_start()<Stereo_stop()] );

    past(Localization_set(?mode) with ?mode==GPS) && GPS_status in [0,0.1];
  }
}
```

In this example the *goto* service cannot run in 2 cases:

1. the *Localization* mode is `STEREO` when the *Stereo* was never started or the last time it was started precedes the last time it was stopped.

2. the *Localization* mode is `GPS` when the *GPS* reception level is less than 10%.

These two constraints are checked as invariant (`maintain:`), so if one of these becomes true then all the running instances of *goto* must be stopped.

5.1 From constraints to control

The constraint specification is used by Ex$^{\mathrm{o}}$Gen to generate the corresponding state checker. This component is based on a structure quite similar to OBDDs[4] [13] named OCRD[5][14]. The choice of an OBDD like structure was driven by the following properties:

- OBDDs are well known to represent first order logic formulas in a canonical and compact form. They are often used to represent graph transitions.

- The resulting data structure is a Directed Acyclic Graph (DAG) so the traversal time is bounded and linked to the graph depth.

- OBDDs are frequently used in model-checking which is an interesting technique to formally validate a model[15].

OCRDs are directed graph quite similar to OBDDs.Terminal leaves of the graph correspond to true ($\top$) or false ($\bot$) results. Each node is a branching test for one variable state: one branch is fired when the result of the test is true and the other when it is false (i.e. each node is a kind of `if then else`). OCRD construction is quite similar to OBBD's one (see [16]). The main difference relies in the fact that nodes are not first order logic predicates. In an OCRD a node has the following form:

$$pred(?v_1 \ldots ?v_n) \; with \; cstr(?v_{i_1} \ldots ?v_{i_m})$$

with:

$pred$	a predicate.
$?v_{1\ldots n}$	variables linked to $pred$.
$cstr(?v_{i_{1\ldots m}})$	constraint on a subset of variables of $pred$, this constraint is a fixed hypercube.

To keep OBDDs properties we have to be sure that each OCRDs node does not depend on another one. Thus the following property has to be satisfied:

$$\forall pred, \forall(cstr_1, cstr_2) : cstr_1 \neq cstr_2 \rightarrow \left(\nexists(?v_{1\ldots n}) : cstr_1(?v_{1\ldots n}) \wedge cstr_1(?v_{1\ldots n})\right)$$

Where $cstr_1$ and $cstr_2$ are constraints applied to $pred$ in the set of nodes of the OCRD.

This property is maintained during the construction of the OCRD. The compiler splits domains of each predicates into a partition when needed.

[4] OBDD: Ordered Binary Decision Diagram

[5] OCRD: Ordered Constrained Rule Diagram

Given this property, each node of the OCRD corresponds to a subset of the state space of $pred$. The resulting OCRD is equivalent to an OBDD where each variable corresponds to a partition of states for each predicate.

The OCRD constructed by Ex^{o}Gen is the general system formula expressing all the valid transitions (i.e. to valid states). Ex^{o}Gen then extracts from this formula a component which role is to maintain this formula true ($\top$) taking into account controllable events and the uncontrollable predicates (incoming events) .

The controllable predicates are the expression of system controllability. They represent the actions the R^{2}C can use to avoid inconsistent state. If there is no controllable predicate then it is impossible to control the system.

The OCRD computed from the Ex^{o}Gen code given in the previous section is illustrated in figure 4. We can see that uncontrollable predicates are checked first – such a node order simplifies the compilation – these predicates are representing current state of the system and incoming events. The controllable predicates are the decisions made by the state checker to maintain the system in a good state. The state checker will just try to maintain this formula true setting controllable predicates values according to this.

