

Incremental Component-Based Construction and Verification of a Robotic System

Ananda Basu[†], Matthieu Gallien^{*}, Charles Lesire^{*}, Thanh-Hung Nguyen[†],
Saddek Bensalem[†], Félix Ingrand^{*} and Joseph Sifakis[†]

^{*} LAAS/CNRS, University of Toulouse
Toulouse, France
Email: firstname.lastname@laas.fr

[†]VERIMAG CNRS/University Joseph Fourier
Grenoble, France
Email: firstname.lastname@verimag.fr

Abstract—Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. Nowadays, robots are critical systems and must meet safety properties including in particular temporal and real-time constraints. We present a methodology for modeling and analyzing a robotic system using the BIP component framework integrated with an existing framework and architecture, the LAAS[†] based on G^{en}M. The BIP componentization approach has been successfully used in other domains. In this study, we show how it can be seamlessly integrated 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.

I. INTRODUCTION

A central idea in systems engineering is that complex systems are built by assembling components (building blocks). Components are systems characterized by an abstraction that is adequate for composition and re-use. It is possible to obtain large components by composing simpler ones. Component-based design confers many advantages such as reuse of solutions, modular analysis and validation, reconfigurability, controllability, etc.

Autonomous robots are complex systems that require the interaction/cooperation of numerous heterogeneous software components. They are critical systems as they must meet safety properties including in particular, temporal and real-time constraints.

Component-based design relies on the separation between coordination and computation. Systems are built from units processing sequential code insulated from concurrent execution issues. The isolation of coordination mechanisms allows a global treatment and analysis.

One of the main limitations of the current state-of-the-art is the lack of a unified paradigm for describing and analyzing the information flow between components. Such a paradigm would allow system designers and implementers to formulate their solutions in terms of tangible, well-founded and organized concepts instead of using dispersed coordination

mechanisms such as semaphores, monitors, message passing, remote call, protocols, etc. It would allow in particular, a comparison of otherwise unrelated architectural solutions and could be a basis for evaluating them and deriving implementations in terms of specific coordination mechanisms.

The designers of complex systems such as autonomous robots need scalable analysis techniques to guaranteeing essential properties such as the one mentioned above. To cope with complexity, these techniques are applied to component-based descriptions of the system. Global properties are enforced by construction or can be inferred from component properties. Furthermore, componentized descriptions provide a basis for reconfiguration and evolutivity.

We present an incremental componentization methodology and technique for an already existing autonomous robot software developed at LAAS. The methodology considers that the global system architecture can be obtained as the hierarchical composition of larger components from a small set of classes of *atomic* components. Atomic components are units processing sequential code that offer interactions through their interface. The technique is based on the use of the *Behavior-Interaction-Priority* (BIP) [2] component framework which encompasses incremental composition of heterogeneous real-time components.

The main contributions of the paper include:

- A methodology for componentizing and architecting autonomous robot systems.
- Composition techniques for organizing and enforcing complex event-based interaction using the BIP framework.
- Validation techniques for checking essential properties, including scalable compositional techniques relying on the analysis of the interactions between components.

The paper is structured as follows. In Section II we illustrate with a real example, the preexisting architecture (based on G^{en}M [6]) of an autonomous robotic software developed at LAAS. From this architecture, we identify the atomic components used for the componentization of the robot software in BIP. Section III provides a succinct description of the BIP

[†]LAAS Architecture for Autonomous System.

component framework. Section IV presents a methodology for building the BIP model of existing $G^{en}M$ functional modules and their integration with the rest of the software. Controller synthesis results as well as “safety” properties analysis are also presented. Section V concludes the paper with a state of the art, an analysis of the current results and future work directions.

II. MODULAR ARCHITECTURE FOR AUTONOMOUS SYSTEMS

Autonomous robots are complex systems that require the interaction/cooperation of numerous and quite different software modules/components. This is usually achieved with the proper architecture, methods and tools.

