Incremental Component-Based Construction and Verification of a Robotic System

Ananda Basu[1] and **Matthieu Gallien**[2] and **Charles Lesire**[2] and **Thanh-Hung Nguyen**[1] and **Saddek Bensalem**[1] and **Félix Ingrand**[2] and **Joseph Sifakis**[1]

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 Architecture for Autonomous System, based on GenoM. 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.

1 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 which seamlessly integrate with the already existing LAAS architecture for autonomous robot. 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 applied to the existing LAAS architecture.
- 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 2 we illustrate with a real example, the preexisting architecture (based on GenoM [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 3 provides a succinct description of the BIP component framework. Section 4 presents a methodology for building the BIP model of existing GenoM functional modules and their integration with the rest of the software. Controller synthesis results as well as "safety" properties analysis are also presented. Section 5 concludes the paper with a state of the art, an analysis of the current results and future work directions.

2 Modular Architecture for Autonomous Systems

At LAAS, researchers 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 robots (e.g. DALA, an iRobot ATRV) and is shown on Fig. 1. The levels in this architecture are :

[1] VERIMAG CNRS/University Joseph Fourier, Grenoble, France
[2] LAAS/CNRS, Unversity of Toulouse, Toulouse, France.

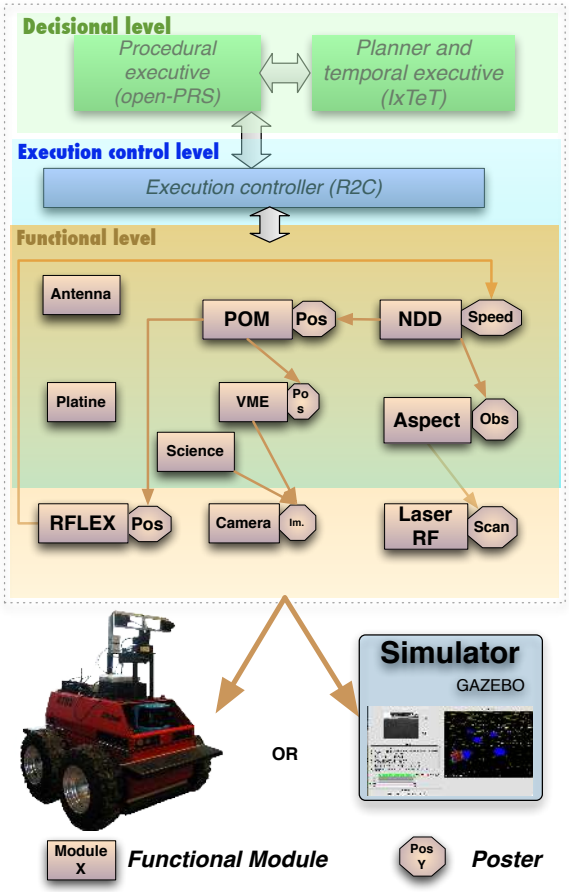

Figure 1. An instance of the LAAS architecture for the DALA Robot.

- 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 GenoM[3]. 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.
- 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 [14] 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.

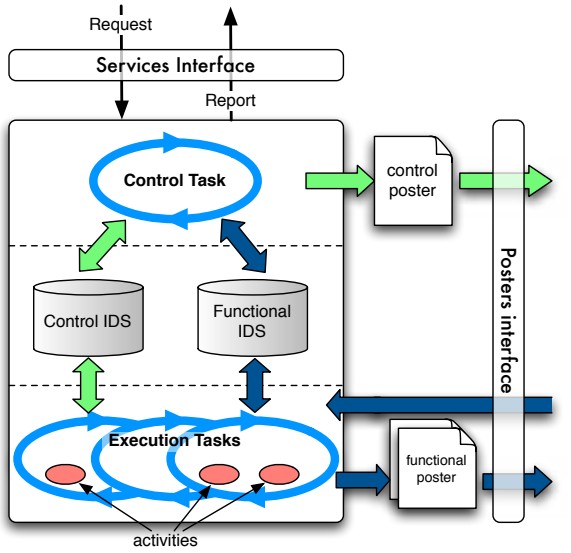

Figure 2. A GenoM module organization.

In this paper the componentization method we propose will allow us to synthesize a controller for the overall execution of all the functional modules 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 level* and *execution control level* with BIP.

