Toward a More Dependable Software Architecture for Autonomous Robots*

Saddek Bensalem†, Matthieu Gallien†, Félix Ingrand‡, Imen Kahloul‡, Thanh-Hung Nguyen†
†Verimag Laboratory, Grenoble I University, CNRS, France
‡LAAS/CNRS, Toulouse University, France

***Abstract*—Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. Nowadays, robots are getting closer to humans and as such are becoming critical systems which must meet safety properties including in particular logical, temporal and real-time constraints.**

We present an evolution of the LAAS Architecture for Autonomous System and its tool G^en_oM. This evolution is based on the BIP (Behaviors Interactions Priorities) component based design framework which has been successfully used in other domains (e.g. embedded systems). In this study, we show how we seamlessly integrate BIP in the preexisting methodology. We present the componentization of the functional level of a robot, the synthesis of an execution controller as well as validation techniques for checking essential safety properties. This approach has been integrated in the LAAS architecture and we have performed a number of experiment in simulation but also on a real robot (DALA).

I. INTRODUCTION

A. Toward More Dependable Robots

Autonomous robots are designed to perform high level tasks on their own , or with very limited external control. They are needed in situations where human control is either infeasible or not cost-effective. In most cases:

1) they operate in highly variable, uncertain, and time-changing environments;
2) they must meet real-time constraints to work properly; and
3) they are often interacting with other agents, both humans and other machines.

For example, service home robots will need to contend with all the complexities of sensing, planning, acting in real time in an uncertain, dynamic environment; to interact intelligently with humans and other robotic systems; and to guarantee their safety and the one of the people they encounter. Some examples, such as tour robots, or nurse robots, have demonstrated their reliability through extensive experimentations and testing, but in limited environments [1], [2]. We are far from giving formal assurances of safety that would be needed before deploying more widely such robots. In such applications, the need for guarantees of safety, reliability, and overall system correctness is acute.

*Part of this work has been funded by the ANR-SSIA AMAES project and by the FNRAE MARAE project.
A shorter version of this paper will appear in IEEE Robotics and Automation Magazine, Special Issue on Software Engineering for Robotics (2008).

The degree of assurance we can provide today is based on extensive simulation and testing. The goal of simulation is to catch errors as early as possible in the design phase to reduce the need for more costly testing of the implemented system. Both simulation and testing suffer from being incomplete: each simulation run and each test evaluates the system for a single set of operating conditions and input signals. For complex autonomous and embedded systems it is impossible to cover even a small fraction of the total operating space with simulations. Finally, testing is already too expensive; today building a test harness to simulate a component's environment is often more expensive than building the component itself.

B. Architecture and Tools

Designing and developing software for an autonomous robot is quite a challenging and complex task. One has to take into account the following context:

- there is a wide range of software "types" to integrate (from low level servo loop, to data processing, up to high level automated action planning and plan execution).
- the temporal requirements of these software components vary a lot (from hard real-time, to polynomial, and up to NP complete decisional algorithm).
- the various software components are developed by different programmers, with different backgrounds who in most case know little about the other components involved,

To address these concerns, the robotic community has relied on architectures and tools. these architecture and tools rely on a number of good software engineering practices.

- The software components are organized in levels or layers. Most of the time, these layers correspond to different temporal requirements, or to different abstraction requirements.
- The architecture and tools provide some control flow mechanisms to support requests or commands with arguments passed from one component to another, as well as reports sent back to the requester upon completion.
- Similarly, some data flow mechanisms are provided to offers access to data produced by a component to another component.

Some architectures go further and provide:

- Interoperability library to convert data from one framework to another (e.g. your low level functional components may be written in C or C++, while your high

level planner or execution controller use a symbolic representation)

- Software tools which encapsulate the components and provide a clear API of what each component provides as services, or exported data structures (e.g. GenoM).
- Software development environment to map particular services in threads, processes and even CPUs.
- Seamless integration with higher level tools for autonomy (action planner, plan execution control, FDIR, etc)

All of these properties are welcomed and have been of great value in the development of software platforms in the robotics community. This has resulted in a large number of successful architectures (LAAS [3], CLARAty [4], etc), middleware (PlayerStage [5], ORCA2 [6], etc), and tools (CARMEN [7], YARP [8], etc) etc[1]. Each of them has pros, cons, strong points and limitations. The reader will find references and some comparisons in the surveys presented in [9] (with respect to some communication performances, or memory footprint) but also in [10] (along a number of criteria grouped in four categories). Nevertheless, as of today, these architectures, tools and middleware have achieved a lot, and they have allowed the deployment of numerous successful robotic experiments. However none of them rely on any formal model which allows to synthesize controller correct by construction, nor to verify safety properties on the resulting system while these guarantee may soon be expected by certification bodies and imposed on manufacturers.

C. Desirable and Critical Properties

None of the current architecture and associated tools are able to unambiguously answer simple questions such as:

- Can you prove that your nursebot will not start full throttle while an elderly person is walking while leaning on it?
- Can you guarantee that the arm of your service robot is not going to open its gripper while holding a cup of coffee and drop it on the carpet?
- Can you prove that there is no deadlock in the initialization sequence of your robot?
- Can you prove that there is no race condition in a perception action loop?
- Can you prove that current speed of the robot is consistent with the range of your sensor and the data processing duration?
- etc

These are difficult questions, even for regular software, and a fortiori even more for autonomous robots software. But one must admit that little has been done to address them on a complete robotics system. In [11], the authors present an interesting approach based on a the Esterel synchronous language which however suffer from the limitation of the synchronous programming paradigm. Meantime, robots are becoming more and more pervasive, and the time will soon come where a certification body will require robot software developers to exhibit what is being done to address such serious security and dependability issues. It is not clear if just having good software engineering practices will be enough.

Roboticists are interested in a number of properties, and one need to "translate" these in formal statement or constraints, such as (but not limited to):

- deadlock detection (e.g. to prove that adding a particular software module on the robot will not lead to deadlock during execution),
- temporal constraints (e.g. to guarantee that any particular initialization sequence of the robot will execute action A before action B),
- timed constraints (e.g. to guarantee that a particular perception/action loop takes less than a given amount of time).
- etc.

D. Formal Methods

Formal verification is the process of determining whether a system satisfies a given property of interest. Today the best known verification methods are model checking and theorem proving, both of which have sophisticated tool support and have been used in non-trivial case studies, including the design and debugging of microprocessors, cache coherence protocols, internetworking protocols, smartcards, and air traffic collision avoidance systems (see [12] for other examples). Model checking in particular has enjoyed huge success in industry for verifying hardware designs. Formal verification can be used to provide guarantees of system correctness. It is an attractive alternative to traditional methods of testing and simulation, which for autonomous and embedded systems, as argued above, tend to be expensive, time consuming, and hopelessly inadequate. By formal verification we mean not just the traditional notion of program verification, where the correctness of code is at question. We more broadly mean design verification, where an abstract model of a system is checked for desired behavioral properties. Finding a bug in a design is more cost-effective than finding the manifestation of the design flaw in the code.

