Dependable Execution Control for Autonomous Robots

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

***Abstract*— This paper presents a new approach to integrate real-time execution control on autonomous systems and how such an approach integrates in their software architecture. The use of decisional autonomy is becoming more widely accepted as a solution to the increasing need to deploy complex systems (robots, satellites, etc) able to perform non trivial tasks in various environments. We present an overview of the organization of such systems. Then we explain why the increasing complexity of functional components as well as the presence of autonomy components become an obstacle to system safety and dependability. To address this issue, we propose the integration of an execution control component in the software architecture. This component is synthesized from a model of the acceptable and dangerous state using model-checking techniques. The execution controller has a generic representation of system behavior and, according to some specified system constraints, acts as a "safety bag" allowing acceptable states and avoiding forbidden ones. The controller uses an OBDD[1] like data structure which offers a bounded execution time, and which can be formally validated offline to check temporal properties. Real experimentations have been made on our autonomous mobile robots, and have confirmed it can catch in real-time design errors from the decisional components which would have lead to disastrous consequences.**

I. Introduction

Advanced systems such as 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. First because little has been done in the field of validating autonomous components[2], second the increasing number of the functional components, their raising intrinsic complexity and their interactions is becoming a serious concern to be addressed by system architects. Nevertheless, if these autonomous systems are to be seriously considered (e.g. for costly missions or for interacting with humans) one must propose approaches to make them safe and dependable (e.g. avoid non nominal and dangerous system states, which could lead to the loss of mission or to harm humans).

The paper is organized as follow. We present in section II a "classical" autonomous system architecture and the reasons why it is rather difficult to prove safety and dependability of autonomous systems based on this architecture. A solution to address this problem is to introduce a component acting as a "safety bag" [2] to control that it will never let the system engage in inconsistent state. The role and the requirements of this component are further detailed in section III. The Request & Resource Checker – presented in section IV – is specified according to these requirements. It is supported by a formal representation of the system described step by step in section V. This model is used to specify the constraints of the system and to generate the corresponding controller based on an OBDD like data structure described briefly in section VI. Section VII presents some encouraging experimental results on one of our robotic platforms. We then conclude and present some of the prospectives opened by this work.

[1]OBDD: Ordered Binary Decision Diagram

[2]This does not mean that they are intrinsically harder to validate than traditional program. But various real world experiences [1] show that the trust people put in these components is, as of today, rather limited.

II. Autonomous Systems: The Complexity Dilemma

An autonomous robot is commonly seen as a system integrating perception and action in a dynamic environment and, most important, deliberative capabilities, with minimal human supervision. These systems make an increasing use of advanced and complex techniques in order to enhance their robustness to environment variability.

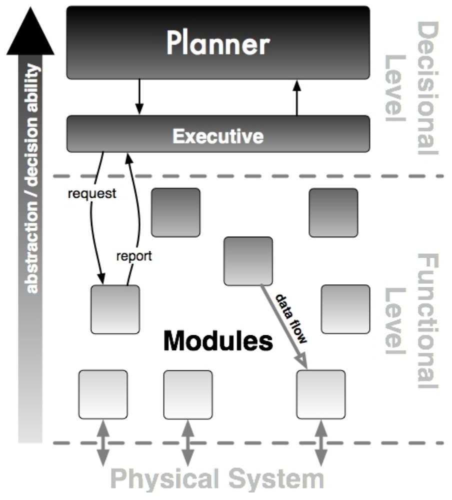

Fig. 1. A Generic Robots Architecture

As illustrated in Fig. 1, most autonomous robots architectures [3], [4] are organized in two main layers or levels:

- **Decisional Level:** This level centralizes the high level decisional capabilities of the system. It may embeds a number of components such as (but not limited to): a *planner*, which produces high level plans to achieve goals, and an *executive* which decomposes and refines plans into atomic actions executable by functional components.

