Rigorous design of robot software: A formal component-based approach

Tesnim Abdellatif, Saddek Bensalem, Jacques Combaz, Lavindra de Silva, Félix Ingrand

Rigorous Design of Robot Software: A Formal Component-Based Approach ☆,☆☆

Tesnim Abdellatif[a], Saddek Bensalem[a], Jacques Combaz[a], Lavindra de Silva[b], Felix Ingrand[b,*]

[a] *Verimag/CNRS, Grenoble I Uni., France.*
[b] *LAAS/CNRS, Toulouse Uni., France.*

1. Introduction

For the large scale deployment of robots in places such as homes, shopping centers and hospitals, where there is close and regular interaction with humans, robot software integrators and developers may soon need to provide guarantees and formal proofs to certification bodies that their robots are safe, dependable, and behave correctly. This also applies to robots such as extraterrestrial rovers, used in expensive and distant missions, which need to avoid equipment damage and mission failure. Such guarantees may involve proofs that a rover will not move while it is communicating or even worse, while it is drilling, that the navigation software has no fatal deadlock, or that a service robot will not extend its arm dangerously while navigating or will not open its gripper while holding a breakable object.

The most common method to ensure the correctness of a system is testing (see Broy et al. (2005) for a survey). Testing techniques have been effective for finding bugs in many industrial applications. Unfortunately, there is, in general, no way for a finite set of test cases to cover all possible scenarios, and therefore, bugs may remain undetected. Hence, in general, testing does not give any guaran-

☆ 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. We thank Rongjie Yan for some useful discussions.

* Corresponding Author.

Email addresses: tesnim.abdellatif@imag.fr (Tesnim Abdellatif), saddek.bensalem@imag.fr (Saddek Bensalem), jacques.combaz@imag.fr (Jacques Combaz), ldesilva@laas.fr (Lavindra de Silva), felix@laas.fr (Felix Ingrand)

tees on the correctness of the entire system. Consequently, these approaches are impractical with complex autonomous and embedded systems for even a small fraction of the total operating space.

We have successfully proposed a novel software engineering methodology for developing safe and dependable robotic systems (Bensalem et al., 2010a, 2011). With our approach, one can provide guarantees that the robot will not perform actions that may lead to situations deemed unsafe, i.e., those that may eventuate in undesired or catastrophic consequences. Our approach (Section 3) relies on the integration of two existing state-of-the-art methods, namely, *(i)* the GenoM tool of the LAAS architecture (Fleury et al., 1997), used for specifying and implementing the lowest level of robotic systems, and *(ii)* the BIP software framework for formally modeling and verifying complex, real-time component-based systems (Basu et al., 2006). In this paper, we extend this approach to be used on complex robotic systems for designing both the decisional and functional levels. We first present (in Section 4) a high-level language that allows roboticists to easily express specific constraints on the system. In light of early experimental findings, we also provide insights into more recent work focused on real-time features of a system. Indeed, we developed a real-time version of BIP (Section 5), which takes into account execution time and deadlines. We also used the real-time BIP engine as a temporal-plan execution controller (Section 6). Section 7 presents results on all of the above work. We then conclude the paper with a future work section (Section 8), presenting a multi-CPU distributed version of BIP which allows us to run it on modern robotic platforms, and how we plan to use GenoM3 to extend our approach toward the ROS ecosystem, and a discussion section (Section 9) .

2. State of the Art in Building Robot Software using Formal Methods

Despite a growing concern to develop safe, robust, and verifiable robotic systems, overall, robot software development remains quite disconnected from the use of formal methods. Moreover, the extent to which formal methods has been used is quite different between the functional level and the decisional level of robot software architectures.

2.1. The Decisional Level Design

Formal methods have been more widely used together with "decisional components" of robotic systems. The main reason is perhaps because these decisional components already use a "model" (for planning, diagnostics, etc.). In (Williams et al., 2003), the authors propose a system relying on a model-based approach.

The objective is to abstract the system into a state transitions based language modeling the dependability concerns. The programmers specify state evolutions with invariants and a controller executes this maintaining these invariants. To do that, the controller estimates the most likely current state—using observation and a probabilistic model of physical components—and finds the most reliable sequence of commands to reach a specified goal (i.e., with a minimum probability of failure). In (Goldman et al., 2000), the authors present the CIRCA SSP planner for hard real-time controllers. This planner synthesizes off-line controllers from a domain description (preconditions, postconditions and deadlines of tasks). CIRCA SSP can then deduce the corresponding timed automaton to control the system on-line, with respect to these constraints. This automaton can be formally validated with model checking techniques. Similarly, (Bordini et al., 2003) discusses an approach for model checking the AgentSpeak(L) agent programming language aimed at reactive planning systems. The work describes a toolkit called CASP (Checking AgentSpeak Programs) for supporting the use of model checking techniques, in particular, for automatically translating AgentSpeak(L) programs into a language understood by a model checker. In (Simmons et al., 2000), the authors present a system which allows the translation from MPL (Model-based Processing Language) and TDL (Task Description Language)—the executive language of the CLARAty architecture(Nesnas et al., 2003)— to SMV, a symbolic model checker language.

In (Kress-Gazit and Pappas, 2010), the authors discuss an approach for automatically generating correct-by-construction robot controllers from high-level representations of tasks given in Structured English, which are translated into a subset of Linear Temporal Logic and eventually into automata. In their work, complex and continuous missions can be specified using the basic prepositions 'between', 'near,' 'within,' 'inside,' and 'outside.' An example of such a mission is "stay near A unless the alarm is sounding," where A is a location. Likewise, (Wongpiromsarn et al., 2010) also deals with the synthesis of correct-by-construction controllers based on temporal logic specifications. Here, finite state automata based controllers are synthesized by a trajectory planner to satisfy a given temporal specification, which is based on an abstract model of the physical system. The authors show how the correct behavior of an autonomous vehicle can be maintained using the robot controller automatically synthesized.

2.2. The Functional Level Design

On the functional side of robotic systems, the situation is quite different. There are many popular software tools available (e.g., OROCOS (Bruyninckx, 2001),

CARMEN (Montemerlo et al., 2003), Player Stage (Vaughan and Gerkey, 2007), Microsoft Robotics Studio (Jackson, 2007), and ROS (Quigley et al., 2009)) to develop the functional level of robotic systems. There are even some works which compare them, e.g., (Shakhimardanov and Prassler, 2007; Kramer and Scheutz, 2007). Yet, none of these architectural tools and frameworks proposes any extension or link with formal methods, and validation or verification tools.

Recently, we proposed the R^2C (Ingrand et al., 2007), a tool used between the functional and decisional levels of a robotic system. The main component of R^2C is the *state checker*. This component encodes the constraints of the system, specified in a language named Ex°Gen. At run-time it continuously checks if new requests are consistent with the current execution state and the model of properties to enforce. Another interesting early approach to prove various formal properties of the functional level of robotic systems is the ORCCAD system (Espiau et al., 1995). This development environment, based on the Esterel (Boussinot and de Simone, 1991) language, provides extensions to specify robot "tasks" and "procedures." However, this approach remains constrained by the synchronous systems paradigm.

More generally, as advocated in (Bensalem et al., 2008b), an important trend in modern systems engineering is model-based design, which relies on the use of explicit models to describe development activities and their products. It aims at bridging the gap between application software and its implementation by allowing predictability and guidance through analysis of global models of the system under development. The first model-based approaches, such as those based on ADA, synchronous languages (Halbwachs, 1992) and Matlab/Simulink, support very specific notions of components and composition. More recently, modeling languages, such as UML (Jacobson et al., 1999) and AADL (Feiler et al., 2006), attempt to be more generic. They support notions of components that are independent from a particular programming language, and put emphasis on system architecture as a means to organize computation, communication, and implementation constraints. Software and system component-based techniques have not yet achieved a satisfactory level of maturity. Systems built by assembling together independently developed and delivered components often exhibit pathological behavior. Part of the problem is that developers of these systems do not have a precise way of expressing the behavior of components at their interfaces, where inconsistencies may occur. Components may be developed at different times and by different developers with, possibly, different uses in mind. Their different internal assumptions, when exposed to concurrent execution, can give rise to unexpected behavior, e.g., race conditions, and deadlocks.

All these difficulties and weaknesses are amplified in embedded robotic systems design in general. They cannot be overcome, unless we solve the hard fundamental problems concerning the definition of rigorous frameworks for component-based design.

3. Our Approach

In past work we have proposed an approach (Bensalem et al., 2009, 2011) to develop safe and dependable functional levels of complex, real-world robots, which relied on the integration of two state-of-the-art technologies, namely: *(i)* GenoM (Fleury et al., 1997)—a tool (part of the LAAS architecture toolbox) that is used for specifying and implementing the functional level of robots; and *(ii)* BIP (Basu et al., 2006)—a software framework for formally modeling complex, real-time component-based systems, with supporting tool-sets for verifying such systems.

3.1. GenoM *: The LAAS Architecture Tool*

The *functional* level is the lowest level of most robotic systems. It includes all the basic, built-in action and perception capabilities. These processing functions and control loops (e.g., image processing, obstacle avoidance, and motion planning) are encapsulated into controllable, communicating modules. At LAAS, we have developed GenoM[1] (Fleury et al., 1997) to generate these modules by instantiating a generic canvas (see Figure 1) whose components and algorithms have been developed and debugged for more than 15 years.