Unfortunately, after decades of research formal verification has not become part of standard engineering practice. One reason is that these techniques do not scale: code size is too large for practical program verification; the underlying mathematical formalisms (i.e., logics) do not handle all features of the programming language or all behavioral aspects of the system; and proof methods lack compositionality. Another reason is that the tools do not scale: model checkers are limited by the size of the state spaces they can handle; theorem provers require too much human time and effort for too little perceived gain; and the tools are not integrated to work with others found already in the engineer's workbench.

The software is an integral part of autonomous robot systems[2]. The shortcomings of current design, validation, and maintenance processes make software, paradoxically, the most

[1]We invite the reader to check the wiki: `http://wiki.robot-standards.org/index.php/Current_Middleware_Approaches_and_Paradigms` for a good overview and comparative analysis.

[2]Our most complex robotic experiments have source code whose size is in the order of half million to one million lines.

costly and least reliable part of the systems used in critical application. In the following we will lay out what we see as an autonomous robot software design challenge. In our opinion, this challenge raises not only a technology questions, but more importantly, it requires the building of a foundation that systematically and even-handedly integrates, from the bottom up, computation and physicality [13].

E. The Claims of this Paper

This paper presents an evolution of the LAAS architecture which integrates a state of the art component based design approach (BIP). This evolution allows us to produce a complete controller correct by construction which enforces *online* the safety properties modeled. The resulting model can also be checked *offline* with Verification & Validation tools and suites (deadlock detection, timed automata, etc). Of course, the proposed approach cannot cope with *silent* failures, i.e. situations where a sensor or an effector misbehave while the system has no mean to observe this failure.

The paper is organized as follow. Section II presents the existing LAAS architecture with an emphasis on the GenoM tool, while section III introduces the BIP componentization approach for embedded systems. Section IV explains how we merged the later in the former to obtain a model of all the generic components of any GenoM module. Section V presents a real example on the DALA robot and show how we write a BIP model for the functional modules involved in its navigation activity. We then explain how we produce a controller correct by construction (section V-B) and what are the properties we are able to show on the model (section V-C). We conclude the paper with the prospective, and future research avenues we intend to explore.

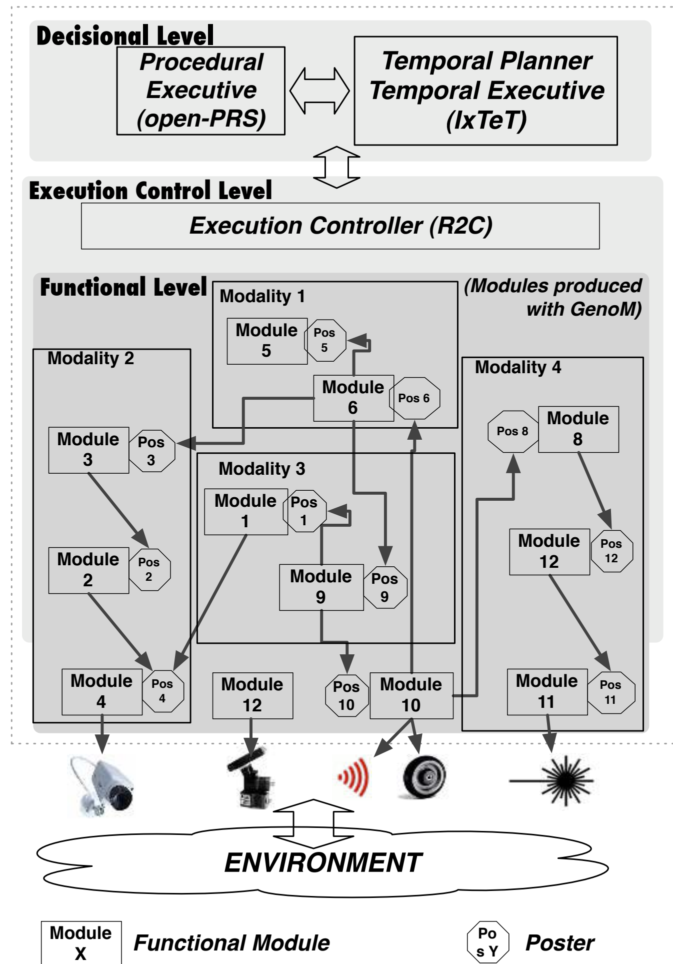

Fig. 1. The LAAS Architecture.

II. An Existing Architecture...

At LAAS, researchers have developed a framework, a global architecture, that enables the integration of processes with different temporal properties and different representations. As presented on fig. 1, this architecture decomposes the robot software into three main levels, having different temporal constraints and manipulating different data representations [14]. This architecture is used on all our robots at LAAS (e.g. DALA, an iRobot ATRV; HRP2; Rackham, an iRobot B21; Jido, etc) and in other institutes. The levels in this architecture are :

- a *functional* level: it includes all the basic built-in robot action and perception capacities. These processing functions and control loops (e.g., image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating modules developed using GenoM[3]. Each module provides *services* which can be activated by the decisional level according to the current tasks, and exports *posters* containing data produced by the module and for others (modules or the decisional level) to use.
- a *decisional level*: this level includes the capacities of producing the task plan (using the IxTeT planner) and supervising its execution (with OpenPRS), while being at the same time reactive to events from the functional level.
- At the interface between the decisional and the functional levels, lies an *execution control level* that controls the proper execution of the services according to safety constraints and rules, and prevents functional modules from unforeseen interactions leading to catastrophic outcomes. In recent years, we have used the R2C [15] to play this role, yet it was programmed on the top of existing functional modules, and controlling their services execution and interactions, but not the internal execution of the modules themselves. One of the goal of this study is to embed such models in the controller correct by construction produced with BIP.

This study focuses, for now, on the functional level and on the execution control level. Our approach heavily relies on the existing functional module generation tool: GenoM but also completely replaces the R2C.

[3]The GenoM tool as well as other tools from the LAAS architecture can be freely downloaded from: `http://softs.laas.fr/openrobots/wiki/genom`

A. GenoM Functional Modules

The LAAS/GenoM methodology is based on the encapsulation of each basic functionality of the robot in a module. For example the basic sensors and effectors are managed by their own module (e.g. one module for the camera pair, one module for the laser range finder, etc). More complex functionalities are encapsulated in higher level module (e.g. a module doing stereo correlation will use the image taken by the camera module, a module building an obstacle map will use the LRF scan, etc). Last, some complex modalities, such as navigation, are obtained by combining a number of modules.

