A Verifiable and Correct-by-Construction Controller for Robots in Human Environments*

Saddek Bensalem† Lavindra de Silva‡ Matthieu Gallien† Félix Ingrand‡ Rongjie Yan†

***Abstract*—Autonomous robots are complex systems that require the interaction and cooperation between numerous heterogeneous software components. In recent times, robots are being increasingly used to assist and replace humans. Consequently, robots are becoming critical systems that must meet safety properties, in particular, logical, temporal and real-time constraints. To this end, we present an evolution of the LAAS architecture for autonomous systems, and its tool GenoM. This evolution relies on the BIP component-based design framework, which has been successfully used in other domains such as embedded systems. We show how we integrate BIP into our existing methodology for developing the lowest (functional) level of robots. Particularly, we discuss the componentization of the functional level, the synthesis of an execution controller for it, and how we verify whether the resulting functional level conforms to properties such as deadlock-freedom. Our approach has been fully implemented in the LAAS architecture, and the implementation has been used in several experiments on a real robot.**

I. Introduction

As autonomous robots become more and more widespread, the need increases for robotic systems that are safe, dependable, and correct. This is particularly true for robots that have to interact regularly and in close contact with humans or other robots. Consequently, it will soon become commonplace for the developer of robot software to provide guarantees to certification bodies that, for instance, a hospital nursebot will not start moving too fast when an elderly person is leaning on it, that the arm of a service robot will not open its gripper while holding a bottle, or that there will not be a deadlock while a service robot is navigating in an office.

A certain level of dependability and safety can be provided with thorough software testing and extensive simulation. The goal of software testing is to "validate" and "verify" that the software meets a given set of requirements, and the goal of simulation is to detect errors as early as possible in the design phase. Unfortunately, both simulation and testing have the disadvantage of being incomplete, in the sense that each simulation run and each test evaluates the system only against a small subset of the foreseeable set of operating conditions and inputs. Hence, with complex autonomous and embedded systems it is often impractical to use these techniques to cover even a small fraction of the total operating space, not to mention the high cost of building test harnesses.

In this paper, we make a significant step toward building safe and dependable robotic architectures. Robotic architectures are typically organized into several levels, which usually correspond to different temporal requirements (e.g. TREX [1]) or different levels of abstraction of functionality (e.g. the LAAS architecture [2]). The lowest level of the latter type of architecture is the *functional* level, which includes all the basic, built-in action and perception capabilities such as image processing, obstacle avoidance, and motion control. We propose an approach for developing safe and dependable functional levels of complex, real-world robotic architectures. With our approach one can provide guarantees that the robot will not perform actions that may lead to states that are deemed unsafe, which may eventuate in undesired or catastrophic consequences.

Our solution relies on the integration of two state-of-the-art technologies, namely:

- GenoM [2] – a tool (part of the LAAS architecture toolbox) that is used for specifying and implementing the functional level of robots; and
- BIP [3] – a software framework for formally modeling complex, real-time component-based systems, with supporting toolsets for, among other things, verifying such systems.

This integration allows us to synthesize for our Dala robot a complete functional level that is correct by construction, which can be checked *offline* for properties such as deadlocks using verification tools and suites. Moreover, our integration allows safety constraints to be modeled and included, which can then be enforced *online* by the resulting controller. With the inclusion of such constraints, one can guarantee that the functional level will not reach unsafe states, even if bugs exist in user-supplied programs at higher levels of abstraction (e.g., the decisional level). Specifically, developing a functional level using our approach consists of the following steps: *(i)* developing the functional level using the GenoM tool of the LAAS architecture; *(ii)* translating the GenoM functional level into an equivalent BIP model; *(iii)* adding safety constraints into the generated BIP model; and *(iv)* verifying the model with the D-Finder [5] tool in our BIP tool-chain.

Then, we can summarize the contributions of this paper as follows. First, we provide algorithms and data structures for generating from a given GenoM functional level specification an equivalent BIP functional level. The BIP functional level can then be used in place of its GenoM counterpart. We provide an implemented tool that can automate this translation process. Second, we show, using a construction site inspection scenario for the decisional level, how the user can straightforwardly use BIP to specify and enforce different kinds of safety constraints

*Authors are in alphabetical order by last name. Part of this work is funded by the ESA/ESTEC GOAC project and by the FNRAE MARAE project.

†Verimag/CNRS, Grenoble I Uni., France. first.last@imag.fr
‡LAAS/CNRS, Toulouse Uni., France. first.last@laas.fr

on a generated BIP functional level. Third, we present results from using D-Finder to incrementally verify the generated BIP functional level. In particular, we prove that a substantial part of the BIP functional level is deadlock-free, and we report, for the first time, experiences in using D-Finder (e.g. solutions to deadlocks encountered) with a complex, real-world domain.

This paper is organized as follows. In Section II, we present the existing LAAS architecture and the BIP tool-chain; in Section III, we discuss in detail how to generate from a GenoM functional level an equivalent BIP functional level; in Section IV, we show how BIP can be used as a controller of the BIP functional level, in order to prevent the system from reaching "dangerous" states; in Section V, we show how D-Finder was used to analyze the BIP functional level for properties such as deadlocks. Finally, in Section VI, we present our conclusions and directions for future work.

II. Background

A. GenoM

The lowest level of most complex systems and robotic architectures is the *functional* level, which includes all the basic, built-in action and perception capabilities. These processing functions and control loops (e.g. image processing, obstacle avoidance, and motion control) are encapsulated into controllable, communicating modules. At LAAS, we use GenoM[1] [2] to develop these modules. Each module in the functional level of the LAAS architecture is responsible for a particular functionality of the robot. Complex modalities (such as navigation) are obtained by making modules "work together."

For example, the functional level of our Dala robot is shown in Figure 1. This functional level[2] includes two navigation modes. The first one, for mostly flat terrain, is laser based (`LaserRF`), and it builds a map (`Aspect`) and navigates using the near diagram (`NDD`) approach. In particular, *(i)* `LaserRF` acquires laser scans and stores them in the `Scan` poster, from which `Aspect` builds the obstacle map `Obs`; and *(ii)* `NDD` manages the navigation by avoiding these obstacles and periodically produces a speed reference to reach a given target from the current position `Pos` produced by `POM`. The speed reference produced by `NDD` is, in turn, used by `RFLEX`, which manages the low level robot wheels controller in order to control the speed of the robot. `RFLEX` also produces the current position of the robot based on odometry; this position is used by `POM` to generate the current position of the robot. The second navigation mode, for rough terrain, is vision based, and uses stereo images (`VIAM` and `Stereo`) to build a 3D map (`DTM`), which is used as input into an arc based trajectory planner (`P3D`). `P3D` also produces a speed reference which can be used by `RFLEX`. `Hueblob`, using panoramic images taken by `VIAM`, monitors potentially interesting features in the images. Finally, `Antenna` emulates communication with an inspector/operator PDA, and `Battery` emulates the management of the power on the whole platform.

