Online Execution Control Checking for Autonomous Systems*

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

Abstract

This paper presents some recents developments of the LAAS architecture for autonomous systems. In particular, we clarify and specify the role of the Execution Control level of our architecture. This level has a fault protection role with respect to the command issued by the decisional level, which are transmitted to the real system (through the functional level). To implement this Execution Control level, we propose an approach and a tool inspired from the model checking domain. We present a new language, used to specify the model of acceptable and required states of the system (valid contexts for requests to functional module and resources usage). The model written in this language is then compiled in an OBDD (Ordered Binary Decision Diagram) like structure which is used online to check in real-time the constraints and the rules specified. Such model checking approach, used in a synchronous context, provides critical dependable properties. Moreover, these approaches can be further used to check off line more complex temporal properties of the system.

1 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 a satellite RCS jets are not fired when the camera lens protection is off? etc. A partial response to this problem is to use a planner which will only synthesizes valid and safe plans. Still, high level planners do not (cannot) have a complete model representing the full extend of their actions. Moreover, some of these actions are refined by the supervisor/executive, therefore the particular sequence of commands sent to the physical system is not completely controlled by the planner.

A solution to guarantee this fault protection property is to integrate a system that formally controls the validity of the commands sent to the physical system and prevents it to enter in an inconsistent state. This controller must check system consistency online during system execution without affecting the system basic functionalities, such as computation time. This checking is made in a synchronous hypothesis execution context.

The LAAS[1] architecture, presented in section 2, foresaw such mechanism in its execution control level, but for various reasons, the approach and tools proposed to fill this functionality was not used. Section 3 presents the Execution Control Level roles and requirements, with a state of the art of related works. Section 4 gives an informal description of the proposed approach, the tool and the language we use for the Execution Control Level, while section 5 presents the link between our proposed approach and the Ordered Binary Decision Diagram model. We then conclude the paper and consider future works and research directions.

2 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.

*This paper has been published in The 7th International Conference on Intelligent Autonomous Systems (IAS-7), March 25-27, 2002, Marina del Rey, California, USA.

†List of authors in alphabetical order.

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

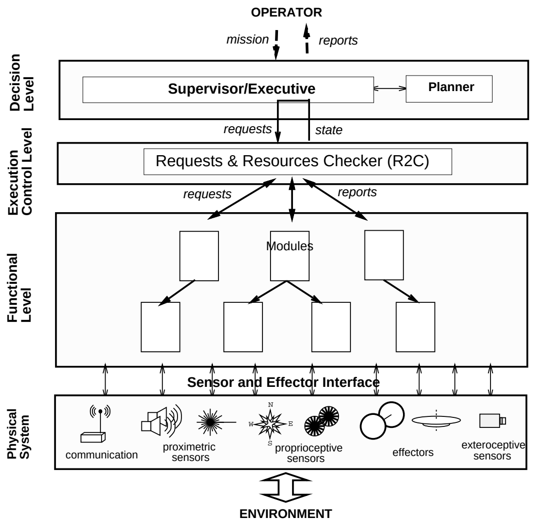

Figure 1: The LAAS Architecture.

- *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 [10], which is connected to the underlying level, to which it sends requests that ultimately will 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 application 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 [7]. 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 of the type of treatments they do. 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 G$^{en}_{o}$M 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

3 Execution Control Level

3.1 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 non formal methods and tools. Still to be able to use such advanced decisional tools, one need to design systems which in the worse cases prevents the system of engaging in 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. In one cycle, all inputs are parsed simultaneously, all outputs are produced "instantaneously" and simultaneously. This is of course an hypothesis, but it provides strong determinism to the whole checking process.
- 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 need to provide a user-friendly specification language.

3.2 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 some ways 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 for resources checking and appeared 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 [2] 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 [12] 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 an 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 [9], 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 [13] the authors present a system which allow the translation from MPL (Model-based Processing Language) and TDL (Task Description Language) to SMV a symbolic model checker language. Compare 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.

4 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 [7] and its namespace.

4.1 Overview

The R2C (see figure 2) is designed to support 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.

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.