- **Functional Level:** This level, controlled by the decisional one, includes all the basic system functionalities (sensors, effectors, ...) and processing (motion planning, image processing, ...). It may be organized into modular components including a set of services corresponding to one particular functionality or to a physical component.

On one hand, the decisional level provides some high level decision-making capabilities, however such capabilities require a global and somewhat abstract view of the system. Indeed, the complexity of the overall system cannot reasonably be globally encompassed by the planner or the executive. As a consequence, the model used at this level must remain at a level of abstraction low enough to enable the system to reason about it in an acceptable time, while being high enough to make their use non trivial.

However, using such high level partial model in say the planner is bound to lead to plans which first need to be refined using yet another partial model (for example a refinement procedure) and at a lower level. This aggregation of partial and independent models (actions planning, procedures, etc) leads to an overall incompleteness of the set of the states covered by the models, and thus to unforeseen interactions or execution traces which can be harmful to the system. To take an example, if one consider a plan produces by a planner, while being perfectly valid with respect to the actions model used by the planner; its execution through a procedural executive providing parallel procedures execution may lead to unexpected effects due to the interactions between the resulting commands issued by this executive.

Another point to consider with respect to decisional components is their growing complexity. Some of these tools offer functionalities like planning under uncertainty, executing learned policies, least committed plans or learning abilities. The introduction of such techniques provide an increased robustness to environment variability but makes these components far more complex to validate. We are then unable to certify that such program will never trigger an action in a particular state which may lead to an unwanted system states.

On the other hand, the functional level embeds modules with little decisional capabilities and close to the hardware and to the environment. Moreover, to increase the modularity of this layer, and the re-usability of these modules, each of them is developed independently and has little knowledge about the others. As a consequence, they are not aware of conflicts or unforeseen interactions which may arise. Moreover, two or more modules may exchanges data (one producer, and one consumer) through link created by the executive. For example, let us consider an example with two modules:

1) The *arm* module controls the arm of the robot and can receive commands to stow or unstow the arm.
2) The *locomotion* module is used to get the robot moving.

Both modules have been developed independently, still the overall design is such that at no time, the robot should move while its arm is not in the stowed position. It is clear that from each module point of view nothing a priori prevents it to accept an unstowed request (respectively a move request) while the robot is moving (respectively while the arm is an unstowed position).

Similarly, one could consider a third module *motion control* which produces a speed reference (e.g. from a motion planner trajectory) executed by the *locomotion* module (through a data flow link established by the executive). If both *locomotion* and *motion control* modules have been developed independently and are indeed used on various platforms, one can imagine that they are both accepting to produce/use a wide range of speed. For a particular robot in a particular environment though, it is clearly inappropriate to allow an arbitrary range of speed. For example a speed of $50cm.s^{-1}$ may be acceptable in a flat terrain for a regular ATRV but not in a rough terrain. While such constraint may not hold for an UMV like vehicle.

Last, for an autonomous robot, it is usually impossible to make sufficient simulation tests to cover the high variability of the environment and the different situations that may occur. So when the real system will be deployed, it is bounded to run in unforeseen and untested situations.

Still, autonomous robots are intended for missions where safety and dependability is critical: they may have to interact directly with humans (museum guide, nurse robot, ...) or they may have to perform missions in dangerous or still unreachable environments (to the human) where any mistake could have dramatic consequences (nuclear plants, extraterrestrial exploration, ...). Even if high level decision capabilities and complex functional level are the key to achieve these ambitious goals, autonomy will remain largely unused if the software architecture does not offer reasonable safety and dependability. The solution we propose is based on an execution control presented in the next section.

III. Execution Control

The main idea supporting execution control is to offer a component that controls the system will never reach an inconsistent state. It is a fault protection system acting as a filter or "safety bag"[2]. It captures all events that can change the system state (request of services, ...) and checks if they do not lead into a prohibited state. Then it eventually proposes alternate actions (rejecting requests, killing services, suspending processes, ...) to keep the system consistent.