[1] GenoM and other tools from the LAAS architecture can be freely downloaded from: `http://softs.laas.fr/openrobots/wiki/genom`

[2] Module names in Figure 1 are given in fixed font.

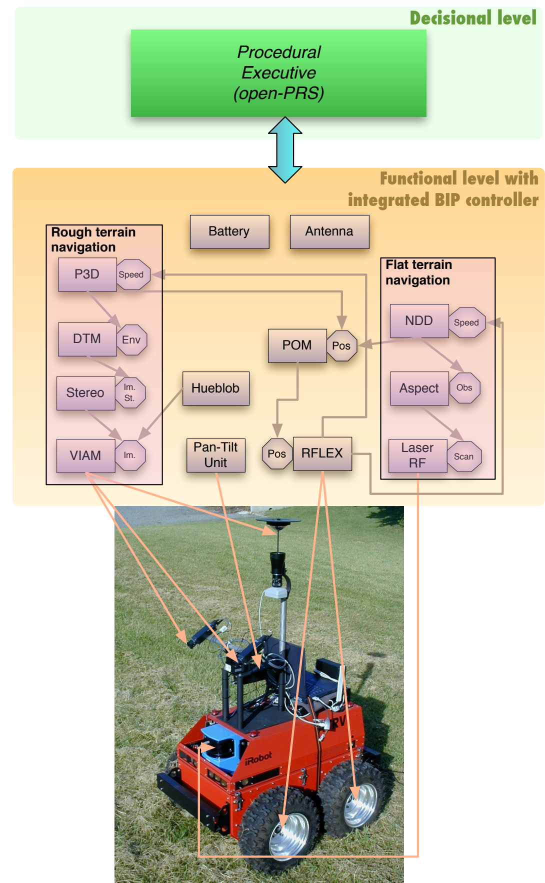

Fig. 1. The complete architecture of Dala.

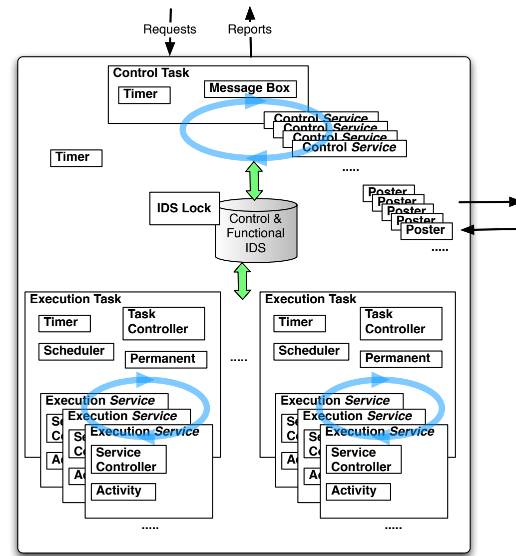

Fig. 2. A GenoM module functional organization and its componentization.

All these modules are built by instantiating a unique generic canvas. This canvas is shown in Figure 2. Each module provides *services*, which can be invoked by the higher (decisional) level according to tasks that need to be achieved. Services can be *execution services*, which initiate *activities* that take time

to execute, or *control services*, which take negligible time to execute and are responsible for setting and returning variable values.[3] For example, the NDD module provides five services corresponding to initializations of the navigation algorithm (*SetParams*, *SetDataSource*, and *SetSpeed*), and launching and stopping the path computation toward a given goal (*Stop* and *GoTo*). Execution services are managed by *execution tasks*, responsible for launching and executing activities within the associated running services. The remaining boxes in the figure correspond to BIP entities, which will be discussed in Section III.

Figure 3 presents the automaton of an activity. Transitions in the automaton correspond to the execution of particular elementary (C/C++) code, called *codels*, available through libraries. Codels actualize activities, and they are responsible for things such as initializing parameters (transition from *start* location), executing the "body" of the activity or its *main* codel (transition from *exec* location), and safely ending the activity, which may amount to things such as resetting parameters and sending error signals.

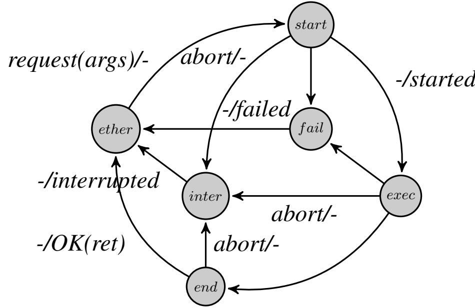

Fig. 3. The execution automaton of a GenoM activity. Transitions are of the form input/output.

Each module can return information to the caller – such as a final report – regarding the status of executed services, and export *posters* for others (modules or the decisional level) to read; posters store data produced by the module.

B. BIP

BIP [3] is a framework for modeling heterogeneous real-time programs. The main characteristics of BIP are the following:

- It supports a model-based design methodology where parallel programs are obtained as the superposition of three layers. The lowest layer describes behavior, the intermediate layer includes a set of connectors describing the interactions between transitions of the behavior, and the upper layer is a set of priority rules describing scheduling policies for interactions of the layer underneath. Such a layering offers a clear separation between behavior and structure (connectors and priority rules).
- It uses a parametrized composition operator on programs. The product of two programs is the composition of their corresponding layers separately. Parameters are used to define the interactions as well as new priority rules between the parallel programs [4]. The use of such a composition operator allows incremental construction, i.e., obtaining a parallel program by successive composition of other programs.
- It provides a powerful mechanism for structuring interactions involving strong synchronization (rendezvous) and weak synchronization (broadcast).

The BIP framework is implemented in the form a tool-chain. This is presented in Figure 4. The BIP tool-chain provides a complete implementation, with a rich set of tools for modeling, execution, analysis (both static and on-the-fly) and static transformations.

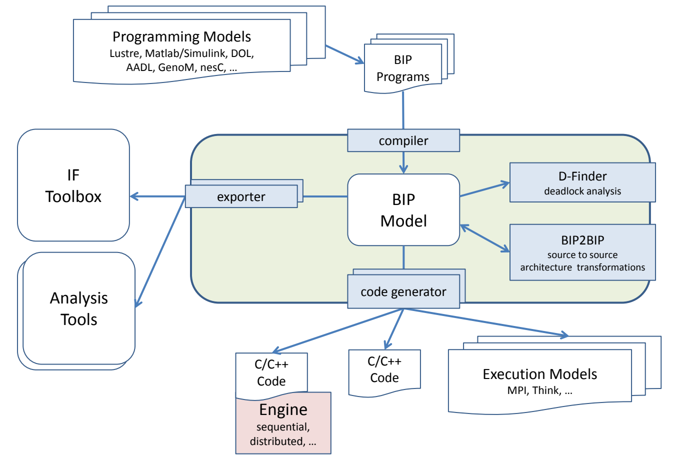

Fig. 4. The BIP tool-chain.