A. Preexisting Software Architecture

At LAAS, we have developed a framework, a global architecture, that enables the integration of processes with different temporal properties and different representations. This architecture decomposes the robot system into three main levels, having different temporal constraints and manipulating different data representations [1]. This architecture is used on a number of robot, in particular DALA, an iRobot ATRV, and is shown on Fig. 1. The levels in this architecture are :

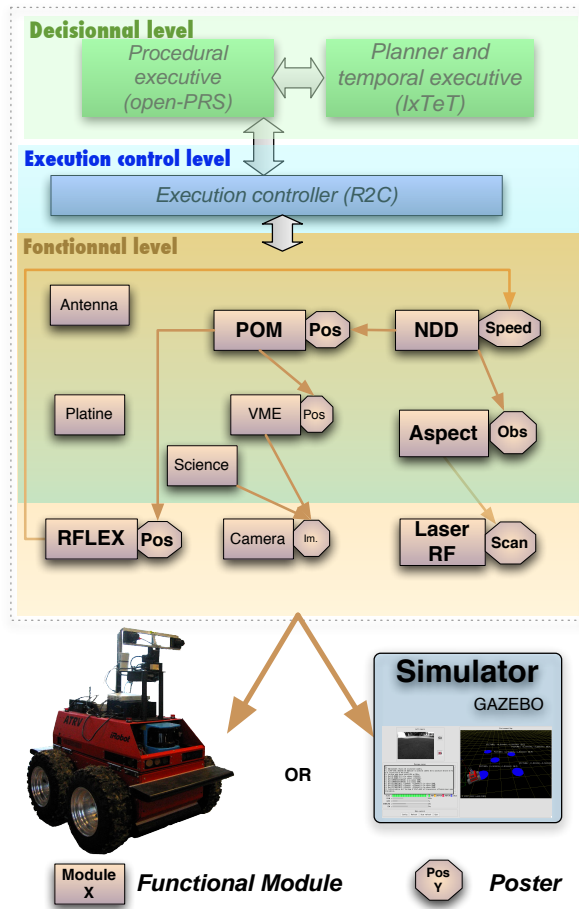


Fig. 1. An instance of the LAAS architecture for the DALA Robot.

- 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 $G^{en}M^2$. Each modules provide *services* which can be activated by the decisional level according to the current tasks, and *posters* containing data produced by the module and for other (modules or the decisional level) to use.
- a *decisional* level: this level includes the capacities of producing the task plan and supervising its execution, while being at the same time reactive to events from the functional level. The coexistence of these two features, a time-consuming planning process, and a time-bounded reactive execution process poses the key problem of their interaction and their integration to balance deliberation and reaction at the decisional 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.

The organization of the overall system in layers and the functional level in modules are definitely a plus with respect to the ease of integration and reusability. Yet, an architecture and some tools are not “enough” to warrant a sound and safe behavior of the overall system.

In this paper the componentization method we propose will allow us to synthesize a controller for the overall execution of all the functional modules (which will be componentized) and will enforce by construction the constraints and the rules between the various functional modules. Hence, the ultimate goal of this work is to implement both the current functional and execution control level with BIP.

B. Componentization of $G^{en}M$ Functional Modules

Each module of the LAAS architecture functional level is responsible for a function of the robot. Complex modalities (such as navigation) can be obtained by having modules “working” together. For example in Fig. 1 (which only shows the data flow of the functional level), there is an implicit processing loop. The module Laser RF acquires the laser range finder and store them in poster Scan, from which Aspect builds the obstacle map Obs. The module NDD (responsible for the navigation) avoids this obstacle while producing a Speed reference to reach a given target from the current position Pos produced by POM. Finally, this Speed reference is used by RFLEX, which controls the speed of the

²The $G^{en}M$ tool can be downloaded from: <http://softs.laas.fr/openrobots/wiki/genom>

robots wheels, and also produces the odometry position to be used by POM to generate the current position.³

All these modules are built using a unique generic canvas (Fig. 2) which is then instantiated for a particular robot function. We shall now describe this generic module, taking NDD as an example of instance, and identifying the “components” which are typeset in *italic*.

Each *module* can execute several *services* started on client requests. The module can send information relative to the executed requests to the client (such as the final report) or share data with other modules using *posters*. E.g. the NDD *module* provides six *services* corresponding to initializations of the navigation algorithm (definition of parameters), launching and stopping the path computation toward a given goal and a permanent service (*SetParams*, *SetDataSource*, *SetSpeed*, *Stop*, *GoTo* and *Permanent*). NDD also exports a poster (*Speed*) containing the speed reference.

The *services* are managed by a *control task* responsible for launching corresponding *activities* within *execution tasks*.