2.1 GenoM 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 explicit periodical processing loop. The module Laser RF acquires the laser range finder and store them in the poster `Scan`, from which Aspect builds the obstacles map `Obs`. The module NDD (responsible for the navigation) avoids these obstacles while periodically 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 robots wheels, and also produces the odometry position to be used by POM to generate the current position.[4]

All these modules are built using a unique generic canvas (Fig. 2) which is then instantiated for a particular robot function.

Each *module* can execute several *services* started upon upper level 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 (*SetParams*, *SetDataSource* and*SetSpeed*), launching and stopping the path computation toward a given goal (*Stop* and *GoTo*) and a

[3] The GenoM tool can be freely downloaded from: `http://softs.laas.fr/openrobots/wiki/genom`

[4] This particular setup will serve as an example throughout the rest of the paper.

permanent service (*Permanent*). To execute this path, NDD exports the `Speed` poster which contains the speed reference.

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

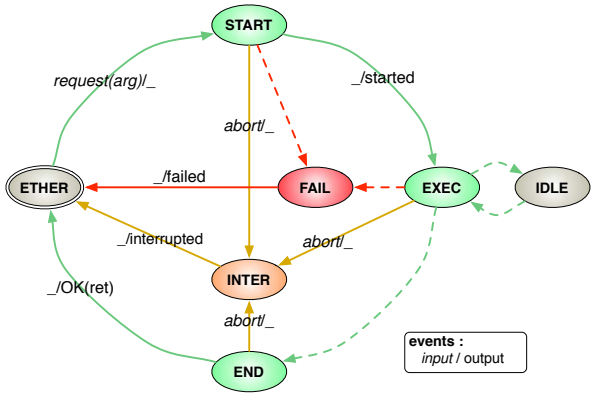

Figure 3. Execution automaton of an activity.

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 automata of an *activity*. Activity states correspond to the execution of particular elementary code (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.

3 The BIP Component Framework

BIP[5] [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 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 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

[5] The BIP tool-set can be downloaded from: `http://www-verimag.imag.fr/~async/BIP/bip.html`.

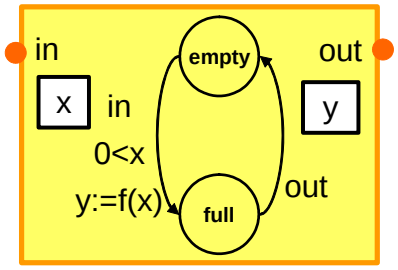

Figure 4. An example of an atomic component in BIP.

which may interact. If all the ports of a connector are incomplete then synchronization is by *rendezvous*. That is, only one interaction is possible, the interaction including all the ports of the connector. If a connector has one complete port then synchronization is by *broadcast*. That is, the complete port may synchronize with the other ports of the connector. The possible interactions are the non empty sublists containing this complete port. 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* [10]. 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.

4 The Functional Layer in BIP

The LAAS architecture makes use of a generic module for its functional layer. If we model this generic module and its components in BIP and if we then instantiate it and connect the existing "codels" to the resulting component, then we have a BIP model of the GenoM modules. Adding the BIP model of the interaction between the modules will give us a BIP model of the overall functional layer.

In order to formalize the componentization approach, we propose the following mapping (+ for one component or more, and . for composing components):

functional level ::= (module)+
module ::= (service)+ . (execution task) . (poster)+
service ::= (service controler) . (activity)
execution task ::= (timer) . (scheduler activity)

As shown in Fig. 5, a component modeling a generic *Service* is obtained from composing the atomic components *service controller* and *activity*. The left sub-component represents the execution task of a service. It is launched by synchronization through port *trigger*.