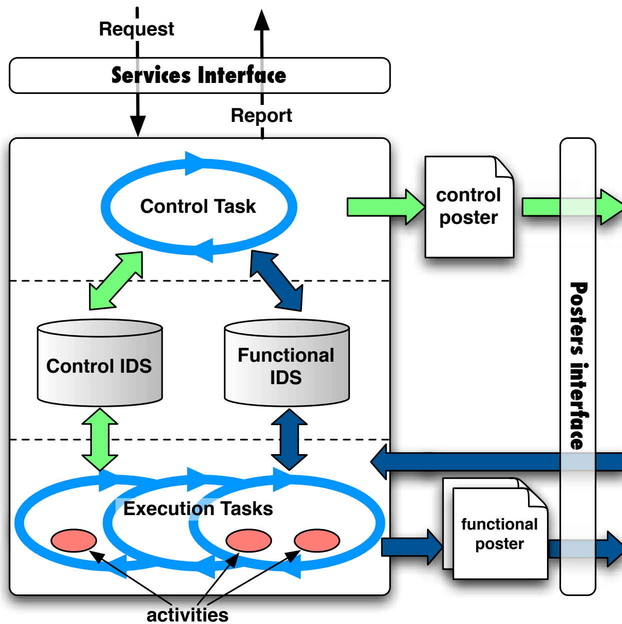

Fig. 2. A GenoM module organization.

All these modules are built by instantiating a unique generic canvas (Fig. 2). Each module is specified by providing the following information: the *internal functional data structure (IfDS)*, the list of *services* (started with requests) provided by this module, the list of *posters* (if any) exported by this module and the list of *execution tasks* with their respective activation period.

- The *internal functional data structure (IfDS)* defines the various "public" C-structures used by the module . These can be used to specify *posters*, arguments and reports to *services*, etc.
- The *services* correspond to the "commands" the module will accept. There are two types of service: the control ones only modify the IfDS and will not be executed by an activity and the execution ones will be executed by an activity. Arguments can be passed and reports (status and values) can be sent back upon completion. For each execution service, one has to specify the various pieces of C code (codels) which have to be executed, and in which *execution task* the *activity* of the *service* will run.
- *Posters* are data structures which are produced by the module and can be read by other modules.
- *Execution tasks* are cyclic tasks (threads in most implementations) which execute the *activities* corresponding to the active *services*.

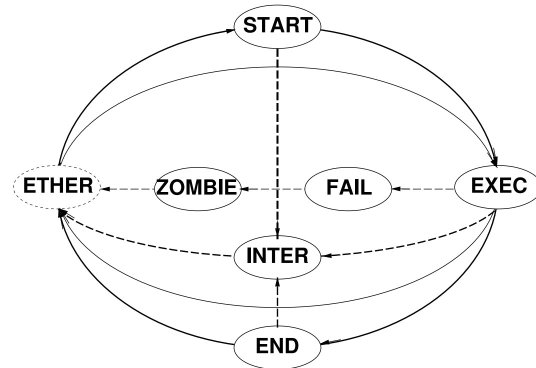

Fig. 3. Execution automaton of an activity. The codels of GenoM are called in each state except *ETHER* and *ZOMBIE*.

The *services* are initially managed by a *control task* which is responsible for launching the corresponding *activities* within the appropriate *execution tasks* and for executing control services. *Control and execution tasks* share data using the internal data structures (IDS). Fig. 3 presents the automata of an *activity* as executed by all the launched *services*. Activity states correspond to the execution of particular codels available through libraries and dedicated either to initialize some parameters (START state), to execute the activity (EXEC state) or to safely end the activity leading to reseting parameters, sending error signals, etc. According to the value returned by the codels, the automata make the proper transition E.g. if the EXEC codel return EXEC, then it stays in this state, and the same codels will be called the next time the activity is run but if it returns OK, it goes in the END state.

The organization of the LAAS architecture in layers and of the functional level in modules are definitely a plus with respect to the ease of integration and reusability. Our goal is not to redesign a new architecture and develop a new set of tools from scratch. Most of the assets offered by the existing setup should be kept. Thus our approach is merely to build on the existing solution.

III. ... AND A COMPONENTIZATION APPROACH FOR EMBEDDED SYSTEMS...

A. Component Based Design

Component-based design is essential to any engineering discipline when complexity dictates methodologies that leverage reuse and correct-by-construction approaches. A central idea in systems engineering, such as robot software, is that complex systems are built by assembling components (building blocks) [16], [17]. This is essential for the development of large-scale, evolvable systems in a timely and affordable manner. Component-based design confers many advantages with respect to monolithic design, such as reuse of solutions, modular analysis and validation, reconfigurability and controllability. Components are systems characterized

by an abstraction that is adequate for composition and reuse, provided via an interface. An interface specifies how a component is viewed by its potential users. Composition and its properties are essential for mastering the component construction process. Component-based design relies on a separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows their global treatment and analysis.

One of the main limitations of the current state-of-the-art is the lack of unified frameworks for describing and analyzing the coordination between components. This is particularly true for robotic systems where the coordination is usually enforced by a high level model, but not from a clean bottom up approach. Such frameworks would allow system designers and implementers to formulate their solutions in terms of tangible, well-founded and organized concepts, instead of using dispersed low-level coordination mechanisms including semaphores, monitors, message passing, remote call, protocols etc. Unified frameworks should allow a comparison and evaluation of otherwise unrelated architectural solutions, as well as derivation of implementations in terms of specific coordination mechanisms. The component-based design problem can be formulated as follows: "build a system meeting a given set of requirements from a given set of components that are known to satisfy another set of requirements." This is an essential problem in any engineering discipline. It lies at the basis of various system-design activities, including modeling, architecting, programming, synthesis, upgrading, and reuse.

Component-based design has been used in hardware. During the past decade, IT developers and end-users have benefited from the commoditization of commercial-off-the-shelf (COTS) hardware (such as CPUs and storage devices) and networking elements (such as IP routers). For VLSI circuit design, component-based design methodologies, supported by CAD tools, have been in use for System-on-Chip products albeit much remains to be done to achieve the level of maturity needed to make this approach a standard in the industry.