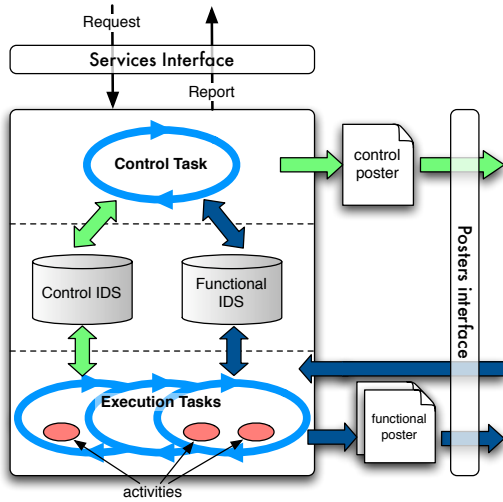


Fig. 2. A $G^{nb}M$ module organization.

Control and execution tasks share data using the internal data structures (IDS). Moreover execution tasks have periods in which the several associated *activities* are scheduled. It is not necessary to have fixed length periods if some services are aperiodic. Fig. 3 presents the behavior of an *activity*, inspired from classical thread life cycle. Activity states correspond to the execution of particular elementary code 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 resetting parameters, sending error signals, etc.

The component-based approach considers the functional level as a hierarchical entity which is decomposed successively into simpler components like modules, and further down to atomic components like execution tasks, activities, etc.

³This particular setup will serve as an example throughout the rest of the paper.

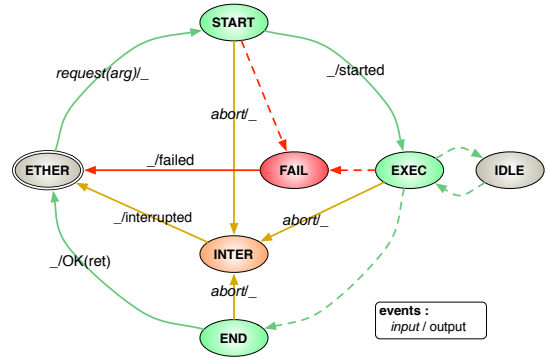


Fig. 3. Execution automaton of an activity.

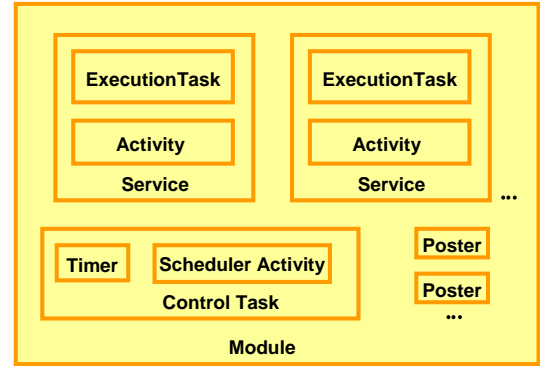


Fig. 4. A componentized $G^{nb}M$ module.

In order to formalize the componentization approach, we propose the following decomposition:

$$\begin{aligned}
 \text{Functional level} &::= (\text{Module})^+ \\
 \text{Module} &::= (\text{Service})^+ \cdot (\text{Control Task}) \cdot (\text{Poster})^+ \\
 \text{Service} &::= (\text{Execution Task}) \cdot (\text{Activity}) \\
 \text{Control Task} &::= (\text{Timer}) \cdot (\text{Scheduler Activity})
 \end{aligned}$$

where “+” means the presence of one or more of the particular component and “.” means the composition of different components. The componentized view of a $G^{nb}M$ module is shown in Fig. 4.

The next section introduces the BIP framework which has been used for the componentized implementation of the functional level of the robot.

III. THE BIP COMPONENT FRAMEWORK

BIP⁴ [2] 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

⁴The BIP tool-set can be downloaded from: <http://www-verimag.imag.fr/~async/BIP/bip.html>.

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 describe the behavior of the component. They are represented as a labeled relation between *control states*.

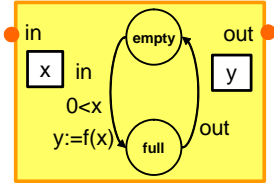


Fig. 5. An example of an atomic component in BIP.

Fig. 5 shows an example of an atomic component with two ports *in*, *out*, variables *x*, *y*, and control states *empty*, *full*. At control state *empty*, the transition labeled *in* is possible if $0 < x$. When an interaction through *in* takes place, the variable *x* is eventually modified and a new value for *y* is computed. From control state *full*, the transition labeled *out* 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, *rendezvous* and *broadcast*.