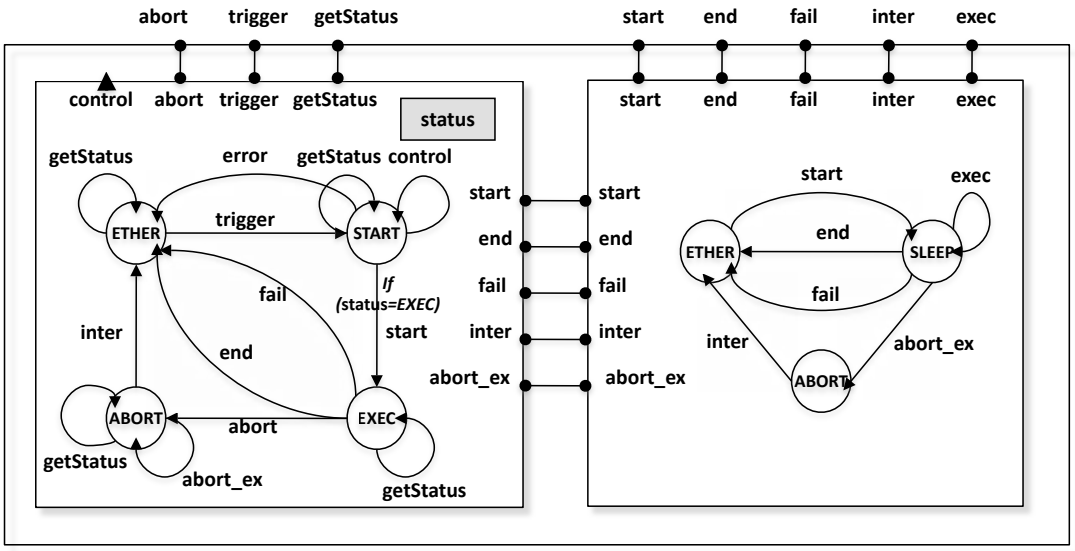

Figure 5. BIP model of a service.

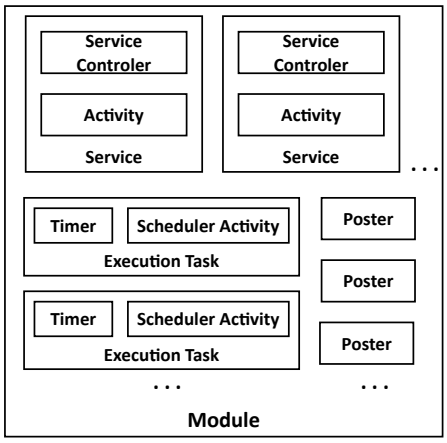

Figure 6. A componentized GenoM module.

The service controller then *controls* 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 subcomponent). 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 *end*, *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 *execution task* and *poster* components to obtain a *module* component (See Fig. 6).

4.1 A Functional Module in BIP

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.

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:

- 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).

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 GenoM modules, which executes the real activities of the robotic system. The code generated for the NDD module has been integrated and executed. In particular, it was fully integrated with the decisional layer by replacing the functional layer originally modeled with GenoM with the one modeled in BIP.

4.2 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 and 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