Figure 2 is an example of a functional level belonging to our Dala rover (Bensalem et al., 2010a). This functional level[2] consists of two navigation modes. The first one, for mostly flat terrain, is laser based. `LaserRF` acquires the laser scan, `Aspect` builds a 2D obstacles map, and `NDD` navigates by producing a speed reference used by the robot wheel controller `RFLEX`. The second navigation mode, for rough terrain, is vision based. `VIAM` takes stereo images, `Stereo` correlates them and passes them onto `DTM` to build a 3D map, which is used by the trajectory planner `P3D`. Other modules implement opportunistic science (`Hueblob`), and emulate communication (`Antenna`) and power and energy management (`Battery`).

[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 2 are given in fixed font.

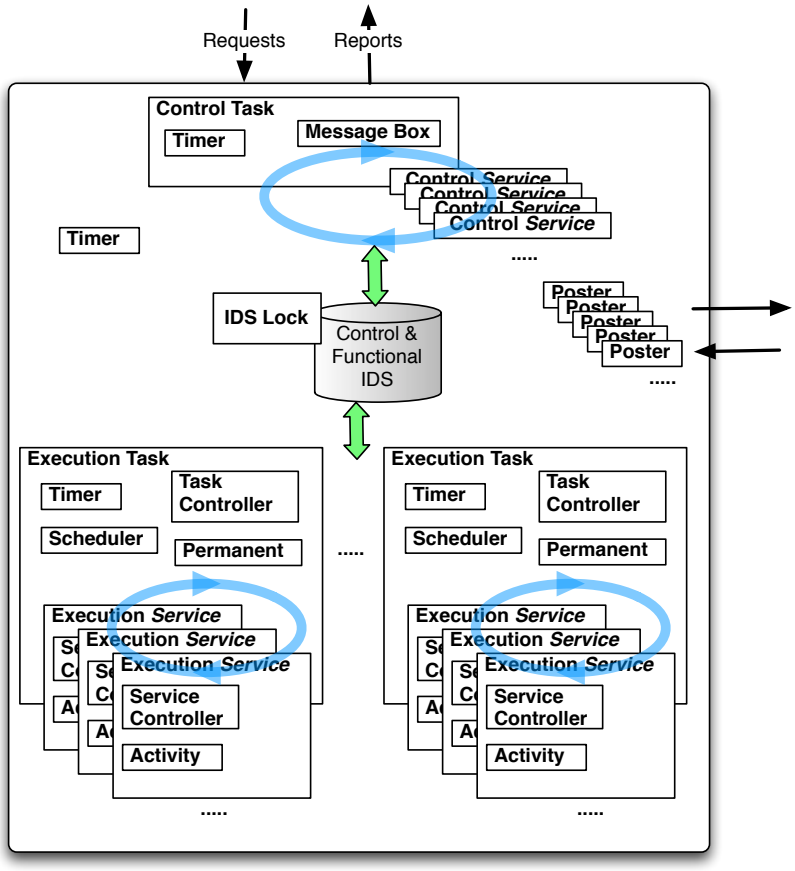


Figure 1: A GenoM module functional organization and its componentization prior to BIP modeling.

This functional level shows the versatility of the GenoM tool, in particular, how a complex functional level could be built with it.

Each GenoM module instance provides specific *services*, which can be invoked by *requests* sent by the higher (decisional) level according to tasks that need to be achieved. These *services* are implemented through the transitions of an automaton that are linked to particular elementary (C/C++) code, called *codels*, which are executed during the transitions. Each GenoM module has a *control task* (which, among other things, handles *requests* and *reports*) and can have multiple user defined *execution tasks*—with different scheduling periods and priorities—in charge of executing particular *services* (in most implementations, each *execution task* is

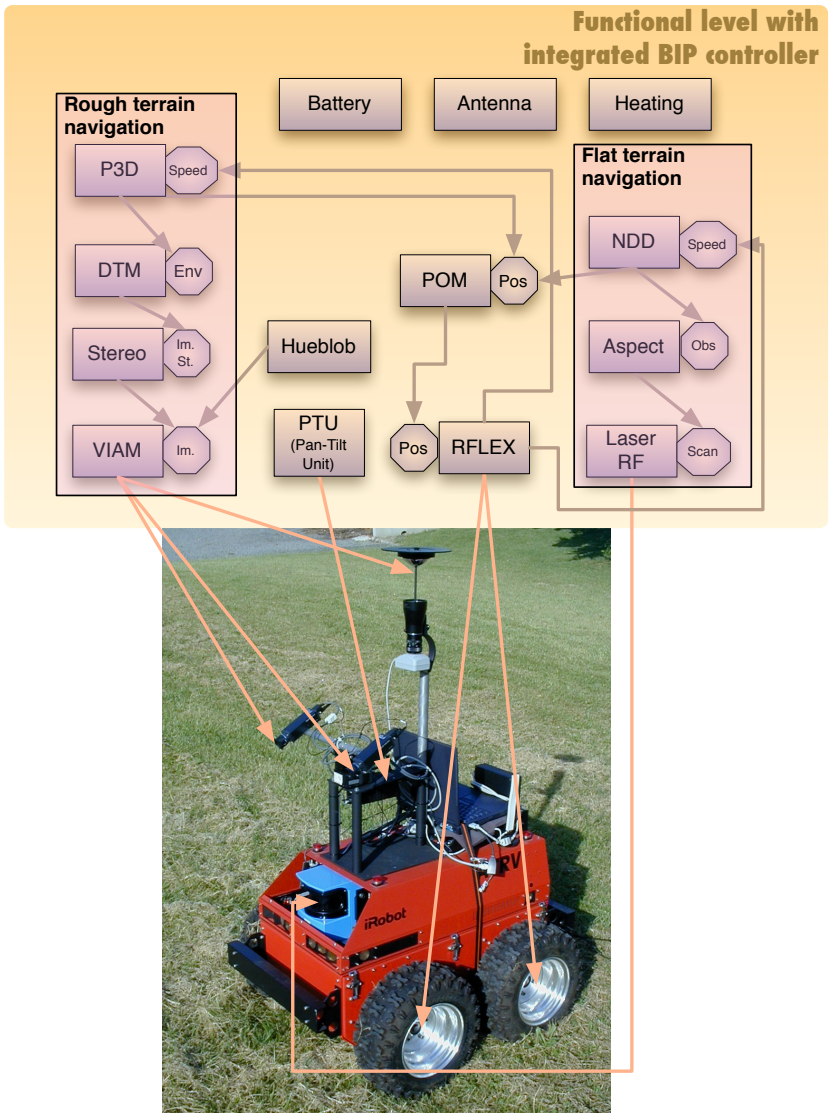


Figure 2: The functional modules of the Dala rover.

a POSIX thread). Upon completion, the *services* return a *report* to the caller. Note that *codels* can be interrupted by the underlying operating system, but all the *codels* of *services* executing in a particular *execution task* are executed one after another (they are in the same thread). Each active *service* is given a slice of the CPU in sequence and executes one *codel* (i.e., performs one transition of its au-

tomaton). A module may also export *posters* containing "shared" data for others (modules or the decisional level) to use.

One of the main differences between GenoM and similar tools is that it not only clearly defines for each module an "external" API to launch *services* and access data through *posters*, but it also enforces a very clear and strict internal organization and behavior of the module (see Figure 1).

3.2. The BIP *Framework*

BIP (Basu et al., 2006) 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 **B**ehavior, the intermediate layer includes a set of connectors describing the **I**nteractions between transitions of the behavior, and the upper layer is a set of **P**riority rules describing scheduling policies for interactions of the layer underneath. Such a layering offers a clear separation between behavior and structure. Second, BIP uses a parameterized composition operator on programs. The product of two programs is the composition of their three corresponding layers separately. Parameters are used to define the interactions as well as new priority rules between the parallel programs (Sifakis, 2005). The use of such a composition operator allows incremental construction, i.e., obtaining a parallel program by successive composition of other programs. Third, for structuring interactions, BIP provides powerful mechanisms including strong synchronization and weak synchronization. A strong synchronization is a rendez-vous, that is, all the interacting components are needed for synchronizing. Weak synchronizations denote broadcasts in which the presence of only one component (called a trigger) is required.

The 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 $P = \{p_1 \dots p_n\}$, 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. Its execution modifies the local data according to function $f_p : V \to V$. A transition is possible if the guard (boolean condition on V) g_p is true and some interaction including port p is offered.

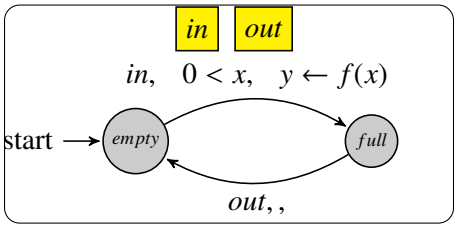


Figure 3: A simple BIP atomic component.

Figure 3 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 micro-step is empty. 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 BIP *Engine*

The BIP engine is a tool to execute online a BIP model. It 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 an 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 execution.

D-Finder : The Verification Tool

D-Finder is a tool to analyze offline the BIP model. It implements a compositional (Bensalem et al., 2008a) and incremental methodology (Bensalem et al., 2010b) for the verification of component-based systems described in the BIP language (Basu et al., 2006). D-Finder is mainly used to check safety properties, particularly deadlock-freedom, of composite components. In other words, D-Finder is used for analyzing the *interactions* between BIP components, not for analyzing the elementary C/C++ code stored inside *codels*.[3] D-Finder applies the compositional verification method proposed in (Bensalem et al., 2008a, 2010b) where 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, and interaction invariants express global synchronization constraints between atomic components.