In this paper, we shall consider that the functional level is composed with a set of modules, each of them offering a set of services. These services can be launched by a request with arguments, they can normally terminate or can be killed. In any case, a report is sent to the requester giving information about the action (success, failure, specific information, ...). On the top of this client/server protocol, we also assume that services can provides data flows exporting informations to other programs (modules or decisional components).These assumptions are clearly not over constraining with respect to the classical architecture used for autonomous systems, for example:

- In the LAAS architecture [3] services are activated by requests and may export data to "posters" offering data exchange between the modules and other components.
- In CLARAty [4] functional modules are object instances, then we can consider that a method call is a request for a service and public attributes may be viewed as data flows.

To control efficiently this functional level, the execution controller must respect some basic 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) to avoid inconsistent states. If this requirement is not validated 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 state transitions. In a practical way, this implies that the system will run as a loop with a bounded cycling time, offering, by this way, 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.

As the decisional components control the functional ones, the execution control component takes place naturally as an interface between these two layers. Our experience has proven that it is not sufficient, indeed the interaction between modules is also done by data flows between them created by the decisional level. For example the motion execution module reads the speed vectors from another module computing the path to get to the next way point. The decisional level just manages the creation and destruction of these data flows but does not explicitly see the exchanged data. So the execution control component also needs to monitor the interactions between the functional components. Thus it fits in between the decisional level and functional level as well as between the functional modules which interact between each other.

A. State of the Art in Execution Control

Many of these concerns are not new, and some autonomous system 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 architecture [3] based on the KHEOPS system [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 resource 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 extensions to specify robots "tasks" and "procedures". However, this approach does not address architecture with advanced decisional level such as planners.

In [8], the authors propose a system based on a model-based approach. The objective is to abstract the system in a state transitions based language abstracting the dependability concerns. The programmers specify state evolutions with invariants and a controller will execute this maintaining these invariants. To do that the controller estimates the most likely current state – using observation and a probabilistic model of physical components – and find the most dependable sequence of commands to reach specified goal (i.e. with a minimum failure probability).

[3]DAG: Directed Acyclic Graph.

In [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 makes use of an abstract representation of services (without explicit representation of arguments nor returned value). 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) – the executive language of the CLARAty architecture[12] – 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.

Another approach to consider is IDEA presented in [13]. 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.

IV. THE REQUEST & RESOURCE CHECKER

Our proposal for execution control is a software component named Request & Resource Checker (R^2C). As shown on Fig. 2, it acts as follows:

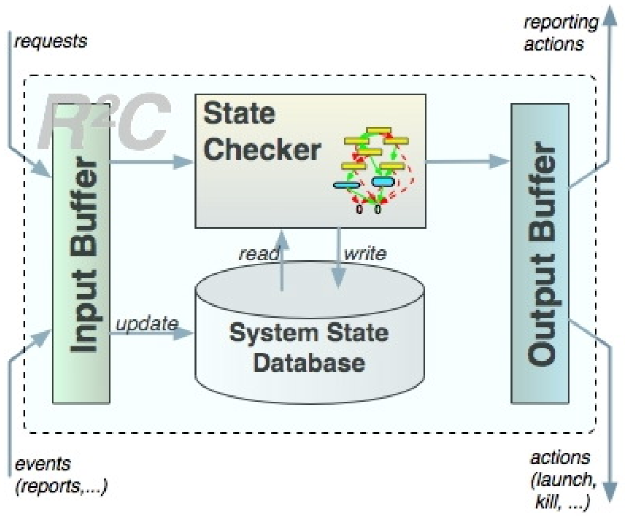

Fig. 2. The Request & Resource Checker

- *Input Buffer* captures the incoming events[4] from the system. Possible events are:
  1) Requests for new services: generally coming from the executive.
  2) End of services, or reports: Coming from functional modules. It gives informations about the causes of the services completion (success, failure, ...) and, possibly, some output values.
  3) Data changes. These data may represent the level of a resource (battery power, fuel level, ...) or data exchanged between components.
- *System State Data-base* maintains a representation of the system state built by the R^{2}C from the flow of events captured in the *Input Buffer*, and the previous system state. If the state has changed, it activates the *State Checker*.
- *State Checker* checks if incoming events are not leading to an inconsistent state and deduces actions to avoid it.
- *Effector* launches actions deduced by R^{2}C and reports its deductions to the clients of the services.