C. BIP *Language*

The BIP language supports a methodology for building components from: *(i)* atomic components, which are a class of components with behavior specified as a set of transitions and having empty interaction and priority layers, and where triggers (labels) of transitions are ports (action names) used for synchronization; *(ii)* connectors, used to specify possible interaction patterns between ports of atomic components; and *(iii)* priority relations, used to select amongst possible interactions according to conditions, whose valuations depend on the state of the integrated atomic components.

An atomic component consists of: *(i)* a set of ports $P = \{p_1 \dots p_n\}$, where ports are used for synchronization with other components; *(ii)* a set of control states/locations $S = \{s_1 \dots s_k\}$, which denote locations at which the components await synchronization; *(iii)* a set of variables V used to store (local) data; and *(iv)* a set of transitions modeling atomic computation steps.

A transition is a tuple of the form (s_1, p, g_p, f_p, s_2), representing a step from control state s_1 to s_2. A transition can be executed if the guard (boolean condition on V) g_p

[3] These variables are stored in a data structure called *Functional IDS*, which stores the data that is shared by all parts of a module.

is true and some interaction including port p is offered. Its execution is an atomic sequence of two microsteps: *(1)* an interaction including p, which involves synchronization between components with possible exchange of data, followed by *(2)* an internal computation specified by the function f_p on V.

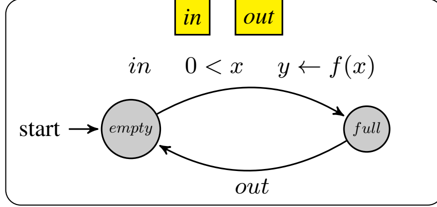

Fig. 5. A simple BIP atomic component.

Figure 5 shows a simple atomic component. This component has: two ports *in*, *out*; two variables *x*, *y*; and control locations *empty*, *full*. At control location *empty*, the transition labeled *in* is possible if $0 < x$. When an interaction through *in* takes place, the variable y is eventually modified when a new value for y is computed. From control location *full*, the transition labeled *out* can occur. The omission of the guard and function for this transition means that the associated guard is true and the internal computation microstep is empty. Note that in the rest of the paper, we do not show, for legibility, guards and functions in figures of BIP components. The BIP description of the reactive component of Figure 5 is the following:

```
component Reactive
  port in,out
  data int x,y
  behaviour
    state empty
      on in provided 0 < x do y ← f(x) to full
    state full
      on out to empty
  end
end
```

Components are built from a set of atomic components with disjoint sets of names for ports, control locations, variables and transitions. We simplify the notation for sets of ports by writing $p_1\|p_2\|p_3\|p_4$ for the set $\{p_1, p_2, p_3, p_4\}$. A connector γ is a set of ports of atomic components which can be involved in an interaction. We assume that connectors contain at most one port from each atomic component. An interaction of γ is any non empty subset of this set. For example, if p_1, p_2, p_3 are ports of distinct atomic components, then the connector $\gamma = p_1\|p_2\|p_3$ has seven interactions: $p_1, p_2, p_3, p_1\|p_2, p_1\|p_3, p_2\|p_3, p_1\|p_2\|p_3$. Each non trivial interaction, i.e., interaction with more than one port, represents a synchronization between transitions labeled with its ports. Given a connector γ, there are two basic modes of synchronization: *(i) strong* synchronization or *rendezvous*, when the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ; and *(ii) weak* synchronization or *broadcast*, when feasible interactions are all those containing a particular port which initiates the broadcast.

A connector description includes its set of ports followed by the optional list of its minimal complete interactions and its behavior. If the list of the minimal complete interactions is omitted, then this is considered to be empty. Connectors may have behavior specified as for transitions, by a set of guarded commands associated with feasible interactions. If $\alpha = p_1\|p_2\| \dots \|p_n$ is a feasible interaction, then its behavior is described by a statement of the form: on α provided G_α do F_α, where G_α and F_α are respectively a guard and a statement representing a function on the variables of the components involved in the interaction. An example of the syntax of a connector is given below. Note that this syntax is a simplified version to that given in the BIP literature.

```
connector conn(c1.p1, c2.p2)
define [c1.p1', c2.p2]
on c1.p1,c2.p2
provided g
do {c2.p2.v ← C}
on c1.p1
provided g
do {}
```

This connector, called *conn*, is a broadcast connector due to the inverted comma next to one of the ports. Port p_1 of component c_1 is the initiator of the broadcast synchronization between ports $c_1.p_1$ and $c_2.p_2$, where c_2 is a component and p_2 is one of its ports. If a strong synchronization involving both ports can occur, then a data transfer takes place, i.e., the variable v of port p_2 is assigned the constant C. No synchronization can take place unless guard g is met.

Finally, a compound component allows defining new components from existing sub-components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities. The instances can have parameters providing initial values to their variables through a named association.

D. *D-Finder*

The D-Finder tool implements a compositional [5] and incremental methodology [6] for the verification of component-based systems described in the BIP language [3]. D-Finder is mainly used to check safety properties of composite components. To check safety properties, D-Finder applies the compositional verification method proposed in [5], [6]. In this method, the set of reachable states is approximated by component invariants and interaction invariants. Component invariants are over-approximations of the set of the reachable states of atomic components and are generated by simple forward propagation techniques. Interaction invariants express global synchronization constraints between atomic components.

When we are concerned with the verification of deadlock properties, we let DIS be the set of global states in where a deadlock can occur. The tool will progressively find and

eliminate potential deadlocks as follows. D-Finder starts with a BIP model as input and computes component invariants CI by using the technique outlined in [5]. From the generated component invariants, it computes an abstraction of the BIP model and the corresponding interaction invariants II. Then, it checks satisfiability of the conjunction $II \wedge CI \wedge DIS$. If the conjunction is unsatisfiable, then there is no deadlock. Otherwise, either it generates stronger component and interaction invariants or it tries to confirm the detected deadlocks by using reachability analysis techniques. When verifying other safety properties with D-Finder, one of the steps is to replace DIS by the set of global states in which property violations occur. The other steps are identical to those of deadlock-freedom checking.

E. BIP *Engine*

The BIP tool-chain provides a platform for executing and analyzing the C++ application code generated by the front-end. The tool-chain includes an engine and the associated software infrastructure. The engine, entirely implemented in C++ on Linux, directly implements the operational semantics of BIP.

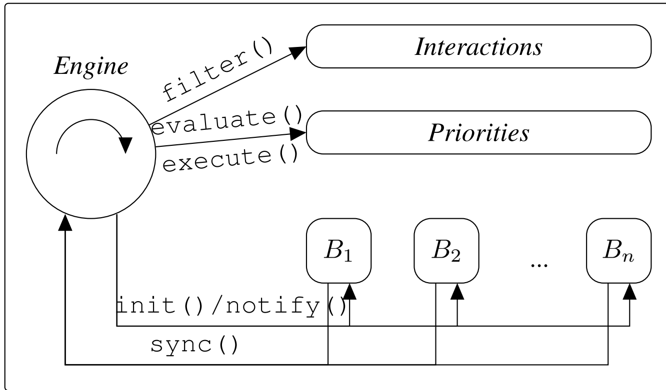

Fig. 6. The centralized engine architecture.