3.3. The GenoM/BIP *framework*

Our integration of GenoM and BIP involved first writing the complete BIP models of all the generic components of the GenoM generic module. Then, in the same way we used the standard GenoM to build a specific C/C++ module (i.e., for a given set of *services*, *codels*, and *posters*), and we built a specific BIP module by reassembling the BIP models of the generic components together with the associated *codels*. This allowed us to automatically synthesize in a bottom up approach, a complete robot functional level model which is correct-by-construction. Hence, we can view the resulting GenoM/BIP model obtained as a large fine grained transition system composed of all the *codels* of the application. On one hand, this model can be run *online* on-board the real robot, thanks to the BIP engine which controls the proper execution of the model, and on the other hand the model can be checked *offline* for properties such as deadlock freedom using verification tools such as D-Finder. Moreover, our integration allows "safety constraints"–those that ensure the functional level will behave in a safe/desired manner–to be modeled and included on top of the automatically generated model, which are then

[3] There are seperate tools provided in the BIP toolchain for such static analysis.

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. Of course, decisional components may also rely on formal methods to prevent such bugs from occurring. Still, in many implementations, there is a gap between the high level planning model and the functional level model.[4] This gap is usually filled by a supervisor/execution controller using "hand-written" procedures (Ingrand et al. (1996)) or state automata (Bohren and Cousins (2010)) responsible for refining and controlling high level parallel action execution.

Note that GenoM modules may have a life cycle and may evolve as the developer changes and improves its implementation. If the changes are only made to *codels*, the BIP model remains the same and need not be generated again. Any other changes (e.g., to request definitions, posters, and tasks) will require regenerating the BIP model (which takes negligible time) and checking again with D-Finder offline.

4. A Language for Specifying Constraints

A criticism made by our robotics colleagues is that the BIP language is, from the perspective of robot developers, too low level to model the type of constraints that one would usually want to specify (e.g., prevent the robot from starting before being properly initialized, and avoiding a move while drilling). Indeed, BIP constraints are expressed over BIP component states, variables and interactions.

In this section, we introduce a high-level language which gives the designer of the functional level the ability to specify certain constraints on how the functional level services should be used. A constraint is specified between two services belonging to a single module, or between two services belonging to two different modules. Such constraints are important to maintain the safe and desirable execution of services. When specifying the set of constraints, the designer can refer to the well defined interface that each functional level module provides.

Constraints in our high level language are specified in plain text, which are then automatically converted into BIP connectors. The connectors ensure that the corresponding constraints are not violated in the BIP functional level. To unambiguously describe the types of constraints we want to enforce on functional levels of robotic architectures, we use a formal framework, namely, the Event

[4] As of today, we do not know of any planner able to produce a plan at the level of detail that one needs to directly drive functional modules such as the ones presented above.

Calculus (Kowalski and Sergot (1986); Shanahan (2000)), to which perhaps the closest related formalism is Allen's Temporal Logic (Allen, 1983) (see Section 6). We choose the former because it, unlike the latter, provides mechanisms to check whether a condition holds at a given point in time, and to check whether an event has occurred at some point in time.[5]

An event $r(\vec{t})$ of the Event Calculus, where $\vec{t}$ is a vector of terms, represents a GenoM request. Hence, we use the term "event" and "request" interchangeably. An example of an event is *goto(10,20)*, where 10 and 20 are respectively x and y coordinates. An event *type* is denoted by $r(\vec{x})$, where $\vec{x}$ is a vector of distinct variables, or in this paper simply as r (or r_i, for some i) when the vector is not important. An example of an event type is *goto(x,y)*. For convenience, we assume that event types are unique across GenoM modules, i.e., that no event type of a GenoM module is equivalent (up to variable renaming) to an event type of some other GenoM module. Finally, in addition to the standard construct $\mathsf{Happens}(r_1, t_1, t_2)$ of the Event Calculus, which denotes that r_1 has executed to completion between timepoints t_1 and t_2, in this paper we also use construct $\mathsf{Happens}^s(r_1, t_1, t_2)$ to denote that r_1 has *successfully* executed between those two timepoints.

In later definitions we use the notion of a *substitution*, which is is defined in the usual way as follows. A *substitution* θ is a finite set of the form $\{x_1/\tau_1, \ldots, x_n/\tau_n\}$, where $x_1, \ldots, x_n$ are distinct variables, and each τ_i is a term such that $\tau_i \neq x_i$. We say that θ is a *ground substitution* if $\tau_1, \ldots, \tau_n$ are ground terms. If $r(\vec{t})$ is an event and $\theta = \{x_1/\tau_1, \ldots, x_n/\tau_n\}$ is a set of substitutions, we use $r(\vec{t})\theta$ in the usual way to denote the expression obtained from $r(\vec{t})$ by simultaneously replacing each occurrence of x_i in $r(\vec{t})$ with τ_i, for all $i \in \{1, \ldots, n\}$.

The first constraint is read as "not r." It guarantees that request r is never executed. More precisely, the constraint holds if and only if there is no point in time such that r has been executed. Note that this does not mean that the request is never sent from some executive (e.g., PRS)—just that it is not allowed to execute. Since GenoM modules are meant to be reusable components, such a constraint is natural when a GenoM module is used on a new hardware platform, and a request of the module is associated with some code that is no longer deemed appropriate, or when the module is used with a different set of modules and one of its requests is no longer necessary due to the existence of a request (in some other module) with similar functionality. For example, a CPU intensive request, such as one that performs complex calculations, may be acceptable on a robot with a powerful

[5]There may well be other formalisms expressive enough for defining our constraint language.

hardware platform, but not on a smaller robot with a basic hardware platform. Formally, the "not r" constraint is defined using the Event Calculus as follows.

$$!r \stackrel{\text{def}}{=} \forall t_1, t_2, \theta, \neg\mathsf{Happens}(r\theta, t_1, t_2).$$

The second constraint is read as "not both r_1 and r_2." It guarantees that a point is never reached where both r_1 and r_2 have been executed, although it may be the case that r_1 has been executed alone, r_2 has been executed alone, or that neither r_1 nor r_2 have been executed. Intuitively, this constraint corresponds to the *NAND* logical operator. Formally, we have the following definition.

$$(r_1 \uparrow r_2) \stackrel{\text{def}}{=} \forall t_1, t_2, t_3, t_4, \theta_1, \theta_2, \neg[\mathsf{Happens}(r_1\theta_1, t_1, t_2) \wedge \mathsf{Happens}(r_2\theta_2, t_3, t_4)].$$

The third constraint is read as "r_1 before r_2." It guarantees that request r_2 is executed only after the successful execution of request r_1. More precisely, the constraint holds if and only if whenever both r_1 and r_2 have been executed, with r_1 having been successfully executed, the end time of r_1 is before the start time of r_2. This constraint is similar to the "before" constraint found in Allen interval temporal logic (Allen, 1983), with the exception that the latter does not distinguish between the success and failure of actions. The formal definition is given below.

$$\begin{aligned}(r_1 < r_2) \stackrel{\text{def}}{=} \\ \forall\theta_1, t_1, t_2\Big[&\mathsf{Happens}(r_2\theta_1, t_1, t_2) \Rightarrow \\ &\exists\theta_2\ \mathsf{Happens}^s(r_1\theta_2, t_3, t_4) \wedge \\ &\forall\theta_2, t_3, t_4[\mathsf{Happens}^s(r_1\theta_2, t_3, t_4) \Rightarrow t_4 < t_1]\Big].\end{aligned}$$

The fourth constraint is read as "f holds before r." This constraint guarantees that fluent f holds at some point in time before r starts executing. Note, however, that f may be false immediately before the time that r starts executing. Nonetheless, this constraint is useful when a condition must hold at some point in history before an event—although not necessarily immediately before the event—or when it is a feature of the domain that a condition will never become false after it becomes true. In our experimental setup, for example, while the event to take a picture of an interesting rock using high resolution cameras should happen only after such a rock is observed via the panoramic camera, the event does not need to happen immediately after the rock is observed. The formal definition for the

mentioned constraint is as follows.

$$(f < r) \stackrel{\text{def}}{=} \\ \forall t_1, t_2, \theta[\mathsf{Happens}(r\theta, t_1, t_2) \Rightarrow \mathsf{HoldsAt}(f, t_3) \wedge t_3 < t_1].$$

It is worth noting that since there are usually multiple components executing in parallel, and while reading a value stored in a poster (e.g., the location of the rover) belonging to a different component the value might change, we could not find an easy way to encode the constraint "f holds *immediately* before r."

The fifth constraint is read as "r_1 does not overlap r_2." It guarantees that there is no point in time in which both r_1 and r_2 are executing. More precisely, the constraint holds if and only if whenever both r_1 and r_2 have been executed, r_1 finishes before r_2 starts, or r_2 finishes before r_1 starts. Such a constraint is useful to guarantee the mutual exclusion between the executions of two requests.

$$(r_1 \nparallel r_2) \stackrel{\text{def}}{=} \\ \forall t_1, t_2, t_3, t_4, \theta_1, \theta_2[\mathsf{Happens}(r_1\theta_1, t_1, t_2) \wedge \mathsf{Happens}(r_2\theta_2, t_3, t_4) \\ \Rightarrow t_2 < t_3 \vee t_4 < t_1].$$

To illustrate how some of the mentioned constraints are represented as BIP connectors, we show below an example of the textual representation and the BIP connectors for the above "does not overlap" constraint. In our rover, there is a requirement that no communication should occur with an orbiter while the rover is moving, and vice versa, to ensure that communication is not disrupted. Then, the textual representation of this constraint is the following.[6]

antenna.Communicate **NOT-OVERLAPS** *rflex.Move*
ELSE *CANNOT-COMM-AND-MOVE*

This translates to the BIP connectors shown below for the *antenna* and *PTU* GenoM modules in Figure 2 (Bensalem et al., 2011). In what follows, *MA* = *MoveAbort*, *CAIS* = *CommunicateAbortIncompatibleServices*, *CT* = *Communicate Trigger*, and *MS* = *MoveStatus*. We only show the first two connectors; the other two are analogous.

connector *AllowCommIfNotMoving*(*antenna.CT*, *rflex.MS*)
define [*antenna.CT*, *rflex.MS*]

[6] We simplify here for clarity. Actually, there are two versions of this constraint: one yields BIP connectors for aborting the executing request to let the new one execute, and the other yields BIP connectors for rejecting the new request so as to not overlap with the one that is already executing.

on *antenna.CT*, *rflex.MS*
provided ¬*rflex.MS.active*
do {}

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

In the above connectors, an interaction involving the *CT* port (which leads to the starting of the *Communicate* service's execution) is possible only after aborting the *Move* service of the *rflex* module. Moreover, the aborted request will return with the *CANNOT-COMM-AND-MOVE* error message.