The main component of the R^{2}C is thus the *state checker*. It encodes the constraints of the system which specify the acceptable and unacceptable states. To specify these constraints we have defined a language, named ExoGen, to model the system and its evolution. The model we use is presented below.

V. System constraints description

As the R^{2}C maintains the functional level consistency, it has to be supported by a model of this layer. We propose here to describe step by step the model we use to represent the system and its constraints.

A. Model of services

Let $\mathbf{S}$ be the set of all the services offered by the modules, and $\mathbf{P}$ the set of running processes. We define the predicate *instanceof*: $\mathbf{P} \times \mathbf{S} \rightarrow \{\top, \bot\}$ expressing that a process is an instance of one service($\top$) or not ($\bot$). At each time point t one can check if a service is running using this formula:

$$\forall t, \forall s \in \mathbf{S} : \\ \textit{running}(s)_t \Leftrightarrow \big(\exists x \in \mathbf{P}, \textit{instanceof}(x,s) : \textit{active}(x)_t\big)$$

Where:

$$\begin{array}{rcl} \textit{active}(x)_{t+1} & \Leftrightarrow & \big(\textit{launched}(x)_t \vee \textit{active}(x)_t\big) \\ & & \wedge \neg \textit{finished}(x)_{t+1} \\ \textit{launched}(x)_t & \Leftrightarrow & \textit{requested}^\circ(x, arg)_t \wedge \neg \textit{rejected}^*(x)_t \\ \textit{finished}(x)_{t+1} & \Leftrightarrow & \textit{end}^\circ(x, ret)_{t+1} \vee \textit{killed}^*(x)_t \end{array}$$

Note: Predicates marked with a $^\circ$ are uncontrollable – they correspond to contingent events – as opposed with the predicates marked with a * , which are the R^{2}C possible actions and thus fully controllable. These special predicates give us information about the controllability and observability of services.

The attributes *arg* and *ret* are respectively the argument value given to the request of service and the report of the service. As the report can indicate one service instance failure we define the predicate *correct* taking a returned value as argument which is true if and only if the returned value indicates a correct process completion.

[4]Events here are all the informations exchanged between the decisional and the functional modules, as well as the one exchanged between modules themselves.

As a process execution has an influence on the subsequent states of the system, we need to know the previous service instances. For example, let consider a service calibrating a laser sensor, if this service fails then we can consider that measures taken from this sensor are not correct. To keep informations of the past services we define the following function:

$$\forall t \geq t_0, \forall s \in \mathbf{S} : \\ \textit{last}(s)_{t+1} = \begin{cases} \textit{undefined} & \text{if} \quad \textit{running}(s)_{t+1} \vee \\ & \quad \big(\forall x \in \mathbf{P}, \textit{instanceof}(x,s), \\ & \quad \textit{end}^\circ(x,r)_{t+1} : \neg \textit{correct}(r)\big) \\ (x, ret, t+1) & \text{if} \quad \neg \textit{running}(s)_{t+1} \wedge \\ & \quad \big(\exists x \in \mathbf{P}, \textit{instanceof}(x,s) : \\ & \quad \textit{end}^\circ(x,r)_{t+1} \wedge \textit{correct}(r)\big) \\ \textit{last}(s)_t & \text{else} \end{cases}$$

With $\forall s \in \mathbf{S} : \textit{last}(s)_{t_0} = \textit{undefined}$.