An important trend in modern systems engineering is model-based design, which relies on the use of explicit models to describe development activities and their products. It aims at bridging the gap between application software and its implementation by allowing predictability and guidance through analysis of global models of the system under development. The first model-based approaches, such as those based on ADA, synchronous languages [18] and Matlab/Simulink, support very specific notions of components and composition. More recently, modeling languages, such as UML [19] and AADL [20], attempt to be more generic. They support notions of components which are independent from a particular programming language, and put emphasis on system architecture as a means to organize computation, communication, and implementation constraints. Software and system component-based techniques have not yet achieved a satisfactory level of maturity. Systems built by assembling together independently developed and delivered components, often exhibit pathological behavior. Part of the problem is that developers of these systems do not have a precise way of expressing the behavior of components at their interfaces, where inconsistencies may occur. Components may be developed at different times and by different developers with, possibly, different uses in mind. Their different internal assumptions, further exposed by concurrent execution, can give rise to emergent behavior when these components are used in concert, e.g. race conditions, and deadlocks. All these difficulties and weaknesses are amplified in embedded system design in general. They cannot be overcome, unless we solve the hard fundamental problems raised by the definition of rigorous frameworks for component-based design.

B. BIP

BIP is a software framework for modeling heterogeneous real-time components. The BIP component model is the superposition of three layers: the lower layer describes the *behavior* of a component as a set of *transitions* (*i.e* a finite state automaton extended with data); the intermediate layer includes *connectors* describing the *interactions* between transitions of the layer underneath; the upper layer consists of a set of *priority* rules used to describe scheduling policies for interactions. Such a layering offers a clear separation between component behavior and structure of a system (interactions and priorities).

BIP allows hierarchical construction of *compound* components from *atomic* ones by using connectors and priorities.

An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of transitions and a set of local variables. Transitions describe the behavior of the component. They are represented as a labeled relation between *control states*.

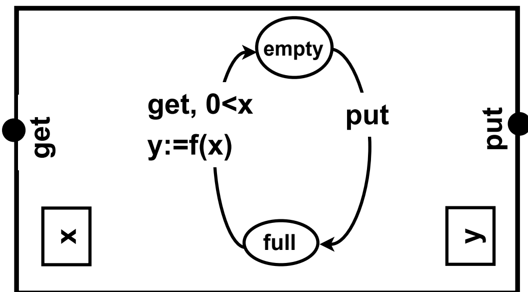

Fig. 4. An example of an atomic component in BIP.

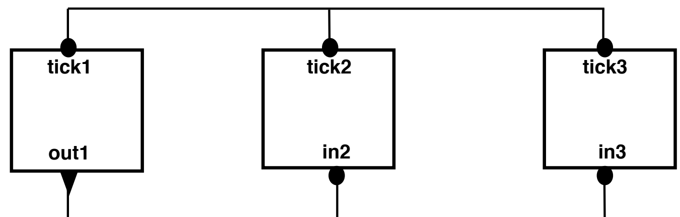

Fig. 5. Possible interactions in BIP (triangle for complete port, and circle for incomplete port).

Fig. 4 shows an example of an atomic component with two ports get, put, variables x, y, and control states $empty$, $full$.

At control state $full$, the transition labeled get is possible if the guard $0 < x$ is true. When an interaction through get takes place, the variable x is possibly modified through the interaction and a new value for y is computed. From control state $empty$, the transition labeled put can occur.

Connectors specify the interactions between the atomic components. A connector consists of a set of ports of the atomic components which may interact. An interaction of a connector is any non empty subset of its set of ports. A typing mechanism is used for the ports in order to determine the feasible interactions of a connector and in particular to model the two basic modes of synchronization. As shown on Fig. 5: *rendezvous* when all the ports are incomplete (tick1 and tick2 and tick3) and *broadcast* when at least one port is complete (out1, out1 and in2, out1 and in3).

Priorities In composite components, many interactions can be enabled at the same time, introducing a degree of non-determinism in the product behavior. Non-determinism can be restricted by means of priorities, specifying which of the interactions should be preferred among enabled ones.

The model of a system is represented as a BIP compound component which defines new components from existing components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities.

The BIP toolset [21] is a collection of tools dedicated to execution and analysis of BIP programs currently providing:

• *A compilation chain that transforms BIP programs into C/C++ code.* Compilation relies on model-based technologies available for Java under the Eclipse platform. Starting from BIP programs, the compiler generates BIP models conforming to a full-fledged BIP meta-model developed using EMF[4]. On the models, we can apply source-to-source transformations as well as static analysis techniques. Finally, models are used to generate C/C++ code to be executed on a dedicated platform, as follows.

• *A platform for execution and analysis of the generated C/C++ code.* The execution platform includes an Engine and the associated software infrastructure for multithreaded execution of the C/C++ code. Each atomic component is assigned to a thread, the Engine being a thread itself. Communication takes place only between the atomic components and the Engine, and never directly between different atomic components – this leads to a centralized architecture. The Engine implements the distributed semantics [22] and is parameterized by a dynamic[5] or lazy[6] oracle. Iteratively, the Engine computes feasible interactions available on ready components. Then, if such interactions exist and the oracle allows them, the Engine selects one for execution and notifies the involved components.

IV. ... MERGED IN AN UNIFIED FRAMEWORK

In section II we have described the "previous" LAAS architecture, while in section III we introduces the BIP framework. The main idea of this work is to retain the modular and leveled organization of the former while merging the later in a new framework. Indeed, if we model the GenoM generic module and its components in BIP and if we then instantiate it and connect the existing "codels" to the resulting component, then we obtain a BIP model of all the GenoM modules. Adding the BIP model of the interaction between the modules (which were encapsulated in the R2C in the previous LAAS architecture) will give us a BIP model of the overall functional layer *and* of the execution control layer. Such a BIP model is then used to synthesize a controller for the overall execution of all the functional modules and to enforce by construction the constraints and the rules inside modules but also between the various functional modules.

In this section we show how we map the GenoM modules and components in BIP. The GenoM generic module organization (Fig. 2) can easily be mapped in a hierarchy of BIP components (Fig. 6): execution tasks, activities, etc. For example, the *service* components are further composed with (at least one if not more) *Execution Task* and *poster* components to obtain a *module* component.

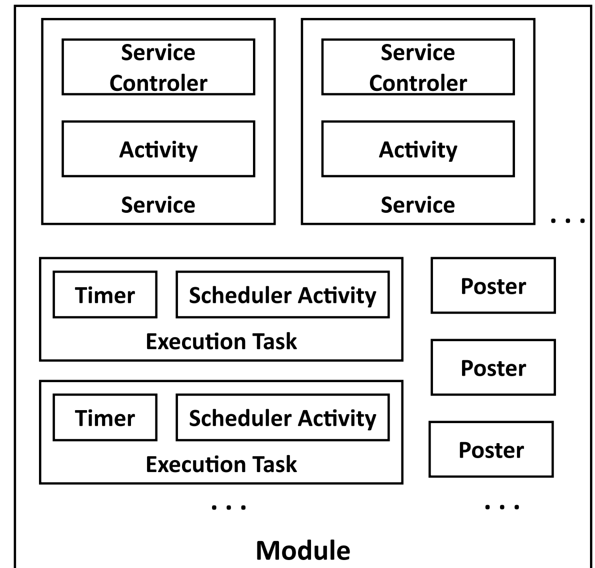

Fig. 6. The componentization of a GenoM module.