4.2 Presentation of ExoGen

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

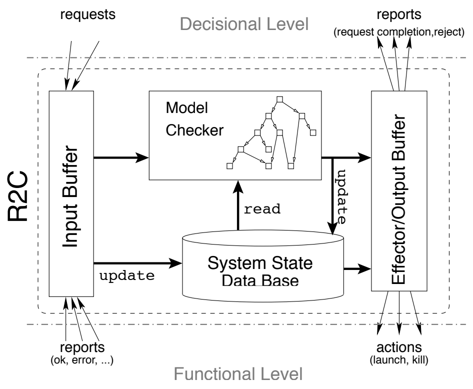

Figure 2: R2C general view.

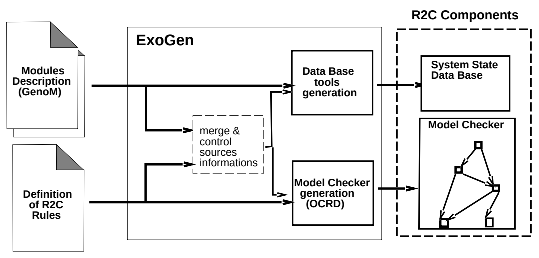

Figure 3: ExoGen development cycle.

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 will contain hundreds of such preconditions (which need to be verified or maintained), as well as their resources usage.

We shall now describe the ExoGen language features:

4.2.1 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, equality of a variable with a constant. They can be defined over the arguments and the results of the requests.

Example: Consider a robot with a camera with two modes: FIXED (the robot cannot move while taking a picture) and TRAVELING (the robot must move while taking a picture). ExoGen code for `TakeImg` request should be[2]:

```
request Cam_TakeImg() { [...]
  fail { precond:
   Last_Done(Cam_SetMode(?m) with ?m==FIXED) && Active(Mov_Move(?_)); }
  check { maintain:
   Last_Done(Cam_SetMode(?m) with ?m==TRAVELING) && Active(Mov_Move(?_)); } }
```

4.2.2 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 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 `Camera_takeImage` request we write:

```
request Cam_TakeImg() {
  uses {
    batLevel: use(10);
    batPower: use(5);}
  [...]}
```

4.3 Internal Model Structures

To allow the real-time checking of the various constraints specified by the user, the ExoGen compiler produces a structure which is a mix of those produced by KHEOPS [8] and OBDD[4] [11]. This structures corresponds to a binary decision Directed Acyclic Graph (DAG) and guarantees a maximum execution time for checking. For example given the request below :

```
request Cam_TakeImg() {[...]
  fail {
   preconds:
    Last_Done(Cam_SetMode(?m) with ?m==FIXED) && Active(Mov_Move(?_)); }
  [...]}
```

The ExoGen compiler will translate the $Active(x)$ predicate as: $\Big(askFor(x) \wedge \neg reject(x)\Big) \vee Running(x)$

The previous code is thus equivalent to the boolean formula:

$$\begin{array}{l} \Big(\Big(\texttt{askFor(Mov\_Move())} \wedge \neg\texttt{reject(Mov\_Move())}\Big) \vee \texttt{Running(Mov\_Move())}\Big) \\ \wedge \quad \texttt{LastDone(Cam\_SetMode(}?m\texttt{)}\ \textit{with}\ ?m = \mathsf{FIXED}) \\ \Rightarrow \quad \texttt{reject(Cam\_TakeImg())} \end{array}$$

The ExoGen compiler generates the OBDD shown on figure 4

In this DAG, we distinguish two types of predicates. The uncontrollable ones – `LastDone`, `askFor`, `Running` and resources predicates – corresponding to external events (external demands and current system state) and the controllable ones – `kill`, `reject` – which correspond to actions the R2C can perform. The principle of the R2C is to keep the overall formula true. To assess this goal the compiler has fixed the value of controllable predicates value to true. For instance in figure 4 if the last correctly

[2]the `Cam_TakeImg` and the `Cam_SetMode` correspond to the two requests (`TakeImg` and `SetMode`) of the `Cam` module defined with GenoM.

[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.