We also need to know argument of a process. This can be done using the *requested*$^\circ$ predicate. Then we define the *argumentof* function supported by this rule:

$$\forall x \in \mathbf{P} : \textit{argumentof}(x) = arg \Leftrightarrow \big(\exists t : \textit{requested}^\circ(x, arg)_t\big)$$

The past of a service is not sufficient to express all the system evolutions. We also need to have an order relation between service dates, so we can point to the last of the past services. Using the *last* function, we define this precedence test:

$$\begin{array}{l} \forall t, \forall (s_1, s_2) \in \mathbf{S} \times \mathbf{S} : \\ s_1 \prec_t s_2 \quad \Leftrightarrow \Big(\quad \big(\textit{last}(s_1)_t = (x, r_1, \tau)\big) \wedge \\ \qquad \qquad \big((\textit{last}(s_2)_t = \textit{undefined}) \vee \\ \qquad \qquad (\textit{last}(s_2)_t = (y, r_2, \tau') \wedge \tau < \tau')\big)\Big) \end{array}$$

B. Model for data flows

Our system must take into account the data flow between services. As presented in section III, they provide the data sharing mechanism of the system.

First we define $\mathbf{M}$ the set of possible data flows. To access the data flow $m \in \mathbf{M}$ we can specify two special services: *read*$[m]() \in \mathbf{S}$ and *write*$[m](value) \in \mathbf{S}$. Using the *last* predicate on the *write*$[m](value)$ service we can determine the current value of m at each time point.

Then we can extract this rule:

$$\begin{array}{l} \forall x \in \mathbf{P}, \forall m \in \mathbf{M}, \textit{instanceof}(x, \textit{read}[m]) : \\ \big(\textit{end}^\circ(x,r)_t \wedge \textit{last}(\textit{write}[m])_t = (p, v, \tau)\big) \\ \Rightarrow \textit{argumentof}(p) = r \end{array}$$

These two specific services are requested by specific processes. The *write*$[m]$ one can be called only by processes owning m and the *read*$[m]$ may be called by any client of m. To allow system specifier to describe these links we define two predicates: *produces:* $\mathbf{P} \times \mathbf{M} \rightarrow \{\top, \bot\}$ expressing that a process will be able to write a new value on a data flow; and *read:* $\mathbf{P} \times \mathbf{M} \rightarrow \{\top, \bot\}$ expressing that a process will probably read one data flow. These two predicates will be used during the system constraints specification phase to express the producers and consumers of one data flow.

C. Constraint specifications

The model previously presented offers a representation based on service instances. It could be interesting to distinguish instances of one service according to their arguments and/or returned values. Indeed these values frequently give information about the service instance behavior. To express them in a general way, we define service classes (classified by constraints applied to these values). These constraints are fixed hypercubes in the domain of arguments and returned values of the service.

Let $\mathbf{Ca}_s$ be the set of constraints we can apply to arguments of $s \in \mathbf{S}$, we can define a predicate which checks one instance of s satisfying $c \in \mathbf{Ca}_s$ is running:

$$\begin{aligned}&\forall t, \forall s \in \mathbf{S}, \forall c \in \mathbf{Ca}_s : \\ &\quad \textit{running}(s, c)_t \quad \Leftrightarrow \Big(\quad \exists x \in \mathbf{P}, \textit{instanceof}(x, s) : \\ &\qquad\qquad\qquad\qquad \textit{active}(x)_t \wedge c\big(\textit{argumentof}(x)\big)\Big)\end{aligned}$$

The generalization of the past of one service with constraints is done adding $\mathbf{Cr}_s$, the set of possible constraints we can apply to the returned values of the service s. Then, using the *last* function, we can express this service has a past which respects the constraint $(ca, cr) \in \mathbf{Ca}_s \times \mathbf{Cr}_s$ with:

$$\begin{aligned}&\forall t, \forall s \in \mathbf{S}, \forall (ca, cr) \in \mathbf{Ca}_s \times \mathbf{Cr}_s : \\ &\quad \textit{past}(s, (ca, cr)) \quad \Leftrightarrow \big(\quad \textit{last}(s)_t = (x, ret, \tau) \wedge \\ &\qquad\qquad\qquad\qquad ca(\textit{argumentof}(x)) \wedge cr(ret)\big)\end{aligned}$$

We can also extend the ordering predicate and the data flow management with constraints following the method used to extend the *past* and the *running* test.