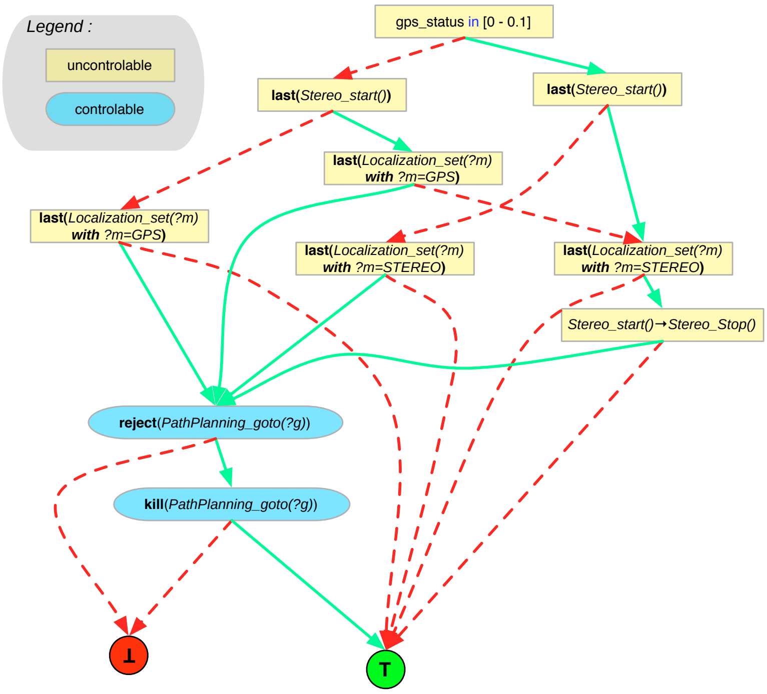
Legend :
uncontrolable
controlable
gps_status in [0 - 0.1]
last(Stereo_start())
last(Stereo_start())
last(Localization_set(?m) with ?m=GPS)
last(Localization_set(?m) with ?m=GPS)
last(Localization_set(?m) with ?m=STEREO)
last(Localization_set(?m) with ?m=STEREO)
Stereo_start()→Stereo_Stop()
reject(PathPlanning_goto(?g))
kill(PathPlanning_goto(?g))
⊥
⊤

Figure 4: resulting OCRD from example

Figure 5: The Dala robot

6 Experimental results

We have implemented and integrated the R^{2}C on the LAAS architecture and the results are quite encouraging. On our ATRV robot, Dala (see figure 5), we have specified 13 context constraints expressing the functional level modules dependencies and conflicts. The resulting OCRD has a maximum depth of 22 with a total size of 399 nodes.

The fault injection of conflicting situations is properly detected and taken care of. The services instances leading to an inconsistent state are properly rejected or killed and the functional level remains in the set of acceptable states. Compiling and building this execution controller took less than 0.10s[6].

An interesting result is the fact that the R^{2}C helped us detect one faulty behavior in one of the decisional level components. The error, due to a coding mistake, was avoided and reported to this component. Unfortunately, as of today, this component has not been programmed to take into account this kind of messages, although it is rather simple to consider these messages as a service failure and to react accordingly. The integration of R^{2}C does not have any noticeable impact on the global reactivity of the system and is able to deduce its action with a maximum time of 300 μs.

7 Conclusion & Future Works

In this paper, we first justify why one needs execution control in autonomous systems, and show that the LAAS architecture for autonomous systems was originally designed taking this concern into account.

We present the R^{2}C, together with its development tool ExoGen, a component which offers a simple, but yet powerful solution to the execution control problem. This component was designed to satisfy the specifications of an execution control component: observability and controllability of the functional level, synchronous and real-time, verifiability of the model used.

The R^{2}C is supported by a data structure similar to OBDD, named OCRD, which encodes the acceptable states of the system. The resulting diagram has a limited depth and thus provides a real time guarantee on its maximum evaluation time. Using an approach equivalent to OBDD, we should be able to use model checking tools to validate some more complex temporal properties of the R^{2}C, thus offering some more formal validations of the model.

The R^{2}C is currently integrated in the LAAS architecture and our first tests show that it performs efficiently on our mobile robots. Still, the current version does not have a complete view on the state change as it just captures events coming from the control flow of the system (requests/reports) and not the data exchanged by data flow (posters reading). We are currently adding this feature to the latest version of the R^{2}C. Another possible extension is to enhance the decisional components to take into account the reports coming from the R^{2}C, i.e. how to recover from a rejected or killed request. Last, we plan to investigate existing model checkers approaches (based on OBDD) to see if they can bring some new advantages to this system.

References

[1] P. Klein, "The Safety Bag Expert System in the Electronic Railway Interlocking System ELEKTRA," *Expert Systems with Applications*, pp. 499–560, 1991.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Time results obtained on an i686 with 1GB of RAM under Linux or a sun-blade 100 with 640 MB of RAM under Solaris 8.

[6] 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.

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

[8] 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.

[9] F. Maraninchi K. Altisen, A. Clodic and E. Rutten, “Using controller synthesis to build property-enforcing layers,” in *European Symposium on Programming (ESOP)*, Apr. 2003.

[10] 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.

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

[12] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt, “Idea: Planning at the core of autonomous reactive agents,” in *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, October 2002.

[13] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[14] F. Ingrand and F. Py, “An execution control system for autonomous robots,” in *IEEE International Conference on Robotics and Automation*, Washington DC (USA), May 2002.

[15] 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.

[16] K. Brace, R. Rudell, and R. Bryant, “Efficient Implementation of a BDD package,” in *Design Automation Conf.*, 1990, pp. 40–45.