We now continue with the fourth and final constraint, which is read as "r only while f." It guarantees that r continues to execute only while fluent f is true. The constraint is formally defined as shown below.

$$(f \sqsubseteq r) \stackrel{\text{def}}{=} \\ \forall t_1, t_2, \theta[\mathsf{Happens}(r\theta, t_1, t_2) \Rightarrow \forall t \in \{t_1, \ldots, t_2\}, \mathsf{HoldsAt}(f, t)].$$

Note that this is the "ideal" meaning of this constraint—it is stronger than what we can guarantee in practice. This is because in BIP, we cannot abort request r *as soon as* f becomes false. At the point at which f becomes false, r might be in the middle of executing a non-interruptible piece of code (e.g., an *exec* codel), resulting in the abort having to wait until the piece of code finishes executing. Moreover, the BIP engine will only detect that f has changed at the next iteration of its main loop. Consequently, some time may have elapsed since f became false. Other than this constraint, all the other constraints mentioned can be expressed using BIP connectors.

As an example of the last constraint described, consider the following connector, which prevents poster data produced by certain modules from being used if the data is 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. In what follows, the *PosterAge* variable keeps track of the amount of time that has elapsed since the last time the associated *Poster* component was written to. Then, in our textual

constraint language the constraint can be specified as follows.

rflex.Move **WHILE** *ndd.RflexPoster.read.PosterAge > 10*
ELSE *NDD-POSTER-NOT-FRESH*

This translates to the following connector.

connector *AbtMoveIfPstrNotFresh*(*rflex.MA*, *ndd.RflexPoster.read*)
define [*rflex.MA*, *ndd.RflexPoster.read*]
on *rflex.MA*, *ndd.RflexPoster.read*
provided *ndd.RflexPoster.read.PosterAge* > 10
do {*rflex.MA.rep* ← *NDD-POSTER-NOT-FRESH*}

We are currently working on proving the soundness of the constraint language with respect to the definitions above—i.e., that the connectors resulting from the high-level constraints between requests makes them behave in the manner defined by the above definitions—by firstly defining a set of connector types per high-level constraint and then showing how each of these sets makes the underlying BIP components behave in the manner defined.

As for completeness, the constraint language is not complete in the sense that more behaviour patterns—albeit not those that we consider to be useful—can be expressed between $\mathsf{G^{en}_{o}M}$ requests using BIP connectors than the patterns expressible using our high-level constraint language. For example, we could write BIP connectors that try to make two given requests start executing at the same point in time (in line with the *starts* constraint in Allen interval temporal logic) by synchronizing the "trigger" ports associated with the requests—which will not necessarily result in the two requests' associated *codels* starting to execute concurrently because that would depend on how they are scheduled by the operating system. Hence, we do not capture such behaviour patterns (i.e., those that provide no guarantees) in our constraint language.

Of course, many BIP (possibly "low level") connectors that are used to define interactions between components don't yield any meaningful constraints at the level of abstraction that interests us, i.e., behaviour patterns between $\mathsf{G^{en}_{o}M}$ requests. For example, while one could write BIP connectors to allow a request to start executing only when another "starts to end," i.e., when the latter starts executing its "end" codel (the codel used to perform any outstanding tasks before the service is terminated), this behaviour pattern is not meaningful at the higher level of abstraction.

5. The Real-Time BIP Framework

In the previous version of the GenoM/BIP approach (Bensalem et al. (2009)), time was taken into account with logical ticks provided by the BIP engine. Although we were still able to model real-time properties using timers and BIP automata with transitions executing C "sleep" actions, this was clearly not enough when it came to providing and controlling a real-timed model of the system. This section discusses a real-time BIP framework, including an extension to the BIP model which supports, similarly to timed automata (Alur and Dill, 1994), the explicit representation of time and clocks, and an extension of the BIP engine.

In this section, we explain how the approach has been improved using real-time BIP, which consists of a real-time extension of the BIP language for modeling real-time systems, and a real-time BIP engine used for its execution (Abdellatif et al., 2010). The real-time BIP engine performs the computation of schedules meeting the timing constraints of the application, depending on the actual time provided by the real-time clock of the platform. The engine directly implements the semantics of the language.

5.1. Motivations: Safely handle real-time features

The autonomous rover Dala uses an initial BIP/GenoM component-based design approach (Bensalem et al., 2009) for specifying and implementing the functional level of the robot. In the original BIP framework, component behaviors were automata extended with data. There was no explicit notion of time, that is, conditions (guards) for enabledness of interactions between components only depended on the values of components' variables. It was possible to enforce timing constraints directly in the components by calling primitives of the platform on transitions. However, we avoided this method because the timing behavior of the model was untrackable. In (Bensalem et al., 2010a) they prefer using global tick synchronizations between the components, that is, time progresses synchronously. In this case time is made explicit in the model, which allows the use of verification techniques. However, using such tick synchronizations is inefficient due to the periodic execution of ticks.

In general, each module instance provides specific *services*, which can be invoked by *requests* sent by the decisional level according to tasks that need to be achieved (see Section 3). A module has an *execution task* with different scheduling periods and priorities, that is in charge of executing particular user defined services. It triggers periodically services for launching and executing associated

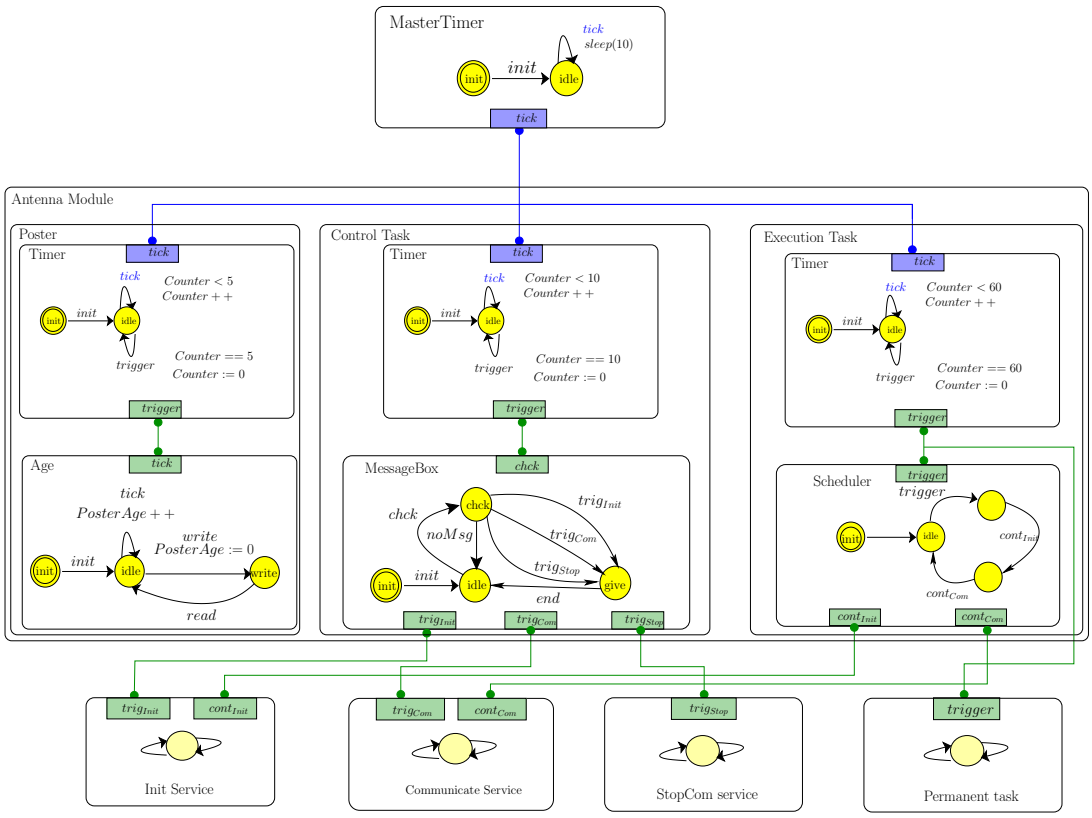


Figure 4: Antenna module implementing timing constraints using ticks.

activities and upon completion the *services* return a *report* to the caller. A *control task*, which among other things, handles *requests* and *reports*, is responsible for setting and returning variable values. A *poster* is a module that may export *posters* containing "shared" data for others (modules or the decisional level) to use.

Time is taken into account with logical ticks provided by the BIP engine. Thus, real-time properties are modeled using *timers* and BIP automata with transitions executing C "sleep" actions. In general, a global *timer* component is responsible for executing the C "sleep" actions and synchronizes all the timer components of the modules that are used to trigger the execution of periodic tasks. In the following subsection, we give as an example, the Antenna module's general structure.

The Antenna module is responsible for the communication with an orbiter and it is structured as follows (see Figure 4).

A set of services. The *Init service* fixes the time window for the communication between the application and the orbiter at initialization. The *Communication service* starts the communication with the orbiter in a bounded duration. The *StopCom service* terminates the on-going communication between the application and the orbiter.

