“Rock Solid” Software: A Verifiable and Correct-by-Construction Controller for Rover and Spacecraft Functional Levels

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

Abstract—With the increasing complexity of exploration rovers, satellites, and space probes, it has now become a challenge to prove that these systems will behave appropriately/safely in all situations that they may encounter. Such systems make use of complex hardware equipment, but also increasingly complex software. For hardware, one can typically use well known and mastered scientific models (e.g. thermal models, electrical models, mechanical models, and aerodynamic models) to predict how the equipment will behave alone and when integrated into the system. For the software, however, despite the confidence of the programmers who think that their piece of code is as “solid as a rock” and has no flaw (when used alone or with other components), unlike hardware, no scientific formal model exists which can take into account the ever increasing complexity of all the software components. To this end, we present an evolution of the LAAS architecture for autonomous systems, and its tool G®M. This evolution relies on the BIP component-based design framework, which has been successfully used in other domains such as embedded systems. 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 rover.

I. INTRODUCTION

As autonomous rovers and spacecraft are deployed for increasingly complex missions, the need increases for proving that these systems are safe, dependable, and correct. This is particularly true for rovers used in expensive and distant missions, such as Mars Rovers, that need to avoid equipment damage and minimize resource usage. Consequently, it may soon become common to require software integrators and developers to provide guarantees and formal proofs to space agencies that, for instance, a rover will not move while it is communicating or even worse, while it is drilling, or that the navigation software has no fatal deadlock.

A certain level of dependability and safety can be provided with thorough software testing and 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 and obstacle avoidance. 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: (i) G®M [2] – a tool (part of the LAAS architecture toolbox) that is used for specifying and implementing the functional level of robots and satellites; and (ii) BIP [3] – a software framework for formally modeling complex, real-time component-based systems, with supporting toolsets for verifying such systems. This integration allows us to synthesize for our Dala rover 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 the higher (decisional) level. Specifically, developing a functional level using our approach consists of the following steps: (i) developing the functional level using G®M; (ii) translating the functional level into an equivalent BIP model; (iii) adding safety constraints into the generated model; and (iv) verifying it with the D-Finder [4] tool in our BIP toolchain.

Then, we can summarize the contributions of this paper as follows. First, we provide algorithms and data structures for generating from a given G®M functional level specification an equivalent BIP functional level. The BIP functional level can then be used in place of its G®M counterpart. We provide an implemented tool that can automate this translation process.
Second, we show, using a Mars Rover scenario, how the user can straightforwardly use BIP to specify and enforce different kinds of safety constraints 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 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 how to generate from a $G^{en}M$ 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 Section V, we show we used D-Finder to analyze the BIP functional level. Finally, in Section VI, we present conclusions and directions for future work.

II. BACKGROUND

A. $G^{en}M$

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 $G^{en}M$ [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 rover 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 rover wheels controller in order to control the speed of the rover. RFLEX also produces the current position of the rover based on odometry; this position is used by POM to generate the current position of the rover. 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 orbiter/lander, and Battery emulates the management of the power on the whole platform.

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)

---

$^1$After $G^{en}M$ 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.
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 (transition from exec location), and safely ending the activity, which may amount to things such as resetting parameters and sending error signals.

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 Framework

BIP [3] is a framework for modeling heterogeneous real-time programs. The main characteristics of BIP are the following. First, 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). Second, BIP 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 [5]. The use of such a composition operator allows incremental construction, i.e., obtaining a parallel program by successive composition of other programs. Third, BIP provides a powerful mechanism for structuring interactions involving strong synchronization and weak synchronization.

1) BIP Language: The BIP language supports a methodology for building components from: (i) atomic components; (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 \( P = \{p_1, \ldots, p_n\} \), where ports are used for synchronization with other components; (ii) a set of control states/locations \( S = \{s_1, \ldots, s_k\} \), 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 \( (s_1, p, g, f, s_2) \), representing a step from control state \( s_1 \) to \( s_2 \). A transition can be executed if the guard (boolean condition on \( V \)) \( g \) is true and some interaction including port \( p \) is offered.

![Fig. 4: A simple BIP atomic component.](image)

Figure 4 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.

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 \( p_1 \parallel p_2 \parallel p_3 \parallel p_4 \) for the set \( \{p_1, p_2, p_3, p_4\} \). A connector \( \gamma \) 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 \( \gamma \) 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 \( \gamma = p_1 \parallel p_2 \parallel p_3 \) has seven interactions: \( p_1, p_2, p_3, p_1 \parallel p_2, p_1 \parallel p_3, p_2 \parallel p_3, p_1 \parallel p_2 \parallel p_3 \).

Each non trivial interaction, i.e., interaction with more than one port, represents a synchronization between transitions labeled with its ports. Given a connector \( \gamma \), there are two basic modes of synchronization: (i) strong synchronization or rendezvous, when the only feasible interaction of \( \gamma \) is the maximal one, i.e., it contains all the ports of \( \gamma \); and (ii) weak synchronization or broadcast, when feasible interactions are all those containing a particular port which initiates the broadcast. 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 \texttt{conn}(c_1, p_1, c_2, p_2)
define \{c_1, p_1, c_2, p_2\}
on \{c_1, p_1, c_2, p_2\}
provided \ g
do \{c_2, p_2 \cdot v \leftarrow C\}
on \{c_1, p_1\}
provided \ g
do \{}

This connector, called \texttt{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.