D. System constraint rules

To specify possible conflicts and constraints of the system, we use the formal model presented above to express contexts leading to inconsistent states.

Here is an example of a simple system with two services:

- The `move` service takes a distance and a speed as arguments.
- The `camera` service is able to take images. An argument specifies if we want a high or low resolution picture.

We want to express that we cannot take any image in high resolution if the robot is moving faster than 0.5 $m.s^{-1}$. This can be expressed with this formula:

$$\begin{aligned}\forall t : \neg\big(\quad & \textit{running}(move, speed > 0.5)_t \wedge \\ & \textit{running}(camera, mode = high)_t\big)\end{aligned}$$

The role of the R^{2}C is to maintain this formula true in all the states of the system. Using the definitions of the *running* predicate we can decompose this formula into atomic predicates including the controllable ones (i.e. *rejected** and *killed**) applied to *move* and *camera* requests instances satisfying the constraints.

To maintain this formula true, the R^{2}C manages the controllable predicate values (i.e. setting them to the proper value). In this case it can reject or kill service instances of *move* or *camera* when incoming events threaten the formula consistency.

To express these constraints in a more "human legible" representation, we have developed a language and its compiler named ExoGen [16].

VI. Maintaining constraint rules in real-time

The previous section shows that the R^{2}C may be seen as a component maintaining a formula true. Still, such approach is reasonable in our context if and only if the R^{2}C deductions are fast enough to keep the synchronous hypothesis "acceptable". The approach we propose to satisfy this requirement is to use Ordered Binary Decision Diagrams (OBDDs, see [15]). This graph based data structure expresses logical formulas with the following properties:

- The resulting structure is a complete factorization of the initial formula. This implies that a predicate value is checked only one.
- The traversal is bounded (complexity is on the order of the variables number).
- We can validate it using model checking techniques.

OBDDs are used to express first order logic formulas. This is not sufficient when our model is more complex with the introduction of constraints. Thus we have defined an OBDD like data structure named OCRD[5](see [16]).

This data structure is quite similar to OBDDs but is able to express formulas with predicates with the following form:

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

The construction algorithm of one OCRD is quite similar to the OBDD one. It differs on the introduction of predicates which are similar but which have different constraints. In this case the compiler makes a partition of the constraints and split nodes accordingly. An example of such OCRD is given in Fig. 3.

The main interest of the OCRD data structure is that the resulting diagram is strictly equivalent to an OBDD. In fact it is an OBDD where each variable corresponds to one partition of the state space of the system. As a consequence OCRDs keep the properties of OBDDs (canonical form, time bounded traversal, ...) while adding some expressiveness.

VII. Experimental results

We have implemented and integrated the R^{2}C on the LAAS architecture[3] and the results are quite encouraging. The platform for our experimentation – Dala, our ATRV robot – is based on the architecture instantiation represented partially in Fig. 4. This robot embeds modules managing the following functions:

- Camera manages the stereo bench.
- SCorrel computes a stereo correlation to from the stereo bench images.
- STEO uses SCorrel outputs to deduce the robot position.
- Lane maintains a 3D map of the environment.
- P3D computes a motion plan based on a 3D representation of the environment and its goal position.
- Platine controls the camera pan/tilt.
- RFLEX manages the wheels and exports the odometry data.
- POM is a module producing the best position estimate according to positions produced by other modules and their precision.
- ...

We have extracted for this system 13 constraints rules representing all the conflicts and faulty interactions between 14 services. For example, one constraint we have specified for the POM module specifies that a position cannot be read from this module if we have not connected this one to any position producer.

The resulting OCRD has a maximum depth of 22 with a total size of 399 nodes. The R^{2}C traversing this graph has a maximum cycle time of 300μs. The overall system reactivity is kept (the global system cycle is approximatively around 100 ms). We have tried to inject many situations threatening these constraints and all of these are

[5]OCRD: Ordered Constrained Rules Diagram.