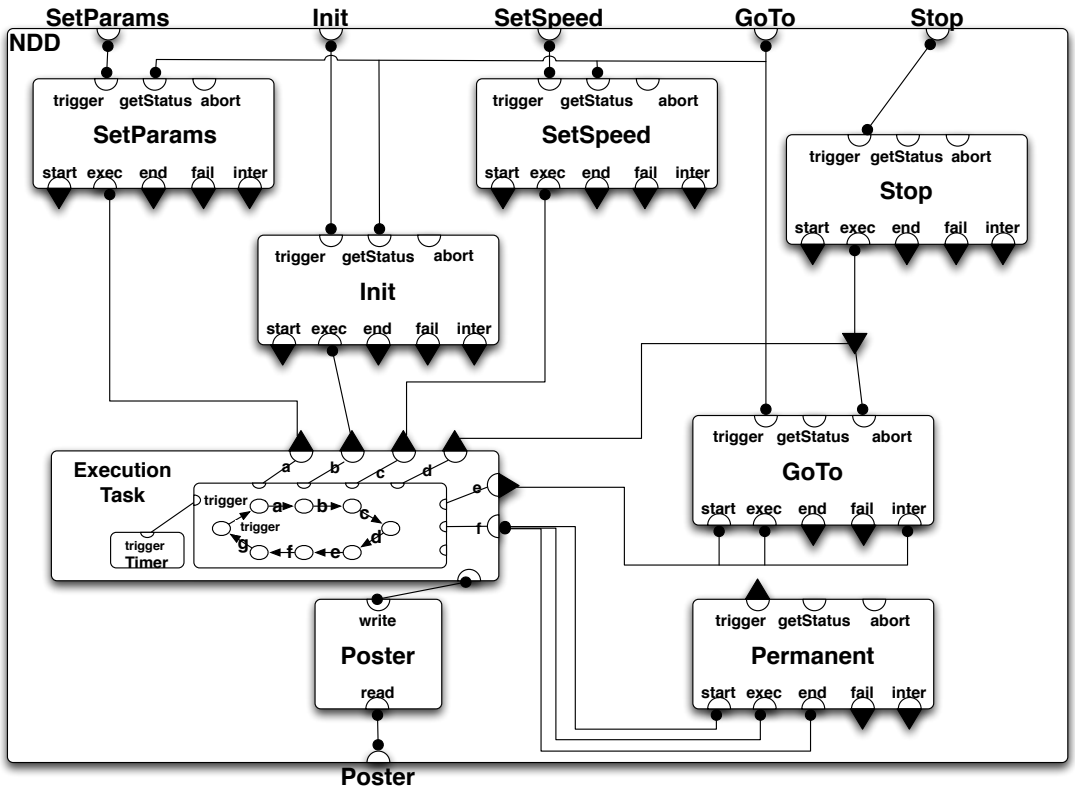

Figure 7. The NDD module.

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`). Such a 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.

4.3 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.

4.3.1 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.

Model checking [15, 3] We used the model-checker tool *Evaluator* [10] 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*. The result obtained by *Evaluator* proves that the initialization property is preserved in the NDD module.

Verification using Observers [17, 13] 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.

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

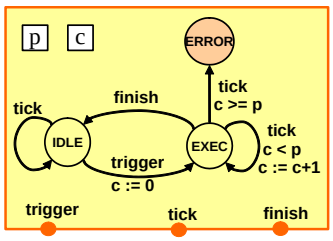

Figure 8. Observer for the control task period verification.

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.

4.3.2 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.

5 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 [12], 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] and T-REX [11], developed at NASA Ames and MBARI, have 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 [9, 18] 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 [16] 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 is 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. Even if 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 implemented and we now have a BIP controller for a subset of the functional layer of DALA, running in simulation and on the robot. The paper shows that it is possible to combine standard verification techniques, based on global state exploration, with structural analysis techniques for deadlock detection. A 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. Another work direction is to extend the BIP model to take into account the decisional capabilities of autonomous systems (action planning, execution control).

REFERENCES

[1] 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*, **17**(4), (1998).
[2] A. Basu, M. Bozga, and J. Sifakis, 'Modeling heterogeneous real-time components in BIP', in *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 *SAT*, Portofino, Italy, (2003).
[5] A. Finzi, F. Ingrand, and N. Muscettola, 'Robot action planning and execution control', in *IWPSS*, Darmstadt, Germany, (2004).
[6] S. Fleury, M. Herrb, and R. Chatila, 'GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture', in *IROS*, Grenoble, France, (1997).
[7] G. Goessler and J. Sifakis, 'Component-based construction of deadlock-free systems', in *FSTTCS*, Bombay, India, (2003).
[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] P. Kim, B. C. Williams, and M. Abramson, 'Executing reactive, model-based programs throgh graph-based temporal planning', in *IJCAI*, Seattle, WA, USA, (2001).
[10] 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).
[11] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, 'T-REX: A deliberative system for AUV control', in *ICAPS WS on Planning and Plan Execution for Real-World Systems*, Providence, RI, USA, (2007).
[12] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, 'CLARAty and challenges of developing interoperable robotic software', in *IROS*, Las Vegas, NV, USA, (2003).
[13] M. Phalippou, 'Executable testers', in *IWPTS*, Tokyo, Japan, (1994).
[14] F. Py and F. Ingrand, 'Dependable execution control for autonomous robots', in *IROS*, Sendai, Japan, (2004).
[15] J-P. Queille and J. Sifakis, 'Specification and verification of concurrent systems', in *Int. Symposium on Programming*, Torino, Italy, (1982).
[16] R. Simmons, C. Pecheur, and G. Srinivasan, 'Towards automatic verification of autonomous systems', in *IROS*, Takamatsu, Japan, (2000).
[17] J. Tretmans, 'A formal approach to conformance testing', in *IWPTS*, Tokyo, Japan, (1994).
[18] 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).