***Timer* components.** *MasterTimer* ensures that there are at least 10 ms between two consecutive synchronizations of the components *Timer*. This is achieved by calling sleep primitives of the platform in *MasterTimer* when executing action *tick*. *Timer* components trigger the functional components at the fixed periods (i.e., ticks) given as parameters. Periodic execution of *Timer* is enforced by a guard involving an integer variable *Counter* incremented at each *tick* execution.

Functional components. They are triggered by the *Timer* components. Component *Age* measures the freshness of the poster at a period of 5 ticks (50 ms). Component *MessageBox* checks the presence of requests using a period of 10 ticks (100 ms). Component *Scheduler* executes activities based on a period of 60 ticks (600 ms).

However, this translation is inefficient when the period of the execution of the Tick connector is small compared to the actual period of activation of the components, because the engine wakes up frequently to count time even if there are no interactions to execute. Moreover, this approach requires for execution times of interactions to be bounded by this period, which is a very strong assumption. Finally, since Tick strongly synchronizes all components in all states, such models can easily deadlock. The real-time BIP engine avoids this translation by directly computing enabled interactions and their associated time constraints.

5.2. Overview of the real-time BIP *Engine*

In real-time BIP, the timing constraints of a component are expressed by using a timed automaton. Timed automata (Alur and Dill, 1994) are automata extended with clocks, which are variables valued either as integer or as real that increase synchronously and are used to measure the passing of time. Clocks can be reset and tested against lower and upper bounds on transitions. We also consider three types of urgency for transitions: lazy (i.e., no urgency), delayable (i.e., urgent just before they become disabled) and eager (i.e., urgent whenever they are enabled). We made also extensions in order to model system communications with the external environment by using Input/Output automata. Internal actions and outputs are triggered by the application, whereas inputs are triggered by the environment. Timing constraints correspond to user-requirements such as deadlines, periodicity and occurrence of inputs.

Figure 5 is an example of a real-time BIP component. It represents a cyclic execution of a system that receives an input *in* from the environment, performs

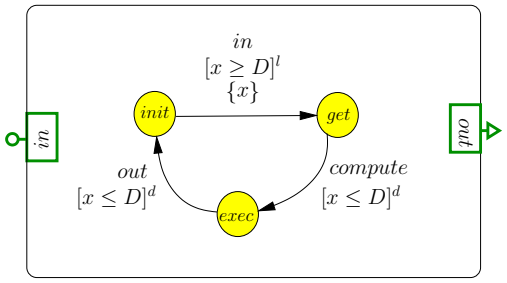


Figure 5: A timed BIP atomic component.

an internal computation and sends an output *out* to the environment. A clock x is used to measure the time elapsed since the last occurrence of *in* (i.e., x is reset by *in*). Both *compute* and *out* must be done before x reaches the deadline D.

The Real-time BIP framework relies on a (centralized and for now single-threaded) real-time engine that computes enabled interactions online (see Figure 6). Our implementation is based on the presence of a centralized notion of time given by a real-time clock provided by the platform. Given a state of the system, transitions of atomic components issued from this state associate timing constraints to ports. Before checking enabledness of interactions, the real-time engine expresses all timing constraints involving local clocks in the context of a single global clock t that measures the absolute time elapsed. Timing constraints of interactions are computed by performing the conjunction between timing constraints of the ports. We also associate urgencies to interactions by considering the maximal urgency of their ports (lazy < delayable < eager). Before scheduling an interaction, the engine updates the global clock t, completed by a check for the presence of a deadline miss (i.e., violation of the urgency of a transition). The real-time engine can be used either for simulating a model (t is updated w.r.t. given execution times) or for its real-time execution (t is updated w.r.t. a platform clock). It is also able to communicate with external devices such as sensors and actuators through an Event Handler. It maps the inputs and outputs specified in the model with the physical events. The real-time Engine computes schedules of actions with respect to timing constraints. When a deadline or an input is missed, it stops the execution and reports the deadlock.

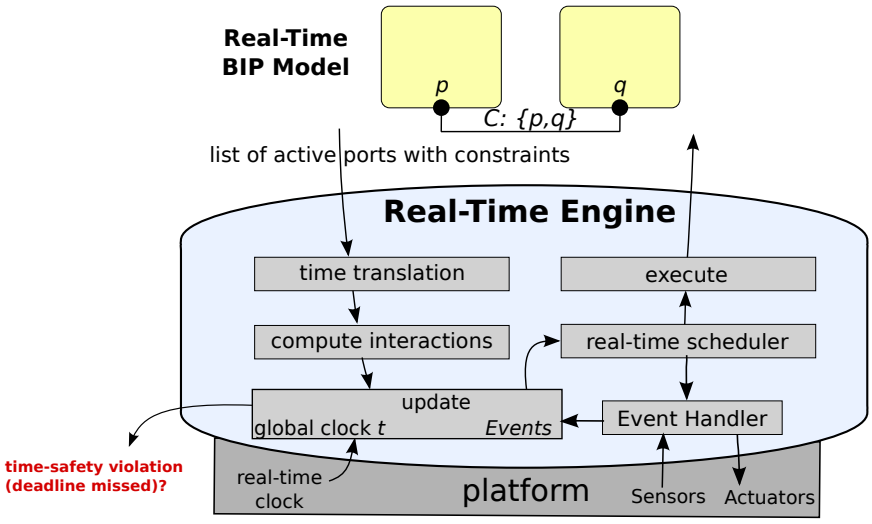


Figure 6: The real-time BIP engine.

5.3. Modeling modules using real-time BIP

In this subsection, we show how we modeled the Antenna module using real-time BIP. Figure 7 is the resulting representation of the Antenna module. For this purpose, we introduced the following.

Clocks and timing constraints. Clock *Ageclk* in *Age* component measures the freshness of the poster. That is, whenever we write a new piece of data, clock *Ageclk* is reset to zero. A timing constraint over clock *Ageclk* can prevent the reading of a poster if the data is older than or equal to a giving value. In our case, the poster is read for $Ageclk < 50ms$. Clock *Pclk* in *Scheduler* component is used to enforce a period of 600 ms to launch a task. The corresponding timing constraint over clock *Pclk* is $Pclk = 600$.

Inputs and outputs. In the first implementation, the communication with the decisional level and the Antenna module is achieved by using a dedicated shared buffer. The decisional level directly writes requests in the buffer, and Antenna periodically reads the buffer using component *MessageBox* to check their presence. The chosen period is 100 ms. Antenna also sends reports when executing the internal action *report* of *MessageBox*. We introduce the notion of Input and Output ports to communicate with the decisional level. We introduce the input port *Request* that replaces the active wait used by the first implementation. The output port *Report* is introduced to send reports to the environment after the treatment of a request.

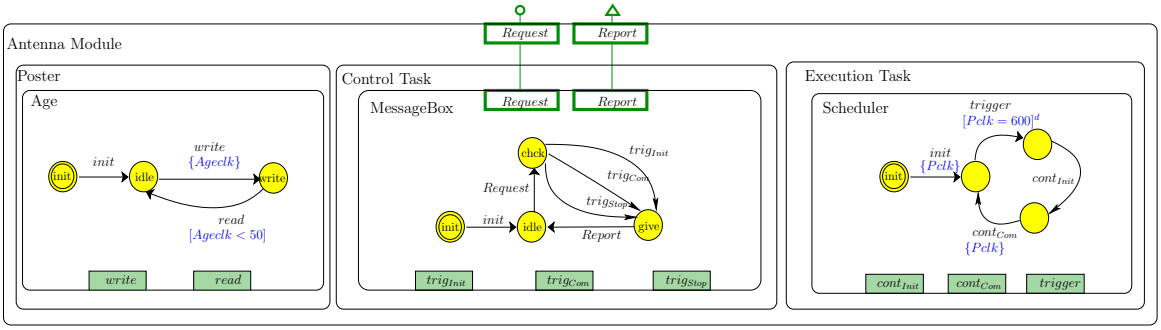


Figure 7: Antenna module implementing timing constraints using clocks.

6. Temporal Plan Execution Control

The decisional level of an autonomous system usually involves temporal planners (Ingrand et al., 2007) that basically choose variable time instants for the execution of actions of a system, e.g., starting or terminating a task. The goal of the planner is to find a valid plan, that is, a list of actions meeting user-defined constraints that can be expressed using various formalisms. Planners can also seek for efficiency by optimizing parameters such as latency, throughput, energy, and memory. The robotic systems we deploy rely on a higher level temporal planner and a plan execution controller using a Timeline-based planning technique. Allen's interval algebra, also called Allen's temporal logic (ATL) (Allen, 1983) is one of the best established formalisms for temporal reasoning since it expresses constraints on plans. The constraints are expressed as boolean formulae over atomic propositions. Validation and correctness of complex plans for systems with decisional autonomy is highly desirable, if not critical in many applications. Since ATL is the logic of planning, an automated translation from ATL to BIP components enables to use rigorous design and implementation techniques and tools in a domain lacking them.

In this section, we introduce the ATL to BIP translation mechanism. We explain how to build correct models for plans, that is, how Timelines are modeled and how to manage the constraints between the different Timelines to build a correct plan. We also give an example of a DALA robot plan model and execution using the real-time BIP framework.