The engine works based upon the complete state information of the components. The execution follows a two-phase protocol, marked by the execution of the engine, and the execution of the atomic components. In the execution phase of the engine, it computes the interactions possible from the current state of the atomic components, and guards of the connectors. Then, between the enabled interactions, priority rules are applied to eliminate the ones with low priority. During this phase, the components are blocked, and await to be triggered by the engine. The engine selects a maximal enabled interaction, executes its data transfer, and triggers the execution of the atomic components associated with this interaction. The second phase is the execution of the local transitions of the notified atomic components. They continue their local computation independently and eventually reach new control states. Here, the atomic components notify of their enabled transitions to the engine and get blocked once more. The two phases are repeated, unless a deadlock is reached or the user wants to terminate the simulation. The scheme of the protocol is shown in Figure 6.

III. Componentization of the GenoM functional level

In this section, we discuss our algorithms for mapping a given GenoM functional level into an equivalent BIP functional level. We start with the mapping from individual GenoM modules to their BIP counterparts. Each GenoM module is mapped to a hierarchy of BIP components, as shown in Figure 2. In addition to representing some GenoM entity, each box in this figure also represents an atomic or compound BIP component.

In the componentization, an *Execution Task* is a compound component consisting of: a *Scheduler* (atomic) component, to control the execution of the associated *Activity* component of some *Execution Service* component; a *Task Controller* (atomic) component to stop the Scheduler if none of the associated Execution Service components are running; a *Timer* component to control the execution period of the Execution Task, provided it has a temporal period specified for it; and a *Permanent* component, provided the Execution Task has a permanent codel.

The *Poster* components store data associated with the module and provide operations for reading from and writing to this data. The *IDS Lock* component represents a semaphore for ensuring mutual exclusion between different Execution Task components and Execution Service components when manipulating Poster components. The Timer component (directly) in the Module component is used by Poster components to determine how much time has elapsed, in terms of “ticks,” since the last modification to their data. Specifically, Poster components contain a variable called *PosterAge* (initially 0) that is incremented for each “tick” in the associated Timer component, and reset whenever the poster is written to. The Service Controller, Activity, and Control Task components are discussed later.

As shown in Figure 2, some of the atomic components are combined to form compound components such as Execution Service. This is done by adding the necessary connectors between the atomic components. In turn, these compound components are combined using connectors to form the even more compound component Module, corresponding to a GenoM module. By combining components incrementally (or “bottom-up”) in this way, we have the guarantee that if its constituent components are proven to be correct with respect to some properties, then the resulting compound component will also be correct, provided it is free of deadlocks. Checking for deadlock-freedom using D-Finder will be discussed in Section V.

The most important components from those mentioned are *Message Box* (Figure 7), *Activity*, and *Service Controller* (Figure 8). Each Module component has, within its Control Task component, a Message Box component, which represents the interface for receiving requests for services and sending back replies. The period with which requests are read is controlled by the Timer component of the Control Task. There are two approaches for handling a newly received request in the Message Box: either *(i)* reject the request along with a specific report explaining the reason; or *(ii)* unconditionally

accept the request. The latter is done via two transitions. The first transition ($abtInc_{b_i}$) is for implementing a GenoM feature of interrupting certain execution services (Execution Service components) that are incompatible with the new request, and the second transition ($trig_{b_i}$) actually executes the request by interacting with either a Control Service or an Execution Service component.

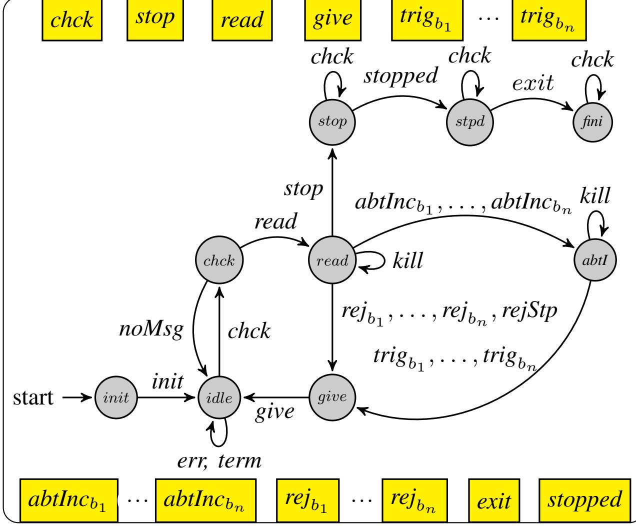

Fig. 7. An (atomic) BIP Message Box component. Transitions with multiple ports (separated by commas) represent multiple such transitions, each with one of the ports.

Each GenoM execution service has one corresponding Service Controller component, which controls its execution by, for example, checking the validity of the parameters (if any) of the request associated with the service, and handling the aborting of the service's execution. This component has two variables *active* and *done*, which are both initially *false*. The execution of the Service Controller starts via a synchronization with port *trig*, which sets *active* to *true*, after which the service can be aborted from any location via synchronization with the *abt* port. On the two transitions to location *ethr*, variable *active* is set to *false*, and variable *done* is set to *true* provided the transition labeled *fin* (denoting successful completion of the activity) was taken. Like a GenoM activity, the execution of the main codel of the Service Controller is initiated by the *exec* transition from the *exec* location. In each location of the Service Controller the status of the service can be obtained by synchronizing with the *stat* port of the component. Note that Service Controller components belonging to different Module components are equivalent except for the code executed during certain transitions (e.g., the ones labeled *ctrl* and *abt*), and that transitions labeled *abt* have higher priority than all other transitions, i.e., whenever a transition labeled *abt* and some other transition are both possible, the former will be taken instead of the latter.[4]

The Activity component corresponds to the GenoM activity automaton shown in Figure 3. This component will wait to be initiated, i.e., for a synchronization between its *start* port and the *start* port of Service Controller, and then wait for its main execution to begin, i.e., for a synchronization between its *exec* port and the *exec* port of Service Controller. In particular, this latter synchronization will, in the Activity component, lead to a transition that executes the main codel of the associated GenoM module. The Activity component is aborted by synchronizing with the *abt* port of the corresponding Service Controller; this synchronization will set a flag that prevents the Activity component from executing its main codel again.

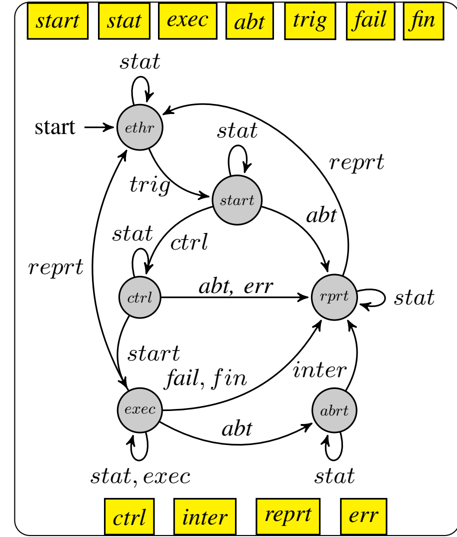

Fig. 8. An (atomic) BIP Service Controller component.