Priorities in BIP are a set of rules used to filter interactions amongst the feasible 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 framework consists of a language and a toolset including a front-end for editing and parsing BIP programs and a dedicated platform for the model validation. The platform consists of an Engine and software infrastructure for executing simulation traces of models. It also allows state space exploration and provides access to model-checking tools like *Evaluator* [12]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability.

The back-end, which is the BIP engine, has been entirely implemented in C++ on Linux to allow a smooth integration of components with behavior expressed using plain C/C++ code.

The following section describes the modeling and verification of the functional layer of the robot in the BIP framework and its integration within the LAAS framework.

IV. MODELING, VERIFYING AND INTEGRATING THE FUNCTIONAL LAYER OF THE ROBOT DALA IN BIP

In modeling the functional layer of the robot in BIP, we have used the hierarchical decomposition of the functional layer as presented earlier in section II-B. Compound components are created by composing sub-components (atoms or compounds) using the connectors between them and priorities (if required), to build the hierarchy of the complete system.

For example, a compound component modeling a generic *service* is obtained from the atomic components *execution task* and *activity* and the connectors between them, as shown in Fig. 6.

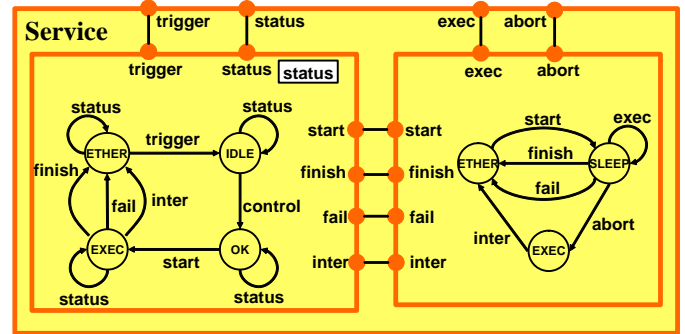


Fig. 6. BIP model of a service.

The left sub-component represents the execution task of a service. It is launched by synchronization through port *trigger*. The execution task then checks the validity of the parameters of the request (if available) and will either *reject* the request or *start* the activity by synchronizing with the activity component (right sub-component). In each state, the status of the execution task is available by synchronizing through port *status*. The activity will then wait for execution (i.e. synchronization on the *exec* port with the control task) and will either safely *finish*, *fail*, or *abort*. Each of the transitions *control*, *start*, *exec*, *fail*, *finish* and *inter* may call an external function.

The *service* components are further composed with *control task* and *poster* components to obtain the *module* components.

The full BIP description of the functional level of the robot, which consists of several modules, is beyond the scope of this paper. We rather focus on the modeling of the NDD module.

A. Modeling the NDD module in BIP

The NDD module contains six *services*, a *poster* and a *control task* as sub-components and the connectors between them, as shown in Fig. 7.

The *control task* wakes up periodically (managed by the bottom-left component with alternating sleep and trigger transitions) and always triggers the *Permanent* service at the beginning of each period. During a period, the services will have authorization to execute through interactions with the control task.

Moreover, the BIP formalism allows complex relations to be defined, such as:

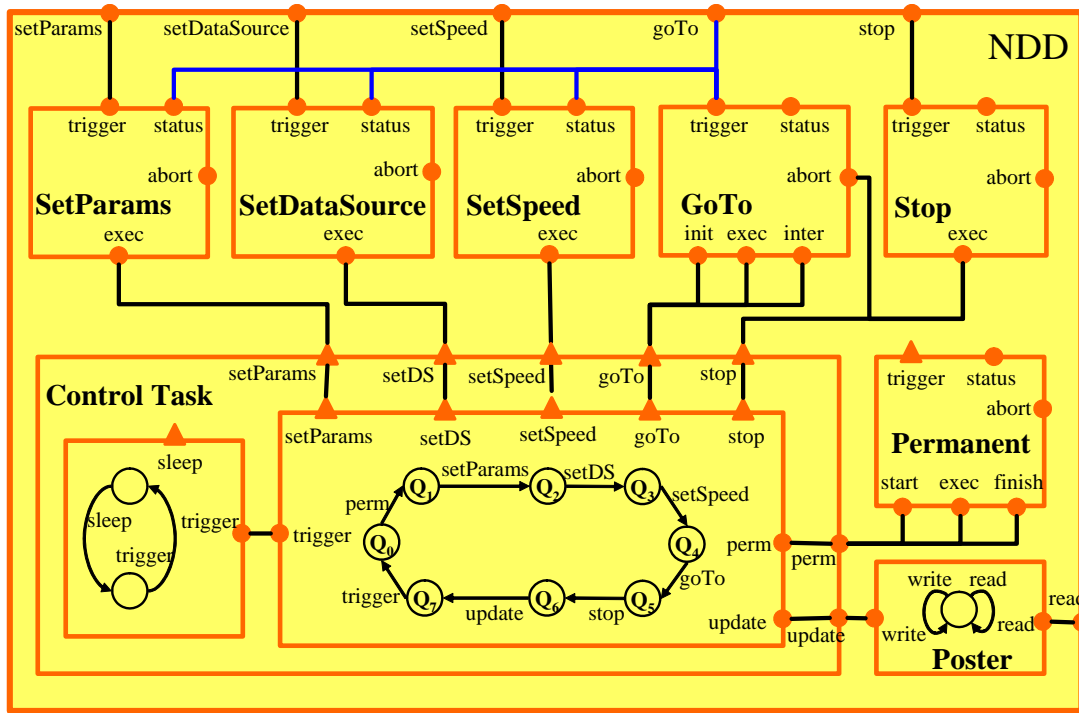


Fig. 7. The NDD module.

- interruptions, as modeled by the connector joining Stop.exec and GoTo.abort; if service Stop is executed, the GoTo algorithm will be aborted;
- constraints, as modeled by the goTo connector (in blue); service GoTo can be launched only if SetParams, SetSpeed and SetDataSource have been already completed (information available through their status port).

B. Executing and Monitoring the BIP model

The BIP tool-chain generates code from the BIP model, which can be executed by the BIP engine. The code contains calls to functions from libraries originally designed for $G^{en}M$ modules, which executes the real activities of the robotic system. The code generated for the NDD module has been integrated and executed in the robot simulation environment of LAAS [9]. In particular, it was fully integrated with the decisional layer by replacing the functional layer originally modeled with $G^{en}M$ with the one modeled in BIP.

There is a services interface in the architecture allowing each $G^{en}M$ module to have its requests called from the decisional level. In the BIP model, each module (e.g. NDD, RFLEX, ...) has an additional component to accept requests and will synchronize with the corresponding ports of the module (for example those of NDD shown in Fig. 7). Hence, the requests sent by the decisional level are received by the component and reports are sent by each service upon completion according to the protocol used by $G^{en}M$ modules. Indeed, the decisional layer does not need any modification to work with BIP.

The following section demonstrates how the methodology

enforces by construction the constraints and the rules between the various functional modules.

C. Functional level Controller synthesis

In the LAAS architecture, a centralized controller (R2C) is used to control the proper execution of the services and to enforce the safety constraints and modules interactions. On the contrary, in the BIP model, we have used separate controllers for each service. 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 POM modules which states that the robot can navigate using the GoTo service of the NDD module only if the module POM has already executed successfully its Run service (which updates poster Pos). The rule is enforced by constructing a connector between port *trigger* of the Goto service and port *status* of the Run service, and guarded by the *status* value. The *status* value of the Run service is updated when Run has been successfully executed.

The next section presents in detail the methods used for the verification of the robotic system and their results.

D. Verification of Safety Properties

The BIP tool-set can perform an exhaustive state-space exploration of the system. Additionally, it can detect potential deadlocks in the system. These features have been used to verify some properties in the model of the robot and for detection of deadlocks. Two kinds of properties have been verified.

1) *Safety properties*: A safety property guarantees that something unexpected will never happen. For the verification of such properties, we used methods based on state-space exploration. The basic idea is to generate all reachable states and state changes of the system under consideration, and represent this as a directed graph called the *state-space*. Two different methods have been applied.

a) *Model checking* [16, 3]: We used the model-checker tool *Evaluator* [12] which performs on-the-fly verification of temporal properties on the state-space generated by the BIP engine on exploration of the system. As an example, we describe the usage of this method in verifying a safety property of the NDD module. It is required that the *GoTo* service is triggered only after a successful termination of *SetSpeed* service. To ensure this, in the BIP model of NDD, we need to guarantee that the interaction *GoTo:trigger* occurs only after the occurrence of the interaction *SetSpeed:finish*. We checked for violations of this property, i.e finding a transition sequence in the state-space where *GoTo:trigger* is not preceded by *SetSpeed:finish*. This safety property can be expressed in branching-time μ -calculus [11] as the following:

$$\mu X.(\langle \text{SetSpeed:finish} \rangle T \text{ or } \langle \neg \text{GoTo:trigger} \rangle X)$$