Overall, we propose the following mapping:

Functional level ::= (Module)+
Module ::= (Service)+ . (Execution Task)+ . (Poster)+
Service ::= (Service Controller) . (Activity)
Execution Task ::= (Timer) . (Scheduler Activity)

where "+" means the presence of one or more of sub-component and"." means the composition of different components.

A component modeling the generic *service* (as presented in section II-A and Fig. 3) is obtained from the atomic components *service controller* and *activity* and the connectors between them, as shown on Fig. 7.

The left sub-component represents the controller of a service. It is launched by synchronization through port *trigger*.

[4]Eclipse Meta-modeling Framework.

[5]For the dynamic oracle, the Engine does not need a complete knowledge of the state of the system in order to compute a dynamic approximation for a given partial state.

[6]The lazy oracle forbids all interactions from partial states. It waits for all the atomic components to finish their computation in order to know all the possible interactions.

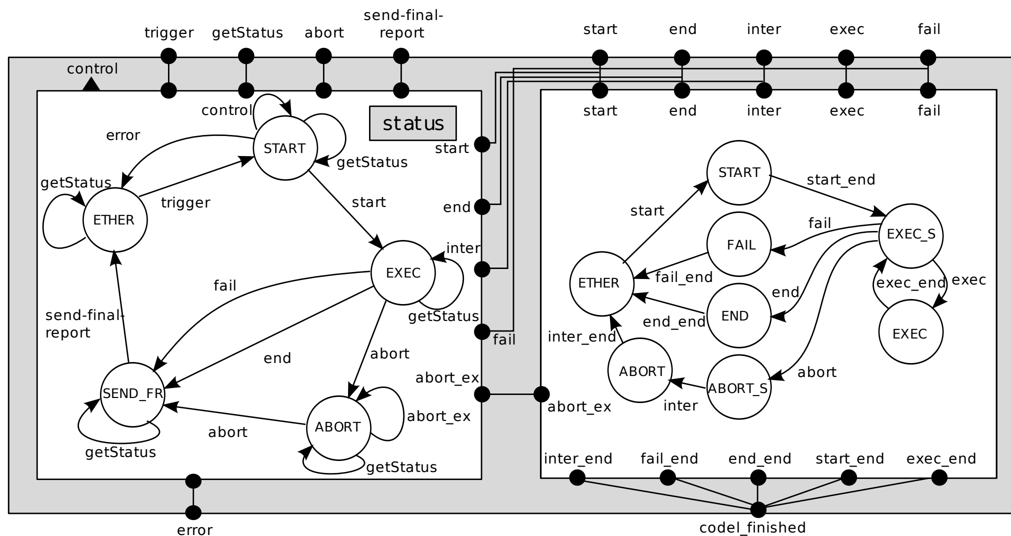

Fig. 7. The BIP generic model of an execution GenoM service.

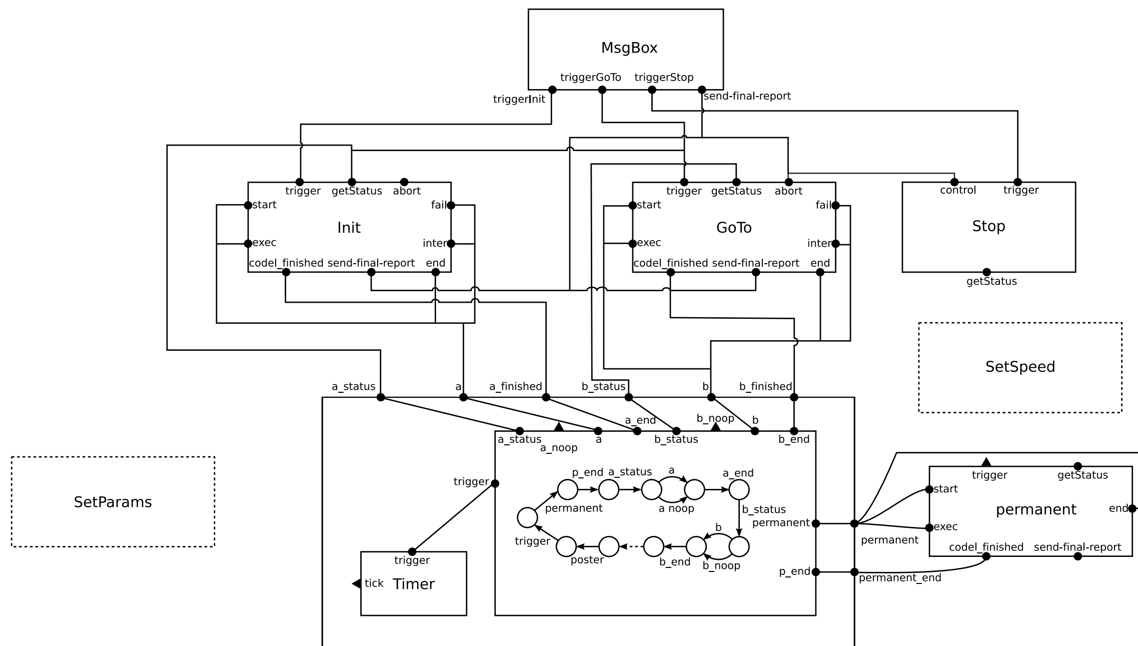

Fig. 8. The partial BIP model corresponding to the NDD module. All services are not included in the figure in order to make it more readable.

The Service Controller *controls* the validity of the parameters (if any) of the request and will either *reject* it or *start* the activity by synchronizing with the activity component (right sub-component). In each state, the status of the service is available by synchronizing through port *status* of the controller component. The activity will then wait for execution (i.e. synchronization on the *exec* port with the control task) and will either safely *end*, *fail*, or *abort*. Each of the transitions *control*, *start*, *exec*, *fail*, *end* and *inter* may call an external function corresponding to GenoM codels.

To ease the integration of BIP in the new framework, we developed a tool that produces a BIP model from a GenoM module description file. Moreover, keep in mind one does not have to rewrite codels as we reuse the existing ones by a simple call in the corresponding atomic component behavior. Thus, for already existing GenoM modules, the BIP model can be obtained without writing a line of code. So there are no "overhead" for the module developper in using BIP. Still, if one want to enforce some safety properties inside a module (intra-module) or between modules (inter-modules), these constraints have to be explicitly added to the resulting BIP model. One has to identify in the safety properties the components involved, then add connectors between component ports and the appropriate guards (if needed) so has to have the BIP engine enforcing these properties.