6.1. Translation of an ATL fomula into BIP components

Allen interval temporal logic (Allen, 1983) is a widely used formalism in the temporal planning community for expressing constraints on plans. It expresses constraints as boolean formulae over atomic propositions on time intervals $I = [s_I, t_I]$, where s_I is the start time of I and t_I its terminating time ($s_I < t_I$). Considered atomic propositions include but are not limited to:

1. I *equals* J, which means intervals I and J coincide (i.e., $s_I = s_J$ and $t_I = t_J$);
2. I *before* J, which means I terminates before J starts (i.e., $t_I < s_J$);
3. I *overlaps* J, which means J starts during I (i.e., $s_I < s_J < t_I < t_J$);
4. I *meets* J, which means J starts when I terminates (i.e., $t_I = s_J$);
5. I *during* J, which means I is included in J (i.e., $s_J < s_i$ and $t_i < t_J$); and
6. I *starts* (resp. *finishes*) J, which means I and J start (resp. *finish*) at the same time (i.e., $s_I = s_J$, resp. $t_I = t_J$).

Moreover, these thirteen relations (six of the above have a symmetrical one) can be extended with numerical (duration) information if needed. It has been shown that Allen interval logic formulae can be translated into timed automata (Rosu and Bensalem, 2006). Their translation into real-time BIP models can lead to an executable BIP model which can also be used jointly with the model of the functional level to check properties that encompass both the decisional and functional levels. The idea is to use one timed automaton (i.e., one component) for modeling each interval. Interactions and priorities—a.k.a. the glue—in BIP offers also an elegant language for expressing constraints between intervals used to describe coincidence of actions.

Translating Allen intervals into BIP *components*

We consider an interval as an execution action, thus it has a starting time and a finishing time. Figure 8 (b) is a representation of the action as an atomic component. It is composed of a set of ports {**begin**, **executing**, **finish**, **no-executing**}, port **begin** corresponding to the beginning of the execution, port **executing** corresponding to the execution, port **finish** corresponding to the end of execution and port **no-executing** corresponding to the time after execution of the interval. Ports **begin** and **finish** correspond to particular time instants of executions that are triggered only once in the BIP component. Ports **executing** and **no-executing** correspond to time intervals that can be triggered as long as we are in the corresponding time intervals. Clock x is reset at initialization. It is used to ensure the beginning of the action within a starting interval $[s_{lb}, s_{ub}]$ (where s_{lb} is the lower

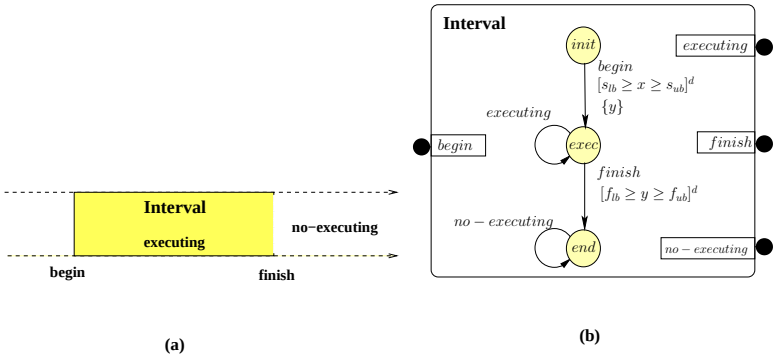


Figure 8: Modeling an interval (a) as a BIP atomic component (b).

bound and s_{ub} is the upper bound), by using constraint $s_{lb} < x < s_{ub}$ with a delayable urgency. Clock y is reset when the action execution begins. It is used to measure the execution time of the action to ensure its termination within a finishing interval $[t_{lb}, t_{ub}]$ (where t_{lb} is the lower bound and t_{ub} is the upper bound), by using constraint $t_{lb} < y < t_{ub}$ with a delayable urgency. Each transition is labeled by an exported port. Exported ports are used as an interface to enable the synchronization with other components.

Translating Allen constraints into BIP *connectors*

Boolean combinations of atomic propositions can be efficiently derived using existing work that establishes the correspondence between boolean formulae and the glue (Bliudze and Sifakis, 2008). Therefore, coincidence of actions (e.g., $t_I = s_J$) can be modeled as a strong synchronization between atomic components using interactions. Each Allen constraint between two intervals can be translated into strong synchronization between transitions of the corresponding atomic components through their exported ports. Ordering of actions (e.g., $t_I < s_J$) can be modeled as a set of priorities. For each Allen constraint we have the corresponding connector type in BIP as follows:

1. *I equals J* synchronizes port begin of I and port begin of J, and synchronizes port finish of I and port finish of J;
2. *I before J* synchronizes port no-executing of I and port begin of J;
3. *I overlaps J* synchronizes port executing of I and port begin of J, and synchronizes port no-executing of I and port finish of J ;

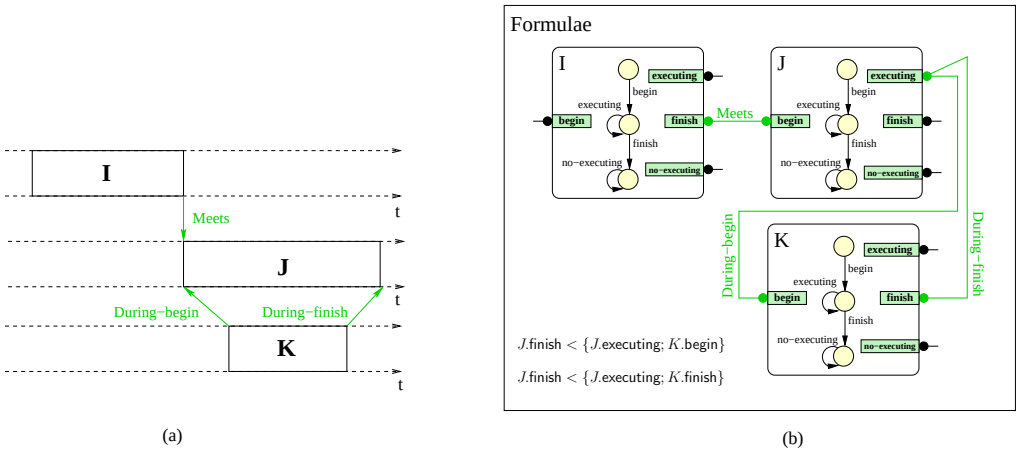


Figure 9: Example of translation: formulae "I meets J and K during J" (a) into a BIP compound component (b).

4. *I meets J* synchronizes port finish of I and port begin of J;
5. *I during J* synchronizes port begin of I and port executing of J, and synchronizes port finish of I and port executing of J; and
6. *I starts J* synchronizes port begin of I and port begin of J.

As an example, we give the translation of the formulae "I meets J and K during J," over 3 three atomic propositions I, J and K, into a BIP compound component (see Figure 9). Propositions are modeled as atomic components and constraints as interactions over those components. The formulae I meets J is modeled by using a strong synchronization between port finish of I and port begin of J. To ensure that K is executed after the start of J, we use strong synchronizations between port executing of J and ports begin and finish of K. We also give a higher priority for the execution of K over the completion of J to enforce the execution of K before the completion of J. Thus, priorities enforce ordering of actions (e.g., $t_I < s_J$).

6.2. An example with a plan of the Dala rover

We have modeled and executed a plan of Dala using the open real-time BIP framework. The Dala robot mission scenario consists of navigating from an initial position to a target position, taking a picture of the place, and going back to the initial position. Each step of the mission requires a strong collaboration between the different modules of the robot. Figure 10 is a graphical representation of the plan. It is composed of six timelines describing the variable states for each module of the robot.

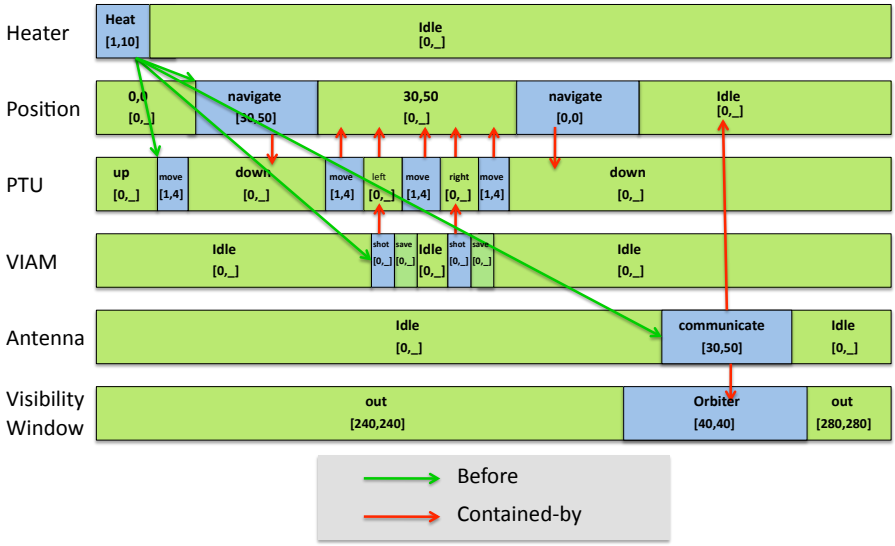
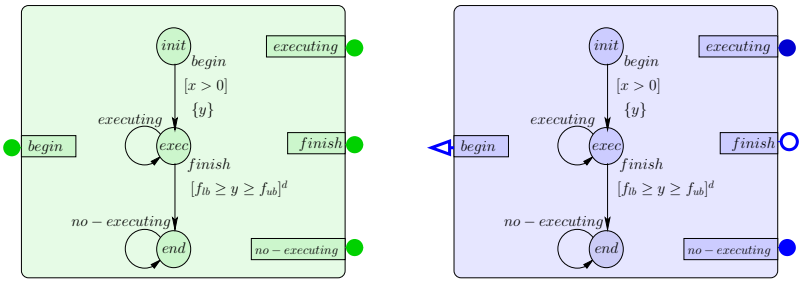


Figure 10: Example of a Dala plan.