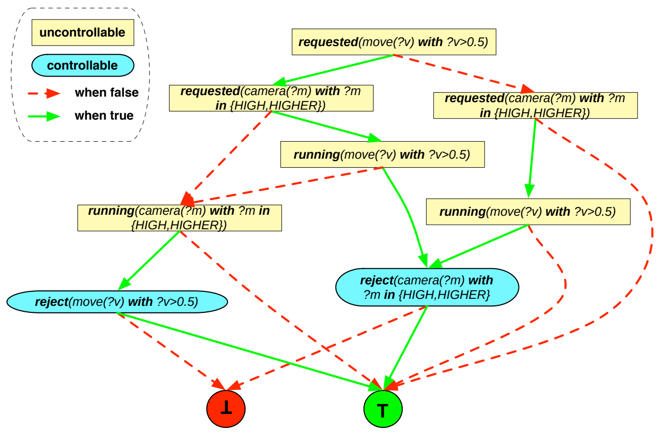
uncontrollable
controllable
when false
when true
requested(move(?v) with ?v>0.5)
requested(camera(?m) with ?m in {HIGH,HIGHER})
requested(camera(?m) with ?m in {HIGH,HIGHER})
running(move(?v) with ?v>0.5)
running(camera(?m) with ?m in {HIGH,HIGHER})
running(move(?v) with ?v>0.5)
reject(move(?v) with ?v>0.5)
reject(camera(?m) with ?m in {HIGH,HIGHER}
⊥
⊤

Fig. 3. Example of OCRD

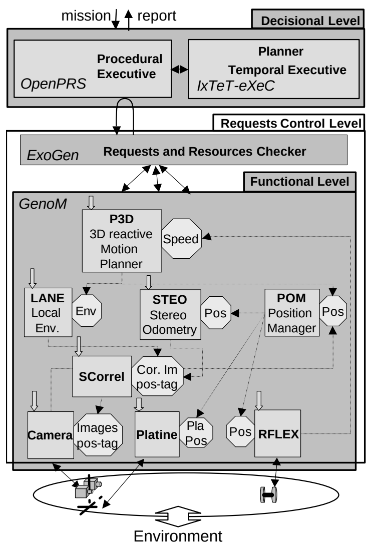

Fig. 4. An instance of the LAAS architecture

detected and treated properly (rejecting requests or killing of existing services) maintaining the system in a state that not threatens functional level consistency.

An interesting result is 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 the executive. Unfortunately, as of today, the executive has not been programmed to take into account this kind of messages. Moreover it is rather simple to consider these messages as a service failure and to react accordingly.

VIII. Conclusion and Perspectives

In this paper, we show that an autonomous robot cannot offer safety guarantees without execution control. We propose a solution based on a synchronous model supported by a specification of controllable and observable events of the system.

The R^{2}C is a quite simple, but yet powerful, component implementing this solution. It uses a general model of the functional level to supervise its state changes.

To control and avoid inconsistent states in real-time, 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 guaranty on its maximum evaluation time. As we use an approach similar to OBDDs, we expect to be able to use model checking tools to validate some more complex temporal properties of the R^{2}C.

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. Finally, 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] N. Muscettola, P. P. Nayak, B. Pell, and B. Williams, "Remote agent : To boldly go where no ai system has gone before," *Aritificial Intelligence*, vol. 103, 1998.

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

[3] 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.

[4] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty Architecture for Robotic Autonomy.," in *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky Montana, March 2001.

[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] 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] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan, "Model-Based Programming of Fault-Aware Systems," *Artificial Intelligence*, pp. 61–75, winter 2003.

[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] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "Claraty and challenges of developing interoperable robotic software," in *International Conference on Intelligent Robots and Systems (IROS)*, Nevada, October 2003, invited paper.

[13] 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.

[14] 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.

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

[16] 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.