So from the point of view of the GenoM module developper, there is absolutely no overhead in using BIP. The only added work he will have correspond to the *new* safety properties he

may want to add to the BIP model for them to be enforced by the resulting controller. For the moment, these properties have to be directly written in BIP, however, we are working on a tool and a language to express them in a form which is more appealing to roboticists.

V. An Illustrated Example and Resulting Properties

The approach presented has been implemented and tested on a real robot as well as its simulation.

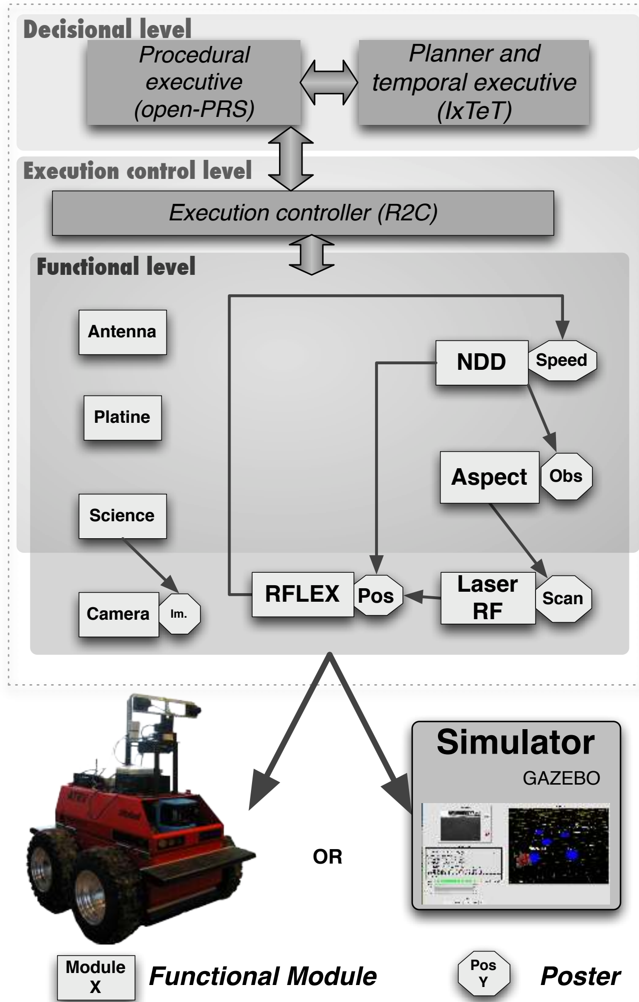

Fig. 9. An instance of the LAAS architecture for the DALA Robot on which we applied the BIP approach.

A. The DALA Robot: previous approach

To illustrate the overall approach, we implemented our new approach on the DALA robot, an iRobot ATRV (see Fig. 9) running a simple, yet complete, functional layer with a laser based navigation using the Near Diagram navigation. The G^en_oM/BIP approach has been used to model the following modules as well as their interactions. We focus our first efforts on these four modules, because together they close the perception action loop, and yet are sufficient to have the robot moving around controlled with our new approach.

- RFLEX: This is the module that ensures the robot locomotion. After proper initialization, upon receiving the `RflexTrackSpeedStart` (with a poster name as argument) it servo-controls (at a period of 4 ticks[7]) the wheels speed contained in the poster given as argument (in our case this poster is produced by NDD). It also maintain the current robot odometry position in a poster `(pos)`.
- Laser RF: This module manages the laser range finder sensor. After proper initialization with the service `Init`, it cyclically (every 20 ticks) produces the position of the closest obstacles with respect to the robot position. These data are stored in the `(scan)` poster.
- Aspect: It uses the `(scan)` poster to periodically (every 4 ticks) produce/update a poster `(Obs)` of the obstacles map in the robot vicinity.
- NDD: This module is responsible for the navigation of the robot, i.e. reach a goal while avoiding obstacles. After proper initialization, it cyclically (every 10 ticks) recovers the current position (in the RFLEX `(pos)` poster) and the obstacles in the Aspect `(Obs)` poster, and it produces the poster `(Speed)` which will be used by RFLEX to speed servo-control the wheels.

This experiment uses other modules (antenna, platine, camera and science) whose role is not critical in the scope of this paper. Moreover, the full BIP description of the four functional modules presented above is also beyond the scope of this paper. We rather focus on the modeling of the NDD module (which is the most complex one) to illustrate the general idea.

The NDD module contains six *services*, one *poster* and has one *execution task* as sub-components and the connectors between them, as shown in Fig. 8 (see Fig. 10 for an example of BIP code of the NDD module). Each service is an instance of the service model presented on Fig. 7. Similarly, the poster and the execution task are instance of their respective BIP model.

The *control task* wakes up periodically (managed by the bottom-left component with alternating sleep and trigger transitions) and always triggers the *Permanent* service[8] at the beginning of each period. For each cycle, the various services (`GoTo`, `SetParams`, etc) will have the opportunity to execute the interaction if possible.

Moreover, the BIP formalism allows complex relations to be defined; such as triggering the Stop control service stops the robot by interacting with the "abort" port of the `GoTo` service. The connector enforcing this interaction is a broadcast between the execution task and the port `Stop.trigger`[9] and `GoTo.abort`. Another property which must always be verified is that a `GoTo` should only occur if it is preceded by the proper initialization with a successful execution

[7]1 tick is equal to 10 ms.
[8]The *Permanent* service executes a codel associated to the execution task in G^en_oM.
[9]Notation used: service.port

of SetParams and SetSpeed services. We model this safety property by a connector between GoTo.trigger, SetParams.getStatus and SetSpeed.getStatus.

These are example of allowed interaction inside the NDD module. However, we can also model interaction between modules. For example to model that NDD can only execute a GoTo (which produced the (Speed) poster) if Aspect has produced a map (Obs), or that RFLEX can only start moving the robot if a proper (Speed) reference has been produced by NDD, etc. All these "operational" constraints will be introduced in the BIP model and the resulting controller will enforce them by construction.

```
component nddGoToExecActivity
  port incomplete start, finished, fail, inter
  port incomplete exec, abort_ex
  port incomplete check_posters_to_update
  port incomplete codel_is_executed, send_final_report
  data {#
    ACTIVITY_EVENT event;
  #}
  extern data ACTIVITY_EVENT event, ETHER, EXEC, END, FAIL
  data int manualPosterNeedUpdate
  data int report
  behavior
    initial
    do manualPosterNeedUpdate = false;
      to Q_ETHER
    state Q_ETHER
      on start do
        {#
          manualPosterNeedUpdate = 0;
          //call the start codel
          event = nddGoToStart(&nddGoToParams,
                               &report,
                               &manualPosterNeedUpdate);
        #}
      to Q_START_RUNNING
    state Q_SLEEP
      on finished provided (event == END) do
[...]
component nddGoToService
  contains nddGoToExecControl NddGoToControl
  contains nddGoToExecActivity NddGoToActivity
  connector connNddGoToStart = NddGoToControl.start,
                               NddGoToActivity.start
    behavior
    end
[...]
```

Fig. 10. Example of BIP model/code for the NDD module.