2) \textit{D-Finder:} The D-Finder tool implements a compositional\cite{4} and incremental methodology\cite{6} for the verification of component-based systems described in the BIP language\cite{3}. D-Finder is mainly used to check safety properties of composite components. To this end, D-Finder applies the compositional verification method proposed in \cite{4}, \cite{6}. In this method, the set of reachable states is approximated by the conjunction between 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.

III. COMPONENTIZATION OF \(\text{GenBM} \) FUNCTIONAL LEVEL

In this section, we discuss our algorithms for mapping a given \(\text{GenBM} \) functional level into an equivalent BIP functional level. We start with the mapping from individual \(\text{GenBM} \) modules to their BIP counterparts. Each \(\text{GenBM} \) module is mapped to a hierarchy of BIP components, as shown in Figure 2. In addition to representing some \(\text{GenBM} \) 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; and a Permanent component. 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 \(\text{PosterAge} \) (initially 0) that is incremented for each “tick” in the associated Timer component, and reset whenever the poster is written to. The Service Controller 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 \(\text{GenBM} \) 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.

The most important components from those mentioned are Message Box (Figure 5) and Service Controller (Figure 6).\footnote{While the Activity component is also important, we will not discuss it due to space constraints.} 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 (\textit{abtInc}_b) is for implementing a \(\text{GenBM} \) feature of interrupting certain execution services (Execution Service components) that are incompatible with the new request, and the second transition (\textit{trig}_b) actually executes the request by interacting with either a Control Service or an Execution Service component.

Fig. 5: An (atomic) BIP Message Box component. Transitions with multiple ports (separated by commas) represent multiple such transitions, each with one of the ports.

Each \(\text{GenBM} \) 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 G^M activity, the execution of the “body” or main code of the Execution Service is initiated by the exec transition from the exec location in the associated Service Controller. 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.  

As shown in Figure 7, each port trig_{bi} 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_{bi} to refer to such an exported connector involving a port trig_{bi}. For example, the exported port (grey circle) in Figure 7 that is associated with the connector involving trig_{bi} is also referred to as trig_{bi}. Similarly, for each Service Controller component b_i, the ref_{bi} and abtInc_{bi} 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 ref_{bi}, abtInc_{bi}, b_i.abt and b_i.stat to refer to these exported ports. Unlike the other ports shown in Figure 7 (e.g., abt), by default, all trig_{bi} 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_{bi} port. On the other hand, by default, all ref_{bi}, abtInc_{bi}, 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 G^M 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 G^M 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 or that the speed reference produced by a navigation mode is “fresh” enough with respect to the sensing data that it uses.

In the previous LAAS architecture, the proper execution of G^M services was managed by a centralized controller.
called R\textsuperscript{2}C [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. One of the main differences between the R\textsuperscript{2}C 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 G\textsuperscript{GR}M. 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 (G\textsuperscript{GR}M and R\textsuperscript{2}C) 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\textsuperscript{2}C.

Before discussing how safety constraints can be encoded as BIP connectors, we first discuss the Mars Rover scenario we have implemented for the PRS [7] based executive at the decisional level. We use this scenario to motivate and illustrate our constraints. Our scenario consists of the rover heating up to a given temperature, and then exploring a predefined set of locations, which involves navigating to them and then taking science pictures. Taking a science picture involves aligning the high resolution cameras – mounted on the pan-and-tilt unit – to face the surfaces near the left and right front wheels of the rover. During navigation, the rover continuously monitors its surroundings for bright red rocks using the low resolution panoramic camera mounted on the mast. If such a rock is found, the rover stops navigating, determines if the rock is still within its front cameras’ visibility area, and then takes a picture of the rock by aligning the front cameras toward it. The rover transmits all new images to the orbiter during given visibility windows. Once all locations have been explored, the rover navigates back to its original location.

Now we 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 and inter-module constraints. In what follows, ports with suffixes Trigger, Reject, Abort, Status, and AbortIncompatibleServices are used to represent (respectively) particular trig\textsubscript{b}, rej\textsubscript{b}, abtInc\textsubscript{b}, 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,\footnote{i.e., where the execution of the service returned a nominal report.} and at least one successfully completed SetSpeed service before a GoTo service can be triggered. Note that SPS = SetParamStatus, SSS = SetSpeedStatus, GT = GotoTrigger, and GR = GotoReject.

```plaintext
class Define [ndd.GT, ndd.SPS, ndd.SSS]
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.GR.done ∨ ¬ndd.SPS.done do {ndd.GR.rep ← PARAMS-OR-SPEED-NOT-SET}

B. Inter-module constraints

Next, we discuss constraints involving multiple modules. First, pictures should not be taken with any high resolution camera while the rover 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. 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 an orbiter while moving, and vice versa, in order to ensure that communication is not disrupted. The connectors for the last constraint are shown below. In what follows, TSSA = TrackSpeedStartAbort, CAIS = CommunicateAbortIncompatibleServices, CT = Communicate Trigger, and TSSS = TrackSpeedStartStatus. We only show the first two connectors; the other two are analogous.