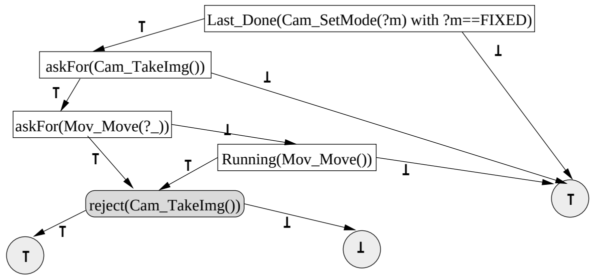

Figure 4: Example of rule DAG.

terminated instance of `Cam_SetMode` was launched with the `FIXED` argument value and the supervisor launches new instance of `Cam_TakeImg` and `Mov_Move` then R2C will do `reject(Cam_TakeImg)` to reach the true leaf.

This DAG only deduces negatives actions (i.e. `kill` and `reject`). The corresponding positives actions (respectively `launch` and `keep`) are deduced by the non-existence of the negative action. For example if system requests `rq` and R2C has not deduced `reject(rq)` then it may execute the action `launch(rq)`.

The deduction is limited to the DAG traversal so program complexity order is proportional to its depth which is fixed at compilation.

5 Formalization of the Executive

In this section we describe how ExoGen constructs the structure described in section 4.3. This data structure is based upon OBDD principles with an extension to support constrained predicates keeping all well known properties of OBDD . We call it OCRD for Ordered Constrained Rule Diagram.

5.1 Definition of the OCRD

The OCRD is a binary DAG. Leaves may be true ($\top$) or false($\bot$). Each node is composed of a predicate and its attached constraint, a branch to activate when the predicate and constraint is true and a branch to activate when it is false. An OCRD node is represented like in OBDD. Figure 5 presents the basic representation of a predicate. The basic constructors are :

$$\begin{array}{lll} \top : & & \to OCRD \\ \bot : & & \to OCRD \\ (\_,\_,\_) : & CPred \times OCRD \times OCRD & \to OCRD \end{array}$$

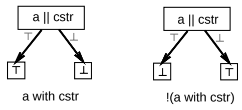

Figure 5: Representation of basics predicates.

The $CPred$ type is the association of a predicate and its constraint. Its constructor is:

$$\_ \parallel \_ : Predicate \times Constraint \to CPred$$

The reduction rule while building complete OCRD for a particular system is the same as OBDD : the DAG $(foo, whentrue, whenfalse)$ becomes $whentrue$ iff $whentrue = whenfalse$. The negation rule of an OCRD works like for OBDD (i.e. only the leafs are replaced by their negations).

The construction rules differ for binary operators. We now describe how we calculate the conjunction of two OCRD. The rules when one of the tree is a leaf($\top$ or $\bot$) are trivial. The conjunction of two OCRD with the same root node is: $(a, t_1, f_1) \wedge (a, t_2, f_2) = (a, t_1 \wedge t_2, f_1 \wedge f_2)$

The preceding rules are exactly the same that OBDD, the difference appears when two nodes differ. For definition of this operation we have to define some operators for the $CPred$ type : Where :

$$\begin{array}{lll} \_ - \_ : & CPred \times CPred & \to CPred \\ \_ \cap \_ : & CPred \times CPred & \to CPred \\ \_ \prec \_ : & CPred \times CPred & \to Boolean \end{array}$$

$$a \parallel c_a - b \parallel c_b = \begin{cases} a \parallel (c_a \wedge \overline{c_b}) & \text{if } a = b \\ a \parallel c_a & \text{else} \end{cases}$$

$$a \parallel c_a \cap b \parallel c_b = \begin{cases} a \parallel (c_a \wedge c_b) & \text{if } a = b \\ \emptyset & \text{else} \end{cases}$$

$$a \parallel c_a \prec b \parallel c_b = \begin{cases} \top & \text{if } a < b \\ \bot & \text{if } a > b \\ c_a <_{cstr} c_b & \text{if } a = b \end{cases}$$

A complete order ($<_p$) is given on predicates. The operator $<_{cstr}$ is an order for constraints. This order is partial because $c_a <_{cstr} c_b$ is not defined for the case that $\exists x / c_a(x) \wedge c_b(x)$