It is worth noting that, unlike its GenoM counterpart, each BIP Execution Service component has exactly one Service Controller component, and consequently, a given Execution Service component cannot have more than one Activity component associated with it (and executing concurrently). While this is indeed limiting, it can be overcome to a certain extent by defining multiple execution service types in the GenoM module, to represent the different activity instances that may be needed at runtime. In our experience, this solution was found to be a reasonable one for the modules that we have developed so far.

Observe from Figure 7 that, due to the semantics of BIP components, an interaction involving a $trig_{b_i}$ port of a Message Box component cannot happen concurrently with an interaction involving some other $trig_{b_j}$ port of the component. Similarly, we add the restriction that no interaction involving a $trig_{b_i}$ port of a Message Box component can happen concurrently with an interaction involving a $trig_{b_j}$ port of some *other* Message Box component (belonging to some other Module component). Without this restriction it would not be possible to add connectors (as we do later in Section IV) to guarantee that two services are not executed concurrently.[5] To add this restriction, we use a simple atomic component which represents a semaphore; its token is obtained by the

[4]This is necessary to ensure, for instance, that a request to abort a service is given priority over starting another step in its execution.

[5]Note that this restriction does not imply that two Service Controller components cannot be executed concurrently.

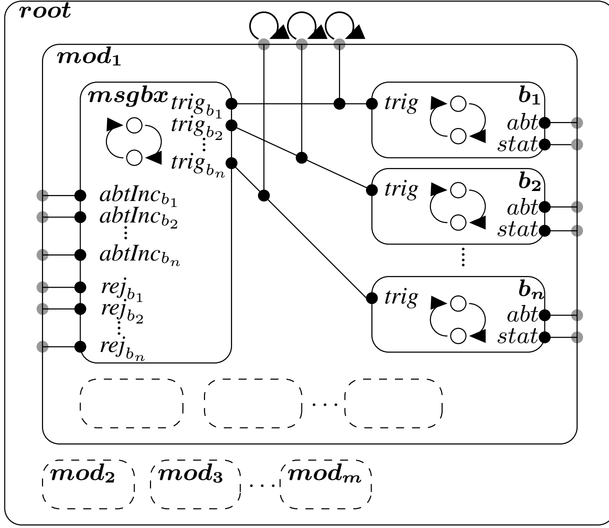

Fig. 9. A high-level illustration of exported ports (in grey), and important connectors within Module components ($\boldsymbol{mod_1, \ldots, mod_m}$). Connectors are between a Message Box ($\boldsymbol{msgbx}$) and the associated Service Controller components ($\boldsymbol{b_1, \ldots, b_n}$).

$read$ transition of Message Box components, and released by the $give$ transition of Message Box components.

As shown in Figure 9, each port $trig_{b_i}$ of a Message Box component is synchronized via rendezvous with port $b_i.trig$ of Service Controller component b_i. All such connectors are exported so that they are "visible" from the root component, i.e., it is possible to interact with them from the root component. The root component in BIP is the top-level (compound) component that includes all the other components. In our case, the root component includes all components of the functional level.

From now on, for convenience, we simply use $trig_{b_i}$ to refer to such an exported connector involving a port $trig_{b_i}$. For example, the exported port (grey circle) in Figure 9 that is associated with the connector involving $trig_{b_1}$ is also referred to as $trig_{b_1}$. Similarly, for each Service Controller component b_i, the rej_{b_i} and $abtInc_{b_i}$ ports of the associated Message Box component, as well as the $b_i.abt$ and $b_i.stat$ ports of b_i are exported so that it is possible to interact with them from the root component. As before, from now on we simply use rej_{b_i}, $abtInc_{b_i}$, $b_i.abt$ and $b_i.stat$ to refer to these exported ports. Unlike the other ports shown in Figure 9 (e.g., abt), by default, all $trig_{b_i}$ ports are possible due to a singleton connector with no guard (i.e., a "no-op" connector) at the root-component level for each $trig_{b_i}$ port. On the other hand, by default, all rej_{b_i}, $abtInc_{b_i}$, $b_i.abt$ and $b_i.stat$ ports are not available (i.e., synchronizations involving any of these ports are not possible) due to all such ports being left unconnected at the root-component level.

To ease the integration of BIP in the new framework, we have developed a tool that automatically produces a BIP model from a GenoM module description file. Still, if one wants to enforce some safety properties inside a module (intra-module) or between modules (inter-module), these constraints have to be explicitly added to the resulting BIP model. Adding such constraints will be discussed in the next section.

IV. Functional level controller synthesis

Since commands to the functional level are sent from the decisional level, i.e., the Procedural Reasoning System (PRS) [7] executive in our case, and since programs written for the decisional level may contain erroneous handcoded procedures, it is important to be able to constrain the decisional level so as to ensure the appropriate/safe execution of GenoM services in the functional level. For example, one may want to ensure that there is never a situation in which too much power is drawn from the battery, that the speed reference produced by a navigation mode is "fresh" enough with respect to the sensing data that it uses, or that the robot will not move when it is taking high resolution pictures or communicating.

In the previous LAAS architecture, the proper execution of GenoM services was managed by a centralized controller called R^2C [8]. The purpose of such a controller is to prevent the system from reaching dangerous states, such as those mentioned, which could lead to undesirable or catastrophic consequences. The R^2C controller maintains its own model and global state of the system. For the latter, R^2C monitors all requests sent from the decisional level to the functional level and all reports sent back from the functional level. If a request sent to the functional level may move the system into an undesirable (or unsafe) state, R^2C takes actions to prevent the state from being reached, such as killing a service or rejecting the request. Otherwise, R^2C allows the request to go through and the result to be returned.

One of the main differences between the R^2C approach and the BIP approach is that the former merely acts as a "filter" below the decisional level to enforce constraints between requests, while relying mostly on the control provided by GenoM. The BIP model and engine, on the other hand, go far beyond this by providing a formal and much finer grained model of the control taking place inside a functional module, which allows the user to specify finer grained constraints on the behaviour of functional modules. Moreover, in our new framework, we have one integrated system with a single model and single global state, rather than two systems (GenoM and R^2C) with two different models and two (possibly inconsistent) representations of the global state. Finally, by using BIP we now have a clearer semantics for constraints specified as BIP connectors, compared to the semantics of constraints specified in R^2C.