```plaintext
class Define [antenna.CT, reflex.TSSS] on antenna.CT, reflex.TSSS provided ¬reflex.TSSS.active do {} connector AllowCommIfNotMoving(antenna.CT, reflex.TSSS)
define [antenna.CAIS, reflex.TSSA] on antenna.CAIS, reflex.TSSA provided true do {reflex.TSSA.rep ← CANOT-COMM-AND-MOVE} on antenna.CAIS provided true do {} connector AbortMovingToComm(antenna.CAIS, reflex.TSSA)
define [antenna.CAIS, reflex.TSSA] on antenna.CAIS, reflex.TSSA provided true do {reflex.TSSA.rep ← CANOT-COMM-AND-MOVE} on antenna.CAIS provided true do {} connector AbortIfPstrNotFresh(reflex.TSSA, ndd.StartRead)
define [reflex.TSSA, ndd.StartRead] on reflex.TSSA, ndd.StartRead provided ndd.PostersAge > 10 do {reflex.TSSA.rep ← NDD-POSTER-NOT-FRESH}
```
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 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 start with a deadlock found while verifying the NDD (Module) component with D-Finder.

Figure 8 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.\(^7\)

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 5). However, according to our mapping from \(G^{\text{BIP}}\) to BIP, no condition on the transition corresponding to any trig\(_0\) 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 8 with two new connectors, of which the first is shown below.

- **connector** ModuleSync\((\text{execTaskTimer.Tick}, \text{posterTimer.Tick}, \text{interfaceTimer.Tick})\)
- **define** \(\text{execTaskTimer.Tick}, \text{posterTimer.Tick}\), \(\text{interfaceTimer.Tick}\)
- **export** Port moduleTick

This connector, exported as moduleTick, 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 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
  \[\text{do } \{}\]

This connector is for global synchronization between all Module components contained in the functional level, where \(\{\text{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-free checking of eight modules by D-Finder.\(^8\) 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 [9]. It was already shown in [9], [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 [10] and SPIN [11].

Finally, 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.\(^9\) Unfortunately, because of the large state space, we were unable to check whether the synchronization between all related Module

\(^7\)More than 10 ms may be taken if at least one of the Timer components takes time to complete their trigger synchronizations.

\(^8\)Module is the name of the module; Locations is the number of control locations in the module; Interactions is the number of interactions in the module; States is the 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.

\(^9\)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.

---

**TABLE I: Results for deadlock-freedom checking.**

<table>
<thead>
<tr>
<th>Module</th>
<th>Components</th>
<th>Locations</th>
<th>Interactions</th>
<th>States</th>
<th>LOC</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaserRF</td>
<td>45</td>
<td>213</td>
<td>202</td>
<td>(2^{30} \times 3^{45} \times 34)</td>
<td>4385</td>
<td>1:22</td>
</tr>
<tr>
<td>Aspect</td>
<td>29</td>
<td>160</td>
<td>117</td>
<td>(2^{37} \times 3^{23})</td>
<td>3029</td>
<td>0:39</td>
</tr>
<tr>
<td>NDD</td>
<td>27</td>
<td>152</td>
<td>117</td>
<td>(2^{22} \times 3^{14} \times 5)</td>
<td>4013</td>
<td>8:16</td>
</tr>
<tr>
<td>RFLEX</td>
<td>56</td>
<td>308</td>
<td>227</td>
<td>(2^{20} \times 3^{35} \times 1045)</td>
<td>8244</td>
<td>9:39</td>
</tr>
<tr>
<td>Antenna</td>
<td>20</td>
<td>97</td>
<td>73</td>
<td>(2^{12} \times 3^{8} \times 11)</td>
<td>1645</td>
<td>0:14</td>
</tr>
<tr>
<td>Battery</td>
<td>30</td>
<td>176</td>
<td>138</td>
<td>(2^{22} \times 3^{7} \times 5)</td>
<td>3398</td>
<td>0:26</td>
</tr>
<tr>
<td>Heating</td>
<td>26</td>
<td>149</td>
<td>116</td>
<td>(2^{11} \times 3^{24} \times 145)</td>
<td>2453</td>
<td>0:17</td>
</tr>
<tr>
<td>Platine</td>
<td>37</td>
<td>174</td>
<td>151</td>
<td>(2^{15} \times 3^{22} \times 35)</td>
<td>8669</td>
<td>0:59</td>
</tr>
</tbody>
</table>
components are deadlock-free. Improving D-Finder to make such an analysis possible is an avenue we intend to explore in the future.

VI. CONCLUSION

There are numerous works that address similar issues to what we address in this paper (e.g. [12], [13], [1]). 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. Other frameworks 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 (G both M). 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 rover, and to demonstrate via fault injections that the BIP engine successfully stops the rover 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 G both M, 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.

REFERENCES