Now we can define the OCRD construction rules for conjunction of $A = (a, t_a, f_a)$ and $B = (b, t_b, f_b)$

$$A \wedge B = \begin{cases} (a, t_a \wedge B, f_a \wedge B) & \text{if } a \prec b \\ (b, t_b \wedge A, f_b \wedge A) & \text{if } b \prec a \\ A \curlywedge B & \text{else} \end{cases}$$

Where :

$$\begin{array}{rcl} (a, t_a, f_a) \curlywedge (b, t_b, f_b) & = & (a \cap b, t_a \wedge t_b, f_a \wedge f_b) \\ & \wedge & (a - b, t_a \wedge f_b, f_a \wedge f_b) \\ & \wedge & (b - a, f_a \wedge t_b, f_a \wedge f_b) \end{array}$$

Our goal in using OCRD as an extension of OBDD, is to keep the OBDD computational and logical properties. So this data structure may be easily used for verification of temporal properties with an appropriate model checker. The complexity of operations (like "restrict", "satisfy one",... see [3]) is exactly the same as for OBDD.

It is well known that the choice of variable ordering largely influences the OBDD size (expressed by the number of its nodes). Even if OBDD minimization is an NP-hard problem, we can find some minimization techniques [5] with a lower complexity (the best is $O(n^2.3^n)$). Furthermore we have to note that this optimization will be done during compilation time. So the duration of the minimization sequence is not critical and will give some guarantees about the generated OBDD size.

6 Experimental results

We have implemented the ExoGen tool and the resulting R2C is conected to the G$^{en}_{o}$M modules and the supervisor implemented with Propice. Preliminary tests have been conducted with "toy" problems, and we are now working on an implementation on our XR4000 Nomadics: Diligent(figure 6 and 7). Our experimentation goal is twofold:
- to test the R2C it on real experiments on board our robots and see how well it behaves, - to test it on very large problems (i.e. with numerous modules, requests and constraints to check) to see how well does it scale (size of the OCRD, traversal time compatible with hard real time, etc).

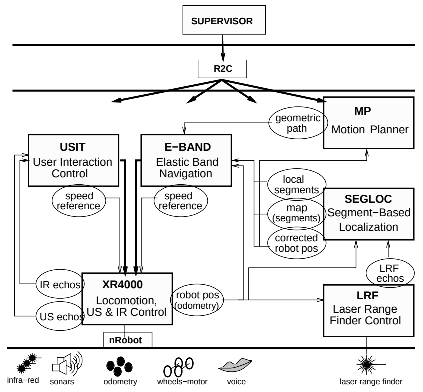

Figure 6: Diligent Architecture

Figure 7: Diligent

7 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 dependability, and in particular with respect to 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 in response to the "commands" coming from the decisional level.

Thus the proposed R2C and its associated tool ExoGen propose a language in which the user can specify, using the G$^{en}_{o}$M namespace, the contexts in which a particular request (with constraints on its arguments) can or cannot be executed.

Our approach uses OCRD which are quite similar to OBDD. Thus the generated model used in a synchronous hypothesis context provides excellent confidence in the system states reachability. Moreover, it offers some real-time properties such as the guaranty of the maximum time taken to check new incoming requests and reports. 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[5] is harder than with a complete representation.

We expect in near future to be add functionalities giving the possibility to check the modules behavior and by this way indicates the supervisor whether services are executable or not.

Our main objective in the future is to experiment formal verification 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, i.e. to represent state transitions, to allow more complex validation functionalities (reachability of a state from a particular state, unreachable states, 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*, 17(4):315–337, April 1998.

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

[3] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] Adelardo A. D. de Medeiros, Raja Chatilla, and Sara Fleury. Specification and Validation of a Control Architecture for Autonomous Mobile Robots. In *IROS*, pages 162–169. IEEE, 1996.

[5] Rolf Drechsler and Wolfgang Gunther. Using lower bounds during dynamic BDD minimization. In *Design Automation Conference*, pages 29–32, 1999.

[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*, pages 201 – 213, Munich, Germany, October 1995. Cambridge Press.

[7] 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.

[8] M. Ghallab and H. Philippe. A compiler for real-time knowledge-based systems. In *IEEE International Workshop on Artificial Intelligence for Industrial*, Hitachy City, Japan, May 1988.

[9] 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.

[10] 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.

[11] 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*, 98(2):142–170, 1992.

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

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