Before discussing how safety constraints can be encoded as BIP connectors in the functional level, we first discuss the construction site inspection scenario we have implemented for the PRS [7] based executive at the decisional level. We use this scenario to motivate and illustrate our constraints. The robot's duty is to work collaboratively with human inspectors and to assist them with the inspections. Our scenario consists of the robot exploring a set of waypoints initially supplied by the human inspector(s), which involves navigating to them and then taking images. Taking an image at a location involves aligning the high resolution cameras – mounted on the pan-and-tilt unit – to face the surfaces on the left and right sides of

the robot. During navigation, the robot continuously monitors its surroundings for potentially unsafe piles of (red) bricks, using the low resolution panoramic camera mounted on the mast. If such a pile is found, the robot stops navigating, determines if the pile is still within its front cameras' visibility area, and then takes a picture of the pile by aligning the front cameras toward it. The robot transmits all new images and associated waypoints to the PDAs of the human inspectors. Once all locations have been explored, the robot navigates back to its original location. Throughout the inspections the robot needs to stay clear of humans at work. Although not implemented in our scenario, one could extend it to make the robot opportunistically detect other potential hazards such as obstructions and cables strewn across walkways.

Now we can discuss in detail some of the constraints we have added into the BIP functional level (i.e., *root* component). We split the constraints into *intra-module* constraints, i.e., those between services belonging to a single module, and *inter-module* constraints, i.e., those between services belonging to two or more modules. In what follows, ports with suffixes *Trigger*, *Reject*, *Abort*, *Status*, and *AbortIncompatibleServices* are used to represent (respectively) particular $trig_b$, rej_b, $abtInc_b$, *b.abt*, and *b.stat* ports. Recall from Section III that all of these are exported ports.

A. Intra-module constraints

In the NDD module, there must be at least one successfully completed *SetParams* service,[6] and at least one successfully completed *SetSpeed* service before a *GoTo* service can be triggered. Note that *SPS* = *SetParamsStatus*, *SSS* = *SetSpeedStatus*, *GT* = *GotoTrigger*, and *GR* = *GotoReject*.

connector *AllowGotoIfArgsSet*(*ndd.GT*, *ndd.SPS*, *ndd.SSS*)
define [*ndd.GT*, *ndd.SPS*, *ndd.SSS*]
on *ndd.GT*, *ndd.SPS*, *ndd.SSS*
provided *ndd.SPS.done* ∧ *ndd.SSS.done*
do {}

connector *RejectGotoIfArgsNotSet*(*ndd.GR*, *ndd.SPS*, *ndd.SSS*)
define [*ndd.GR*, *ndd.SPS*, *ndd.SSS*]
on *ndd.GR*, *ndd.SPS*, *ndd.SSS*
provided ¬*ndd.SPS.done* ∨ ¬*ndd.SSS.done*
do {*ndd.GR.rep* ← *PARAMS-OR-SPEED-NOT-SET*}

For any module with an *Init* service, no other execution service of the module should be allowed unless the last instance of an executed service, if one exists, is a successfully completed *Init* service. The BIP connectors for this constraint are similar to the ones shown above.

B. Inter-module constraints

Next, we discuss constraints involving multiple modules. First, pictures should not be taken with any high resolution camera while the robot is moving, and vice versa, in order to prevent high resolution pictures from being blurred (this constraint does not apply to low resolution panoramic pictures). Hence, we say that moving is "incompatible" with taking a picture with a high resolution camera. To enforce this constraint, whenever a new request is received that is incompatible with a currently executing service, the latter is aborted with a specific error message and the new request is executed. In what follows, note that *AcS* = *AcquireStatus*, *TSST* = *TrackSpeedStartTrigger*, and *TSSR* = *TrackSpeedStart Reject*.[7] We only show the first two connectors; the other two (i.e., *AllowAcquireIfNotMoving* and *RejectAcquireIfMoving*) are analogous.

connector *AllowMoveIfNotAcquiring*(*rflex.TSST*, *viam.AcS*)
define [*rflex.TSST*, *viam.AcS*]
on *rflex.TSST*, *viam.AcS*
provided ¬(*viam.AcS.active* ∧
viamAcParams.bank.id = *"Marlin")*
do {}

connector *RejectMoveIfAcquring*(*rflex.TSSR*, *viam.AcS*)
define [*rflex.TSSR*, *viam.AcS*]
on *rflex.TSSR*, *viam.AcS*
provided *viam.AcS.active* ∧
viamAcParams.bank.id = *"Marlin"*
do {*rflex.TSSR.rep* ← *CANNOT-ACQ-AND-MOVE*}

Likewise, we have connectors to disallow taking pictures with the high resolution camera while the pan-and-tilt unit is moving, and vice versa, and connectors also to disallow communication with a PDA while moving, and vice versa, in order to ensure that communication is not disrupted. The connectors for the first constraint is similar to those shown above. The connectors for the second constraint are shown below. In what follows, *TSSA* = *TrackSpeedStartAbort*, *CAIS* = *CommunicateAbortIncompatibleServices*, *CT* = *Communicate Trigger*, and *TSSS* = *TrackSpeedStartStatus*. As before, we only show the first two connectors; the other two are analogous.

connector *AllowCommIfNotMoving*(*antenna.CT*, *rflex.TSSS*)
define [*antenna.CT*, *rflex.TSSS*]
on *antenna.CT*, *rflex.TSSS*
provided ¬*rflex.TSSS.active*
do {}

connector *AbortMovingToComm*(*antenna.CAIS*, *rflex.TSSA*)
define [*antenna.CAIS*′, *rflex.TSSA*]
on *antenna.CAIS*,*rflex.TSSA*
provided *true*
do {*rflex.TSSA.rep* ← *CANNOT-COMM-AND-MOVE*}
on *antenna.CAIS*
provided *true*
do {}

Finally, the following connector prevents poster data produced by certain modules from being used if the data is

[6]i.e., where the execution of the service returned a nominal report

[7]*Marlin* is the camera model, and *viamAcParams* is a variable that stores the camera model passed (as a parameter) with the most recent request to acquire an image.

not "fresh"; e.g., a speed reference produced by the NDD module is not "fresh" if it has not been updated for more than ten ticks. Recall that the *PosterAge* variable keeps track of the amount of time that has elapsed since the last time the associated Poster component was written to.

connector *AbtMoveIfPstrNotFresh*(*rflex.TSSA*, *ndd.PosterAge*)
define [*rflex.TSSA*, *ndd.PosterAge*]
on *rflex.TSSA*, *ndd.PosterAge*
provided *ndd.PosterAge* > 10
do $\{$*rflex.TSSA.rep* $\leftarrow$ *NDD-POSTER-NOT-FRESH*$\}$

From the constraints presented in this section, it is clear that the BIP language provides a natural syntax for encoding non-trivial constraints on certain aspects of BIP components. Such constraints are fundamental for synthesizing a controller for the functional level. To make it even more convenient for the user, we are currently in the process of developing a higher-level language for specifying constraints; these will then be automatically translated into BIP connectors, such as those shown in this section.[8]

V. Verification with D-Finder

The connectors added in the previous section could cause deadlocks in the functional level, since they amount to adding tighter constraints to certain subsets of components. To check whether the additional connectors may cause deadlocks, and to determine whether (atomic and compound) components by themselves are free of deadlocks, we use D-Finder to first verify atomic components and to then incrementally verify the compound components resulting from their composition. Due to space constraints, we do not discuss our experiences with using D-Finder for verifying properties other than deadlocks, such as "data freshness." We first discuss our results from the verification we carried out for compound components corresponding to individual G$^{en}_{o}$M modules. We start with a deadlock found while verifying the NDD (Module) component with D-Finder.