B. Functional Level Controller Synthesis

Previously, in the LAAS architecture, a centralized controller (R2C) was used to control the proper execution of the services and to enforce the safety constraints on the modules interactions. On the contrary, in the BIP model, the proper execution order and the safety properties are enforced by the BIP connectors between the controllers of different services. A BIP connector has guarded actions associated to each of its possible interactions. Dependency between the controllers of service in different modules are modeled by connectors associated with guards which represents either some valid execution condition or some safety rule. The composite behavior of these local controllers, synchronized by the connectors and restricted by priorities, is equivalent to the behavior of the centralized controller.

As an example, we had to enforce a rule between the NDD and the RFLEX modules which states that the robot can move using the TrackSpeedStart service of the RFLEX module only if the module NDD has already executed successfully its GoTo service (which updates the poster (Speed)). Such a rule is enforced by constructing a connector between port trigger of the TrackSpeedStart service and port status of the GoTo service, and guarded by the status value.

With respect to the generation of the real controller, the BIP tool-chain generates code from the BIP model of generic components, and from the G^{en}oM module definition. The resulting model has for respectively Laser RF, Aspect, NDD and RFLEX; 88, 77, 116 and 100 States; 60, 60, 89 and 89 connectors; and 27, 22, 29 and 35 components with at most a depth of 4 levels. This model is then linked to the corresponding G^{en}oM codels. With G^{en}oM, these codels were triggered in the corresponding service/automata states, similarly, with BIP the codels are now executed upon the transition of the corresponding component automata.

The code generated for the four modules NDD, RFLEX, Aspect, and Laser RF has been integrated and executed on the real robot.The performance analysis shows that when running the BIP generated controller the CPU load is on average globally 2.5 higher (with peaks at 3.0) than when one runs the pure G^{en}oM generated code. There are mainly two reasons for this overhead. The first one is that the componentized code has not yet been optimized (compare to G^{en}oM code). The second reason is that there is a price to pay for the BIP engine to check all the interactions before executing them. This is done in one main program which starts a thread for every states transition (in component) which executes some code. During the experiment, the engine fires approximately 550 interactions per seconds. So even if the checking and firing of an interaction is really small, the penalty comes from the large number of interactions fired.

C. Verified Safety Properties

While the constraints imposed by the software architecture of autonomous system facilitate development, they are also the potential source of undesired or unexpected task interactions. In particular, there are special classes of decisional and real-time bugs that frequently arise in the use of such architectures. In this work, we addressed three classes of problems:

1) Ordering violations: arise at the behavioral level when several behaviors recommend conflicting actions. In autonomous systems, there is typically an explicit arbitration mechanism that chooses among the different behaviors. In this case, bugs arise when the priority mechanism leads to the wrong choice of action for a given set of input conditions. Typically, this is because the developer has made some implicit assumption about the external, or internal, state of the system at the behavior is triggered. By making these assumptions explicit, and by reasoning about the interactions of the various components (using the D-Finder tool [23]) we are able to check if the model allows the system to reach a state where orders have been violated or where conflicts arise. D-Finder is

an interactive tool that takes as input BIP programs and applies proof strategies to verify deadlock freedom and other safety properties by computing increasingly stronger invariants.

For example, in the NDD module, it is required that the GoTo service should be executed only after a successful termination of Init, SetParams and SetSpeed services. Using D-Finder, we have been able to show that this property is not preserved. To guarantee this property, we added in the NDD module model a connector to restrict the behavior. This means that the interactions with the trigger port of the GoTo service is possible only if the other services have been successfully executed (i.e. by interacting with the getStatus port of those services). We checked again the modified model and we found that there were still possibilities to violate the property if you restart the Init service after the beginning of SetParams and Init execution fails before the end of SetParams. Finally, to obtain the desired property, we added a new constraint to guarantee the mutual exclusion between the two services.

One can see that some ordering or conflict avoiding constraints can be explicitly added to a model, from the beginning (in which case, they will be enforced by the BIP generated controller) or, they can be checked against an existing model and explicitly added if needed. Thus the development of a "safe" model requires to extensively use a tool such as D-Finder to check each model for deadlock freedom and for satisfaction of the ordering properties.

2) Synchronization violations: typically appear when tasks are mis-synchronized. Excess synchronization can lead to deadlock. Lack of synchronization can lead to resource conflict. To verify whether the BIP model of the functional level satisfies a synchronization-related property such absence of deadlocks, we also use the D-Finder tool (it takes up to three hours to check for deadlock freedom for the NDD module). In our example. the result of one such analysis lead to several potential deadlocks. There were due to a fault made during the modeling of the generic MsgBox component. This component is responsible for the communications between the decisional level and the services provided by the BIP modules. It will regularly check if a new request for a service has been received and then will trigger it. If the service is already running, it is not possible to trigger it and the MsgBox component will be blocked until the execution of the previous instance is finished. This was blocking its associated timer which was blocking all other timers and lead to a system deadlock. The obvious solution was to not block, if you cannot do a trigger, but instead refuses to start the service. We modified/fixed the model and there was no more deadlocks in the NDD BIP model.

Note that in order to check this property we have to make hypothesis on the behavior of the codels of the NDD module. These codels return the next state of the automata for the corresponding service (see Fig. 3 or Fig. 7). For example if we are in the START_S state and the value returned is EXEC, then the activity automaton has to go in the EXEC_S state; but if we are in the ABORT_S state, this value is not a priori possible and will provoque a deadlock. Thus we are making the "reasonnable" hypothesises, that codels returns values "consistant" with the state in whichthey where launched. Hence, avoiding dedlocks which are impossible to reach.

3) Data freshness: As already mentioned above, the communication and the data transfers between the modules of the robot are achieved using posters (shared memory in most implementation). Each poster contains data needed by the module which is reading it. The very near future behavior of each module depends very strongly on the content of the posters it is reading.

The data freshness property consists in reading posters which are "up-to-date". If the module is still keeping an old value of the poster, it is using, then this may cause it to make wrong decisions which may be critical in some cases. For instance, consider the situation where the robot is in front of an obstacle while it is moving. The obstacle shall be detected by the Laser RF module. The latter writes this information in its poster (Scan) which is read by the Aspect module. In turn, Aspect writes the obstacle map in its (Obs) poster which is read by NDD. If all these operations and the corresponding processing do not happen fast enough then the robot may keep moving and collide the obstacle.