- In the Heater Timeline, the initial action has to set the heater to a given value with action *Heat*. It has a timing constraint on its termination in the interval [1, 10]. When it finishes, the *Heater* is at state *idle*.
- In the Position Timeline, the robot is first at position *(0,0)*. It can navigate to any position (x,y) with action *navigate* in the interval [30, 50].
- In the PTU Timeline, the PTU initial position of the robot is *up*. It can move down, left or right with action *move* in the interval [1, 4].
- In the VIAM Timeline, the camera is at state *idle*. It takes a photo with action *shot* then saves it.
- In the Antenna Timeline, the antenna is first at state *idle*. It communicates with an orbiter in order to send the picture with the action *communicate* in the interval [30, 50].
- In the Visibility Window timeline the action *orbiter* is responsible for the communication window visibility for communicating with the orbiter.

Each timeline is modeled as a compound component composed of atomic components representing the actions. The chronological order between these actions is expressed by constraint "Meets." Actions are either uncontrollable ac-



(a) State Action

(b) Uncontrollable Action

Figure 11: Modelling the robot basic actions using BIP.

Uncontrollable Action	Robot Command
Initialization	*InitialiseRobot*
Heat (value)	*HeatRobot* value
navigate (x,y)	*MoveRobot* x y
move (x,y)	*MovePTU* x y
shot (image)	*TakeSciencePic* image

Table 1: Mapping Between the uncontrollable actions and the robot commands

tions, i.e. actions that can send and receive requests from the functional level, or state actions that describe the expected state in which the robot should be. Figure 11 is a representation of the atomic components corresponding to either a state action (a) or uncontrollable action (b). The atomic component representing an uncontrollable action is an open component capable of sending requests and receiving reports from the environment. Thus, its begin port is an **input** to send the corresponding command of the action to the robot (see Table 1 representing the mapping between actions and the robot commands). The finish port is an **output** to receive a notification when the action has finished. With this method, we are able to detect timing constraints violations if the actions take more time than expected. The uncontrollable actions of the plan are *heat*, *navigate*, *move*, *shot* and *communicate*.

Figure 12 represents the BIP representation for timelines Heater and Position of the robot. We express constraints between actions of different timelines by using Allen constraints. Each compound component exports ports that are involved in synchronizations with other components. Port "Heat.no-executing" is exported from port "no-executing" of component "Heat" and port "navigate.begin" is exported from the interaction in which port "begin" of component "navigate" is involved. The constraint "heat before navigate" is modeled by synchronizing those

two ports.

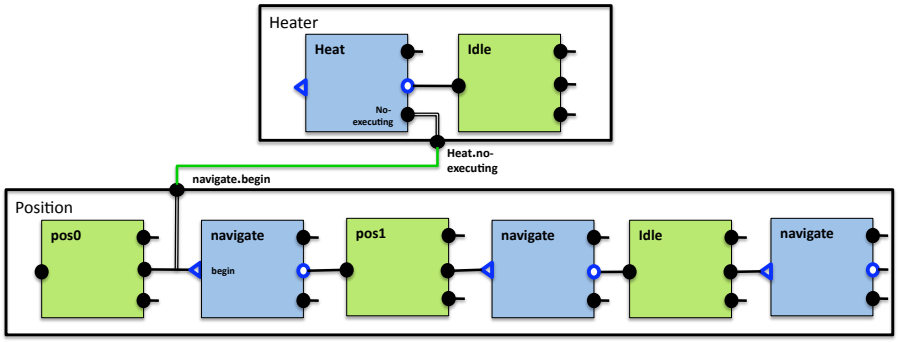


Figure 12: Example of modeling Timelines using BIP.

We note that some ports can be involved in several synchronizations (constraints) and thus, it is necessary to merge the interactions in which they are involved into a single connector. The transformation of Allen Temporal Logic into BIP models is tedious and error prone due to the merging mechanism. For this purpose, we extend the BIP language in order to express the Allen Temporal Logic between components by introducing keywords for Allen constraints. From this new language, we automatically generate the corresponding connectors by a correct model to model transformation tool.

We give the new syntax for the description of the Timeline Heater in Figure 13 and the model of the whole plan in Figure 14. Each declaration of an Allen definition is identified by the keyword allen. Then, we define the Allen relation between an action1 and action2 with the appropriate allen-constraint (meets, before, overlaps, starts, ends, during, equals).

7. Experiments and Results

We have proposed a novel approach to developing functional levels of robots, which involves the synthesis, from a $G^{en}_{o}M$ model, of a correct-by-construction BIP controller that encodes and enforces designer-supplied inter-module and intra-module safety properties. We found that the time taken to generate a BIP model from a given $G^{en}_{o}M$ model is negligible.

```
Compound type Heater
   component uncontrollable-action Heat [1,10]
   component state-action idle
   allen Heat meets idle
end
```

Figure 13: BIP syntax for the “Heater“ Timeline.

```
Compound type Plan
   component Heater heater
   component Position position
   component PTU ptu
   component VIAM viam
   component Antenna antenna
   component Visibility-Window visibility-window

   allen heater.heater before position.navigate1
   allen heater.heater before ptu.move1
   allen heater.heater before viam.shot
   allen heater.heater before antenna.communicate
   allen position.navigate1 during ptu.down
   allen position.navigate0-bis during ptu.down-bis
   allen ptu.move1 during position.position1
   allen ptu.move2 during position.position1
   allen ptu.move3 during position.position1
   allen viam.shot1 during ptu.left
   allen viam.shot2 during ptu.right
   allen antenna.communicate during position.idle
   allen antenna.communicate during visibility-window.orbiter
end
```

Figure 14: BIP syntax for the plan.

We were able to run experiments[7] with a complete functional and decisional level on our rover, and to demonstrate via fault injections that the BIP engine successfully stops the rover from reaching undesirable/unsafe situations, and that it reports appropriately to the decisional level. An example of an undesirable situation is where radio communication is attempted while the rover is moving. The rover motion will likely cause the antenna to lose its pointing orientation, and consequently the signal. When we injected a fault at the decisional (PRS) level to send a request to the `Antenna` module to initiate a communication while the robot was moving, the BIP engine successfully stopped the motion and returned an error message indicating that the motion was stopped to allow communication (which is presumably more important) to take place.

7.1. Deadlock-freedom Verification

We have used the D-Finder tool to formally verify our functional level to check for properties such as deadlocks and data freshness. Using D-Finder, we were able to check the deadlock-freedom of all thirteen (single) modules in reasonable amounts of time,[8] even those consisting of thousands of lines of BIP code. We were also able to check that the group of four laser-based navigation modules `RFLEX`, `LaserRF`, `Aspect`, and `NDD` are together deadlock-free.

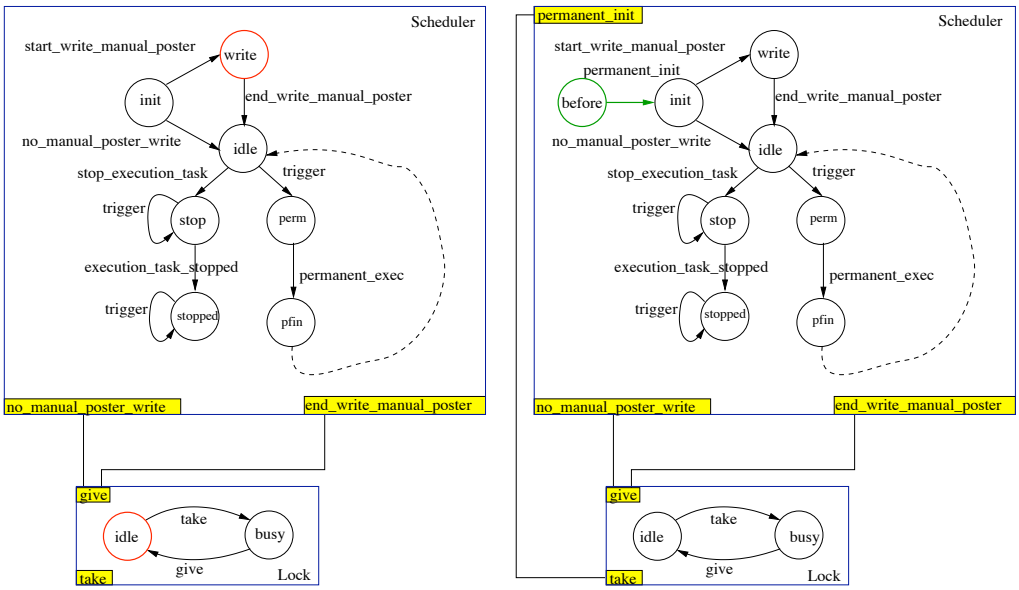
An example of a simple deadlock detected by D-Finder and our solution to it is shown in Figures 15(a) and 15(b). The figures show two subcomponents belonging to all BIP top-level components (e.g., `NDD`), where the *Scheduler* component controls an associated service, and the *Lock* component represents a semaphore to ensure mutual exclusion when multiple services write to the same poster. In this design, the *Scheduler* component should synchronize (via some appropriate port) with the *take* port of the *Lock* component before writing to the poster, and then with the *give* port after writing to the poster. Due to a design flaw, the *Scheduler* component in Figure 15(a) did not do the former, which resulted in a deadlock caused by a synchronization with *give* being impossible.[9]

---

[7] To convince the reader that we have implemented the whole approach on real robots, we have made available a video on the web (`http://db.tt/YhzodGDk`—please download it to see it in full resolution and the proper captions) showing a complete experiment with the Dala rover running the functional level (see Figure 2) in BIP, with various fault injections leading the BIP engine to prevent them and report them to the decisional level (OpenPRS in this case), which takes corrective actions.