Figure 10 shows some of the components and associated connectors of the NDD component. Observe that there are three Timer components, one for the Control Task component, one for the Execution Task component, and one for the Poster component. The purpose of a Timer component is to make a *trigger* port available when the elapsed time in terms of "ticks" reaches a predefined value or period. To ensure that the duration between two contiguous *tick* synchronizations in a Timer component is equivalent to such a duration in any other Timer component, we strongly synchronize all *tick* ports of the mentioned Timer components with the *tick* port of the MasterTimer component. This component will effectively ensure that there are at least 10 milliseconds (ms) between two contiguous synchronizations involving all these *tick* ports.[9]

Although this design seemed correct, we found a non-trivial deadlock while verifying the NDD component with D-Finder. Intuitively, the reason for this deadlock is the strong synchronization between the Timer in Control Task and the Timer in Execution Task. Specifically, the deadlock scenario identified was the following: the Message Box is in location $abtI$; the Scheduler is in location $idle$; Execution Services *SetParams* and *GoTo* have started executing and they are respectively in locations $exec$ and $abrt$; variable t in the Timer of the Execution Task (ExecTaskTimer) has been reset to zero; and variable t in the Timer of the Control Task (InterfaceTimer) has reached the maximal value.

In this scenario, Scheduler is waiting to synchronize with the $trigger$ port of ExecTaskTimer in order to start the next round of execution; ExecTaskTimer is waiting for variable t in InterfaceTimer to be reset (via the synchronization involving its $trigger$ port) in order to continue with the synchronization between the four connected $tick$ ports; InterfaceTimer is waiting for Message Box to return to location $idle$ via location $give$, so that the InterfaceTimer can reset its variable t and perform the synchronization with the four connected $tick$ ports; and Message Box, after having aborted the *GoTo* service, is waiting to trigger the *Stop* service, in order to return to location $idle$ via location $give$ (see Figure 7). However, according to our mapping from G$^{en}_{o}$M to BIP, no condition on the transition corresponding to any $trig_{b_i}$ port in the Message Box component will be met because the *SetParams* and *GoTo* services have already started executing, and all other services in the NDD module have been declared by the user as incompatible with at least one of these services. Consequently, a deadlock state has been reached.

Our solution to this deadlock was to modify the connector synchronizing the *tick* ports to allow InterfaceTimer to not participate in the synchronization via the connector if it cannot participate. In precise terms, we have replaced the strong synchronization between the Timer components in Figure 10 with two new connectors, of which the first is shown below.

connector *ModuleSync*(*execTaskTimer.Tick*, *posterTimer.Tick*, *interfaceTimer.Tick*)
define [*execTaskTimer.Tick*, *posterTimer.Tick*]′, *interfaceTimer.Tick*
export *port Port moduleTick*

This connector, exported as *moduleTick*, executes a strong synchronization between the three *tick* ports if the *tick* port of the InterfaceTimer component is available, and otherwise, the connector executes a strong synchronization only between the *tick* ports of the ExecTaskTimer component and PosterTimer component. This solves the deadlock because it allows the Timer components inside the Module component to continue executing even if the Message Box component is waiting for

[8] Examples of constraints in this new language are $(r_1 \prec r_2)$ and $(r_1 \nparallel r_2)$, where the former means that request r_2 can only start after request r_1 has completed, and the latter means that the execution of r_1 should not be overlapped with the execution of r_2.

[9] More than 10 ms may be taken if at least one of the Timer components takes time to complete their *trigger* synchronizations. Waiting for 10 ms is implemented using the *usleep* system call. There is ongoing work [9] to extend BIP with the ability to model time, which will remove the need for the system call and improve the accuracy of measuring time.

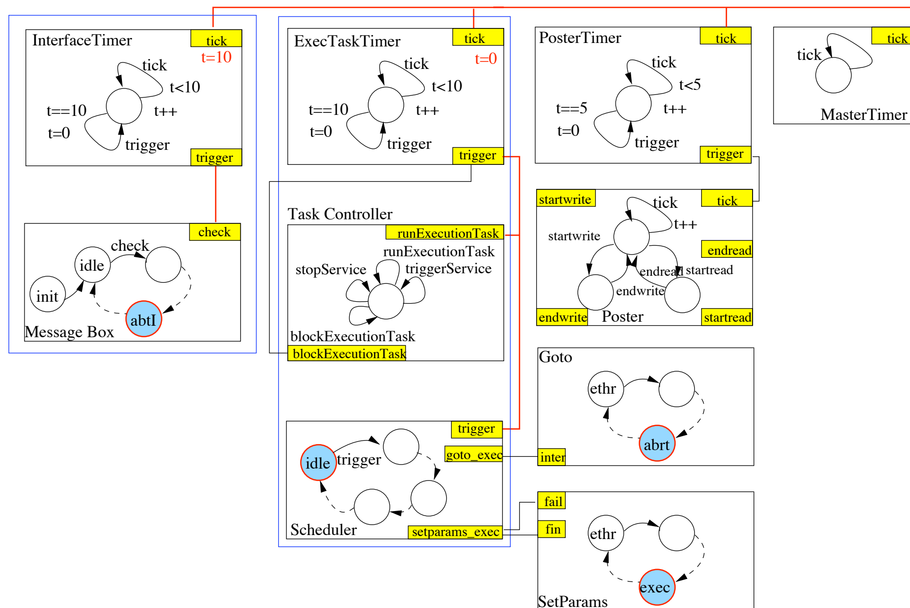

Fig. 10. A deadlock scenario caused by synchronization between Timer components. Dotted lines stand for ignored locations and transitions.

TABLE I
RESULTS FOR DEADLOCK-FREEDOM CHECKING.

Module	Components	Locations	Interactions	States	LOC	Minutes
LaserRF	43	213	202	$2^{20} \times 3^{29} \times 34$	4353	1:22
Aspect	29	160	117	$2^{17} \times 3^{23}$	3029	0:39
NDD	27	152	117	$2^{22} \times 3^{14} \times 5$	4013	8:16
RFLEX	56	308	227	$2^{34} \times 3^{35} \times 1045$	8244	9:39
Antenna	20	97	73	$2^{12} \times 3^{9} \times 13$	1645	0:14
Battery	30	176	138	$2^{22} \times 3^{17} \times 5$	3898	0:26
Platine	37	174	151	$2^{19} \times 3^{22} \times 35$	8669	0:59

a service to be aborted. The second connector is shown below.

connector *InterModuleSync* $(masterTimer.Tick, moduleTick_1, \ldots, moduleTick_n)$
define *masterTimer.Tick,* $moduleTick_1, \ldots, moduleTick_n$
on *masterTimer.Tick,* $moduleTick_1, \ldots, moduleTick_n$
provided *true*
do {}

This connector is for global synchronization between all Module components contained in the functional level, where $\{moduleTick_i\}_{1 \leq i \leq n}$ is a set of connectors of type *ModuleSync*, one for each Module component in the functional level composed of n Module components.