To avoid such a situation, we want a poster reader to refuse to take into account data which are too old. Thus, if the service find outdated data in the poster, it will stop by itself.

To illustrate this we consider the read of the poster (Obs) of the Aspect module by the permanent service of NDD. There are two possibilities. First, we may force the property by adding an adequate guard on the connector used to model the read. In some way, we synthesize a (timed) controller which guarantees the property to be satisfied. The second possibility consists in considering the periodic behavior of each module and to check whether the freshness property is satisfied for the considered periods of the two modules or not. In this case, the property is not explicitly forced but is checked offline with D-Finder.

In both cases (on-line and off-line), one can use the so-called *observers* to monitor the property. An observer is a BIP component which encodes the considered property. This BIP component has a particular state called ERROR. This state is reached as soon as the property is violated. To be able to detect if the property is violated or not we compose the observer component with the existing model.

VI. Conclusion and Perspectives

Programming autonomous robots is still in the ad-hoc phase, and suffers from the lack of paradigm and model which encompass the full autonomous robot software design challenge. Current robotic software suffers from limitations that are introduced by many manual steps, such as system integration, which proceed mostly by "trial and error" (i.e. test and tweak). Current models are inadequate, because they address only isolated aspects of autonomous robot systems, while their interactions are not always well understood. Meanwhile, as the need to make autonomous robots more dependable and safer rises, so does the requirements on the dependability of the overall software which "drives" these systems.

We propose a mathematical basis for autonomous robot systems modeling and analysis which integrates both abstract-machine models and transfer-function models in order to deal with computation and physical constraints in a consistent, operative manner. The theory, the methodologies, and the tools encompass heterogeneous execution and interaction mechanisms for the components of a system, and they provide abstractions that isolate the subproblems in design that require human creativity from those that can be automated.

We present an approach integrating component-based construction and validation of robotic systems. We show that a complex robotic system can be considered as the composition of a small set of atomic components. Although we build up on the pre-existing modular LAAS architecture for autonomous robots, and model in BIP all the generic components of this architecture, such an approach could be used with other robot software architectures and tools.

The approach has been fully implemented and we now have a G$^{en}{}_{o}$M/BIP controller for the navigation part of a functional layer of DALA (an iRobot ATRV), running in simulation and on the real robot. This controller enforces online by construction the interactions model (intra-module and inter-module). Our first runs on the robot show that the BIP engine performance are good enough for a simple yet complete robotics experiment. At this stage, the controller is multi-threaded but not multi-CPU. Current research at Verimag is being conducted to address this limitation with an interface model of the host hardware.

The paper shows that it is possible to use structural analysis techniques for deadlock detection and for verification of safety properties. Another possibility is the online monitoring of the functional level execution using observer components, which would be able to generate feedback actions for the decisional level which can be useful for error-recovery. Another work direction is to extend the BIP model to take into account the decisional capabilities of autonomous robots. To this effect, we could model part or all of the decisional layer and components. For example we already had a study which uses UPPAAL tiga as a planning system [24]. A similar model could be made in BIP and we could apply some reachability analysis to perform planning.

REFERENCES

[1] M. Montemerlo, J. Pineau, N. Roy, S. Thrun, and V. Verma, "Experiences with a mobile robotic guide for the elderly," in *Proceedings of the AAAI National Conference on Artificial Intelligence*. Edmonton, Canada: AAAI, 2002.

[2] A. Clodic, S. Fleury, R. Alami, R. Chatila, G. Bailly, L. Brthes, M. Cottret, P. Dans, X. Dollat, F. Elise, I. Ferran, M. Herrb, G. Infantes, C. Lemaire, P. Lerasle, J. Manhes, P. Marcoul, P. Menezes, and V. Montreuil, "Rackham: An interactive Robot-Guide," in *IEEE International Symposium on Robot and Human Interactive Communication (demonstration session) (RO-MAN), University of Hertforshire, Hatfield, UK, 06/09/06-08/09/06*. http://www.ieee.org/: IEEE, 2006, pp. 502–509.

[3] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *IJRR*, 1998.

[4] I. 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, Oct. 2003, invited paper.

[5] B. Gerkey, R. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the Int. Conf. on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, 2003.

[6] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *IROS'06 workshop on Robotic Standardization*, Beijing, China, 2006.

[7] M. Montemerlo, N. Roy, and S.Thrun, "Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit," in *Proceedings of the Int. Conf. on Intelligent Robots and Systems (IROS 2003) workshop on Robotic Standardization*, Pittsburgh, PA, USA, 2003.

[8] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: Yet another robot platform," *International Journal of Advanced Robotics Systems, special issue on Software Development and Integration in Robotics*, vol. 3, no. 1, 2006.

[9] A. Shakhimardanov and E. Prassler, "Comparative evaluation of robotic software integration systems: A case study," *Intelligent Robots and Systems*, Jan 2007.

[10] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, 2007.

[11] 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: Cambridge Press, October 1995, pp. 201 – 213.

[12] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, 1996.

[13] T. Henzinger and J. Sifakis, "The discipline of embedded systems design," *IEEE Computer*, vol. 40, no. 10, pp. 36–44, 2007.

[14] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *IJRR, Special Issue on Integrated Architectures for Robot Control and Programming*, vol. 17, no. 4, 1998.

[15] F. Py and F. Ingrand, "Dependable execution control for autonomous robots," in *IROS*, Sendai, Japan, 2004.

[16] J. Sifakis, "A framework for component-based construction extended abstract," in *SEFM*, 2005, pp. 293–300.

[17] T. Henzinger and J. Sifakis, "The embedded systems design challenge," in *FM: Formal Methods*, ser. Lecture Notes in Computer Science 4085. Springer, 2006, pp. 1–15.

[18] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[19] J. Ivar, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison Wesley Longman, 1998, no. ISBN 0-201-57169-2.

[20] P. H. Feiler, B. A. Lewis, and S. Vestal, "The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems," in *IEEE International Symposium on Computer-Aided Control Systems Design*, 2006, pp. 1206 – 1211.

[21] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *SEFM*, Pune, India, 2006.

[22] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis, "Distributed semantics and implementation for systems with interaction and priority," in *Proceedings of FORTE'08*, ser. LNCS. Springer, June 2008.

[23] S. Bensalem, M. Bozga, J. Sifakis, and T.-H. Nguyen, "Compositional verification for component-based systems and application," in *6th International Symposium on Automated Technology for Verification and Analysis*, Seoul, South Korea, October 2008.

[24] Y. Abdeddam, E. Asarin, M. Gallien, F. Ingrand, C. Lesire, and M. Sighireanu, "A comparison between CBTP and TGA approaches," in *ICAPS*, Providence, RI, USA, 2007.