The result obtained by *Evaluator* proves that the initialization property is preserved in the NDD module.

b) *Verification using Observers* [18, 14]: For a given system S and a safety property P , we construct first an observer for P , i.e. an automaton which monitors the behavior of S and reports an error on violation of P . The verification consists of exploring the state-space of the product system. Such a method has been used to verify a timing property in the NDD module. It is needed to verify that the total time taken by all the services called within a period does not exceeds the period.

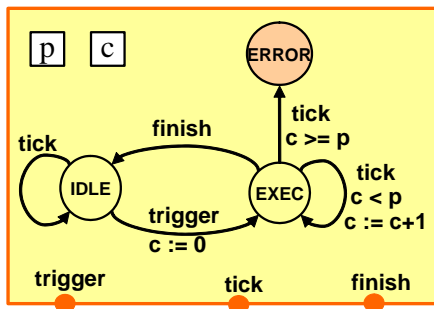


Fig. 8. Observer for the control task period verification.

In BIP, it is possible to model time as symbolic time [2] by using *tick* ports and clock variables in every timed component.

Time progress is by strong synchronization of all the *tick* ports. The clock variables are incremented on a *tick*, to model function execution times. Fig. 8 shows the observer component used to verify the timing property of the NDD module. It has a clock variable c and a parameter p representing the period of the *control task*. It synchronizes with the *control task* and tracks the cumulative time taken by the services triggered by *control task*. If this time exceeds the period p , the observer moves to the *ERROR* state. During exploration, if a global system state, containing the *ERROR* state of the observer is reachable, then the property is violated.

Such a method can also be used to verify timing properties between several modules. The processing loop presented in section II-B manages obstacle avoidance: obstacles detected by the laser are added to the aspect map which is used by NDD to compute a speed reference for RFLEX to control the robot velocity. The following time constraint can be verified: it should take less than a given time (e.g., a second depending on the current robot velocity) between the detection of an obstacle (data written in the Laser poster) and the speed reduction (the execution of the RFLEX permanent service).

2) *Deadlock freedom*: This is an essential correctness property as it characterizes a system's ability to perform some activity over its life time. The BIP toolset allow detection of potential deadlocks by static analysis of the connectors in the BIP model [7]. It generates a dependency graph and for each cycle in this graph, a boolean formula is generated. The satisfiability of the formula is then checked by the tool *minisat* [4], where a solution corresponds to a potentially deadlocked global state. Presence of an actual deadlock can then be verified by reachability analysis of the deadlocked states, starting from the initial state of the system. The analysis for the NDD module found a potential deadlock for the state where all services are in the *EXEC* state, all activities are in the *ETHER* state, and the control task is in the Q_0 state. However, this state is unreachable, hence the deadlock is not possible.

V. STATE OF THE ART, CURRENT RESULTS AND PROSPECTIVE

The design and development of autonomous robots and systems is a very active research field. There are other architectures addressing similar problems: to provide an efficient, reusable and formally sound organization of robot software. CLARAty [13], used on various NASA research rovers, provides a nice object oriented hierarchical organization over two layers, but there is no formal model of the component interactions, nor modules canvas. IDEA [5], developed at NASA Ames, has an interesting modular/component organization with a temporal constraint based formalism. However, complexity of constraint propagation is an obstacle for effective deployment on real-time functional modules. RMPL [10, 19] and its associated tools, propose a system based on a model-based approach. The programmers specify state evolution

with invariants expressed in an “Esterel like” language and a controller maintaining them.

In [8], the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description and then deduce the corresponding timed automata to control the system on-line. These automata can be formally validated with model checking techniques. However, this work focuses on the decisional part of the overall architecture. In [17] the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language) to SMV, a symbolic model checker language. Compared to our approach, this does not address componentization and seems to be more designed for the high level specification of the decisional level.