Table I shows the time taken for computing invariants for the deadlock-freedom checking of eight modules by D-Finder. *Module* is the name of the module; *Locations* is the total number of control locations in the module; *Interactions* is the total number of interactions in the module; *States* is the total number of states in the module – including those in its constituent components; *LOC* is the number of lines of (BIP) code in the module; and *Minutes* is the time taken for D-Finder to return a result. Observe from the table that we were able to check for the deadlock-freedom of all our modules in reasonable amounts of time, even for those consisting of thousands of lines of BIP code. This shows that D-Finder can be used to verify complex, real-world domains, and not just toy examples as shown in previous work [10]. It was already shown in [10], [6] that the component sizes handled by D-Finder are far beyond those that can be handled by other state of the art academic verification tools such as NuSMV [11] and SPIN [12].

Even after correcting individual modules with respect to deadlocks, it is still possible for collections of modules to contain deadlocks or to exhibit unsafe behaviour. Consider once again the two connectors described above, which synchronized all the Module components in the functional level by synchronizing a single MasterTimer component with the associated Timer components of all Control Task, Execution Task, and Poster components. As one might expect, with such a complex synchronization there may be the risk of a deadlock if one of the Timer components cannot perform a *tick* transition due to its *trigger* port never becoming available after the Timer's period is reached. Interestingly, we successfully verified using D-Finder that the synchronization between the Timer components belonging to NDD, Aspect and LaserRF modules are deadlock-free, and moreover, that the synchronization between the Timer components of NDD and RFLEX are also deadlock-free.[10] Unfortunately, because of the large state space, we were unable to check whether the synchronization between all related Module components are deadlock-free.[11] Improving D-Finder to make such an analysis possible is an avenue we intend to explore in the future.

[10]For NDD, Aspect and LaserRF, deadlock-freedom checking took 25 seconds, and for NDD and RFLEX deadlock-freedom checking took 66 minutes and 43 seconds.

[11]When we used D-Finder with four modules, it was unable to find a solution within two days.

VI. Conclusion

There are numerous works that address similar issues to what we address in this paper (e.g. [13], [14], [1], [15], [16], [17]). However, many of these frameworks do not present a formal model that allows to synthesize a controller that is correct by construction, and to verify safety properties on the resulting system. The remaining frameworks from those mentioned either do not address the componentization of the functional level, or they focus on the decisional level of the overall architecture whereas our work focuses on the functional level.

Despite the fact that software has become a large part of robot development, one must admit that the software models used up to now are either too coarse, too high level, or too large and thus very difficult to analyze. We propose a novel approach to developing functional levels of robotic systems, which incorporates a component-based design approach (BIP) in an existing architectural tool for developing functional modules (GenoM). Our approach allows the synthesis of a functional level that is correct by construction. To this end, we use our D-Finder tool to formally verify that a significant part of our functional level is deadlock-free, and that it conforms to other safety properties such as data freshness. Our approach also allows the synthesis of a controller that encodes and enforces user-supplied safety properties, thereby facilitating the development of safe and dependable robotic architectures.

We were able to run experiments with a complete functional and decisional level on the Dala robot, and to demonstrate via fault injections that the BIP engine successfully stops the robot from reaching undesired/unsafe situations like those discussed previously, and that it reports appropriately to the decisional level. In terms of runtime performance, experiments showed that by using the BIP engine, instead of using GenoM, as a controller of the functional level, the CPU load of the Pentium III machine on Dala is doubled. This is not surprising since the BIP engine must compute all the feasible interactions at each step in its execution. Nonetheless, since most real-world actions take time to execute (e.g., in our experiments, moving Dala from (x, y) coordinates $(0, 0)$ to $(4, 0)$ takes approximately 30 seconds), this overhead goes unnoticed in most cases.

We plan to extend our work in various directions: e.g., a real-time BIP engine to take into account wall clock properties, and a distributed engine to distribute it over more than one CPU. Another more ambitious research direction is to study the use of the BIP approach at the decisional level of our autonomous robot.

References

[1] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "A deliberative architecture for AUV control," in *Proc. of ICRA-08*, 2008, pp. 1049–1054.

[2] 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 *Proc. of IROS-97*, pp. 842–848.

[3] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Proc. of Int. Conf. on Software Engineering and Formal Methods (SEFM-06)*, 2006, pp. 3–12.

[4] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "Compositional verification for component-based systems and application," in *Proc. of Int. Symposium on Automated Technology for Verification and Analysis (ATVA-08)*, 2008, pp. 64–79.

[5] J. Sifakis, "A framework for component-based construction extended abstract," in *Proc. of SEFM-05*, 2005, pp. 293–300.

[6] S. Bensalem, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan, "Incremental invariant generation for compositional design," Verimag Research Report, Tech. Rep. TR-2010-6, 2010. [Online]. Available: http://www-verimag.imag.fr/TR/TR-2010-6.pdf

[7] M. Georgeff and F. Ingrand, "Decision making in an embedded reasoning system," in *Proc. of IJCAI-89*, 1989, pp. 972–978.

[8] F. Ingrand, S. Lacroix, S. Lemai, and F. Py, "Decisional autonomy of planetary rovers," *Journal of Field Robotics*, vol. 24, no. 7, pp. 559–580, 2007.

[9] T. Abdellatif, J. Combaz, and J. Sifakis, "Model-based implementation of real-time applications," Verimag Research Report, Tech. Rep. TR-2010-14, 2010. [Online]. Available: http://www-verimag.imag.fr/TR/TR-2010-14.pdf

[10] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "D-Finder: A tool for compositional deadlock detection and verification," in *Proc. of CAV*, 2009, pp. 614–619.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *Int. Journal on Software Tools for Technology Transfer*, vol. 2, pp. 410–425, 2000.

[12] G. J. Holzmann, *SPIN Model Checker, The: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "CLARAty and challenges of developing interoperable robotic software," in *Proc. of IROS-03*, 2003, pp. 2428–2435.

[14] A. Finzi and F. I. N. Muscettola, "Robot action planning and execution control," in *Proc. of Int. Workshop on Planning and Scheduling for Space*, 2004.

[15] P. Kim, B. C. Williams, and M. Abramson, "Executing reactive, model-based programs through graph-based temporal planning," in *Proc. of IJCAI-01*, 2001, pp. 487–493.

[16] R. P. Goldman, D. J. Musliner, and M. J. Pelican, "Using model checking to plan hard real-time controllers," in *Proc. of AIPS Workshop on Model-Theoretic Approaches to Planning*, 2000.

[17] R. Simmons, C. Pecheur, and G. Srinivasan, "Towards automatic verification of autonomous systems," in *Proc. of IROS-00*, 2000.