The paper presents an approach integrating component-based construction and validation of robotic systems. It shows that a complex robotic system can be considered as the composition of a small set of atomic components. The use of BIP is instrumental for achieving this, as it provides powerful constructs for coordinating components. The combination of connectors to describe interactions between components, and priorities to enforce scheduling policies, proves to be essential for incremental modeling. The global model is obtained by progressively composing its atomic components. It is possible to identify in the global model all its atomic components and their interactions. This allows in particular, to study the impact of changes of a component’s behavior or structure on the global behavior and its properties. The paper shows that it is possible to combine standard verification techniques, based on global state exploration, with structural analysis techniques for deadlock detection. This is a very interesting work direction that will be further investigated.

Another useful work direction 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 or restarting of services.

Our work is based on the idea that componentization and reasoned construction supported by an appropriate methodological framework and tools, are instrumental for coping with system design complexity. It is possible to enforce by construction some design requirements and avoid as much as possible an a posteriori validation of the global system. We will work towards achieving such a challenging goal in several directions. These include the formalization of the componentization methodology in particular regarding the interplay between atomic components and their coordination mechanisms. Another important work direction is achieving constructivity, that is guaranteeing some properties by construction or by lightweight global analysis. Finally, the methods and tools should be improved in particular to support incremental system construction and analysis.

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), (1998).
- [2] A. Basu, M. Bozga, and J. Sifakis, ‘Modeling heterogeneous real-time components in BIP’, in *International Conference on Software Engineering and Formal Methods (SEFM)*, Pune, India, (2006).
- [3] E. M. Clarke and E. A. Emerson, ‘Synthesis of synchronization skeletons for branching time temporal logic’, in *Workshop on Logic of Programs*, Yorktown Heights, NY, USA, (1981).
- [4] N. Eén and N. Sörensen, ‘An extensible SAT–solver’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Portofino, Italy, (2003).
- [5] A. Finzi, F. Ingrand, and N. Muscettola, ‘Robot action planning and execution control’, in *International Workshop on Planning and Scheduling for Space (IWSPSS)*, Darmstadt, Germany, (2004).
- [6] S. Fleury, M. Herrb, and R. Chatila, ‘G^{ep}bM: A tool for the specification and the implementation of operating modules in a distributed robot architecture’, in *International Conference on Intelligent Robots and Systems (IROS)*, Grenoble, France, (1997).
- [7] G. Goessler and J. Sifakis, ‘Component-based construction of deadlock-free systems’, in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Bombay, India, (2003). Invited talk.
- [8] R. P. Goldman and D. J. Musliner, ‘Using model checking to plan hard real-time controllers’, in *AIPS Workshop on Model-Theoretic Approaches to Planning*, Breckenridge, CO, USA, (2000).
- [9] S. Joyeux, A. Lampe, R. Alami, and S. Lacroix, ‘Simulation in the LAAS architecture’, in *ICRA Workshop on Interoperable and Reusable Systems in Robotics*, Barcelona, Spain, (2005).
- [10] P. Kim, B. C. Williams, and M. Abramson, ‘Executing reactive, model-based programs through graph-based temporal planning’, in *International Joint Conference on Artificial Intelligence (IJCAI)*, Seattle, WA, USA, (2001).
- [11] D. Kozen, ‘Results on the propositional μ -calculus’, *Theoretical Computer Science*, 27, (1983).
- [12] R. Mateescu and M. Sighireanu, ‘Efficient on-the-fly model-checking for regular alternation-free mu-calculus’, Technical Report 3899, INRIA Rhône-Alpes, France, (2000).
- [13] 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)*, Las Vegas, NV, USA, (2003). Invited paper.
- [14] M. Phalippou, ‘Executable testers’, in *International Workshop on Protocol Test Systems (IWPTS)*, Tokyo, Japan, (1994).
- [15] F. Py and F. Ingrand, ‘Dependable execution control for autonomous robots’, in *International Conference on Intelligent Robots and Systems (IROS)*, Sendai, Japan, (2004).
- [16] J-P. Queille and J. Sifakis, ‘Specification and verification of concurrent systems’, in *International Symposium on Programming*, Torino, Italy, (1982).
- [17] R. Simmons, C. Pecheur, and G. Srinivasan, ‘Towards automatic verification of autonomous systems’, in *International conference on Intelligent Robots & Systems (IROS)*, Takamatsu, Japan, (2000).
- [18] J. Tretmans, ‘A formal approach to conformance testing’, in *International Workshop on Protocol Test Systems (IWPTS)*, Tokyo, Japan, (1994).
- [19] 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*, 24(4), (2003).