[8] We do not have the `POM` module in BIP because its functionality is not very useful for our functional level.

[9] We thank Rongjie Yan for this example.



(a) Deadlock (b) Solution

Figure 15: A deadlock (a) and its solution (b). In both figures the top component is a *Scheduler* and the bottom one is a *Lock*.

One might have noticed that the bug resulting from the above deadlock is caused by a flaw made by the designer of the BIP functional level and not by its user. In addition to detecting such system-level bugs to facilitate convergence towards a "correct" functional level, D-Finder can also help detect user-level bugs caused by subtle logical errors in BIP connectors added manually (without using our high-level constraint language) to the functional level for the purpose of enforcing constraints between associated GenoM requests. Although it was not explicitly distinguished as such a user-level bug in Bensalem et al. (2011), the authors provide a detailed example of a deadlock (and resulting bug) caused by connectors manually added by the user to enforce the "data freshness" constraint, which prevents poster data produced by certain modules from being used if the data is too old. The authors attempt to fix the deadlock with a modified connector, which then has another more subtle logical error, resulting in another deadlock. Basically, the second deadlock was caused by attempting to abort a *Move* service when it had already been aborted, resulting in the service not being in a state in which it is ready to be aborted and blocking certain other interactions. The solution was to let the other interactions proceed even when the *Move* service was not

	real(s)	user(s)	sys(s)	CPU (%)
1st implementation (ticks)	22.6	0.2	0.1	1.32
2nd implementation (real-time Engine)	22.6	0.025	0.04	0.22

Table 2: CPU utilization for Antenna.

ready to be aborted. We refer the reader to Bensalem et al. (2011) for the detailed explanation of this deadlock scenario.

One of the objectives of the high-level language in Section 7.3 is to prevent such bugs resulting from error-prone user-supplied BIP connectors, by automating the process of generating BIP connectors that have a guarantee where they will not introduce deadlocks into an otherwise deadlock-free functional level. While we are in the process of proving this, intuitively, our proof relies on showing that the connectors corresponding to high-level constraints do not disallow transitions from any of the BIP components' states from where transitions would otherwise have been possible.

7.2. Performance of the real-time engine

We conducted experiments to compare the runtime performance of the $\mathsf{G}^{\mathsf{en}}\!\!\%\mathsf{M}$ functional level and the corresponding BIP functional level. Results showed that each BIP module took approximately ten times more CPU time on average than the corresponding $\mathsf{G}^{\mathsf{en}}\!\!\%\mathsf{M}$ module. Likewise, in terms of the usage of the CPUs immediately before the end of the experiment, the $\mathsf{G}^{\mathsf{en}}\!\!\%\mathsf{M}$ functional level used approximately ten times less than the BIP functional level (6.3% versus 52% CPU usage). This additional overhead with BIP comes mainly from the decision making by the BIP engine. In particular, the BIP engine actively computes all the feasible interactions at every cycle, for which the time taken is proportional to the number of interactions in the BIP components.

The real-time engine presented in Section 5 can be used to avoid the computation of Tick synchronizations, and thus reduce the overhead due to the use of BIP. Tick synchronizations are used in untimed BIP models to keep track of the real-time in all components. Timing constraints are modeled in components as boolean conditions involving clocks, that is, integer variables which are increased synchronously at each occurrence of a Tick synchronization. Since the real-time engine directly schedules the interactions at time instants meeting the timing constraints of the model, it completely avoids the need of Tick synchronizations.

We compared the execution of the first implementation of Antenna (i.e., using ticks), and the second implementation (i.e., using the real-time Engine). CPU

utilization is almost 5 times higher with ticks compared to the real-time Engine see Table 2). We also improved the response time of the module to a request (see Figure 16). Note that the generally higher latency of untimed BIP is because, to attend to a newly arrived request, the system has to wait for the next Tick synchronization, which in this experiment happens every 0.1 seconds. We are now working on generalizing these results to the rest of the functional level of the Dala rover application.

The real-time BIP engine is well-suited for executing also the decisional level of robotic systems. In Section 6, we have seen how to model plans in an elegant manner by translating constraints over actions into connectors and priorities. We introduced an example with a plan of Dala that we have successfully executed on the robot. Indeed, the real-time BIP engine becomes a temporal plan execution controller by providing a correct schedule of actions, which communicates with the functional level through an event handler (Abdellatif et al., 2011) that sends the request corresponding to each uncontrollable action and waits for a reply within the time interval corresponding to the termination of the action. The engine detects violations of timing constraints and stops the execution or reports the fault.

Using a BIP model combining plans of the decisional level and the functional level is interesting for increasing the performance of the implementations, since costly communications between these two levels can be expressed as BIP interactions directly handled by the engine in the same process. Indeed, we have seen in Section 5, that we have introduced an input port `Request` and an output port `Report` in order to communicate with the decisional level. In Section 6, we also introduced input ports to send requests to the functional level and output ports to receive reports. The communication between input and output ports of both levels can be handled in the future by a single BIP engine.

7.3. The high-level constraint language

We have used the high-level constraint language presented in Section 4 to obtain and enforce, via the BIP engine, a multitude of low level BIP connectors corresponding to the high-level constraints. At different stages of our experiments on Dala we have specified in total at least 11 different high-level constraints on the functional level, some of which are mentioned in Section 4. These have mapped onto at least 76 BIP connector types and instances on top of those that are a part of the existing BIP model. Each high-level constraint was successfully enforced by the BIP engine at runtime throughout our experiments with the rover.

An aspect on which we intend to improve the current approach is to allow the user to check deadlock freedom on a particular subset of components, or more

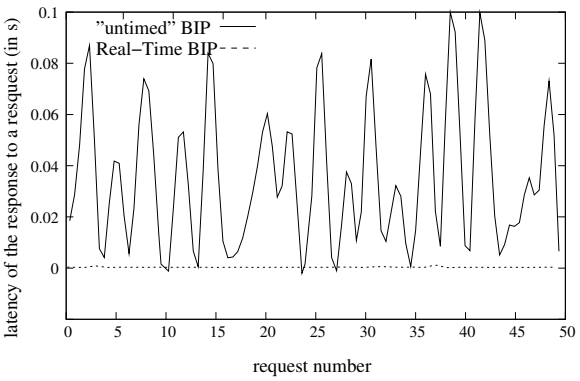


Figure 16: Comparison between real-time BIP and untimed (standard) BIP.

precisely on a particular chain of processing. For example, we may want to focus the deadlock search on the perception/navigation loop, and show that under normal conditions, the navigation algorithm will run properly without reaching a deadlock.

8. Future Work

8.1. Distributed implementations of BIP *models*

So far, our experiments were done on the Dala platform which is a mono-CPU robot. Currently, powerful hardware platforms needed for executing robotic applications are multi-core or many-core platforms. Hence, we needed to investigate a way of distributing the BIP model and the BIP engine over more than one CPU. The application code should be optimally distributed over the platform to take advantage of its computing power. Although distributed systems are widely used nowadays, their implementation is still time-consuming and an error-prone task.

Coordination in BIP is achieved through multi-party interactions (i.e., those across multiple components), and scheduling by using dynamic priorities. Transforming the semantics of (untimed) BIP—which is based on an atomic interaction execution and is defined on a global state model—into a distributed implementation is clearly a nontrivial task. We developed a generic framework allowing the transformation of high-level BIP models into distributed implementations (Bonakdarpour et al., 2010; Jaber, 2010).

Our method involves BIP-to-BIP transformations preserving observational equivalence. We transform multi-party interactions into asynchronous message passing,

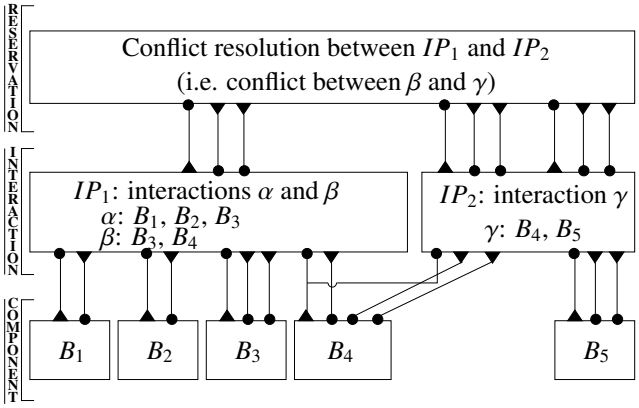


Figure 17: Send/Receive BIP model obtained from BIP-to-BIP transformations.

that is, send/receive primitives. A target *Send/Receive* BIP model is structured in three layers (see Figure 17): *(i)* the *component layer* corresponds to a modified behavior of the components of the original model; *(ii)* the *interaction protocol* consists of a set of components such that each component detects enabledness of a subset of interactions of the original model using partial-state knowledge, and executes them after resolving conflicts (e.g., regarding which interaction to execute when there is more than one involving the same port) either locally or with the help of the third layer; *(iii)* the *reservation protocol* resolves conflicts between components of the interaction protocol layer using committee coordination algorithms such as the token-ring distributed algorithm or the distributed Dining Philosophers algorithm. Notice that an obtained Send/Receive BIP model depends on a user-defined partition of the interactions of its original model, associating subsets of interactions to components of the interaction protocol layer.

We have developed a C++ code generator that performs, given a user-defined mapping of the components of a Send/Receive BIP model, the generation of distributed implementations using communication mechanisms offered by the platform. We have the following backends: Unix processes communicating through TCP sockets, MPI, and threads using semaphores and shared memory. Efficient monolithic code can be produced by merging components using another BIP-to-BIP transformation (Jaber, 2010), according to the mapping of the components.

The method has been fully implemented in a toolset allowing the automatic generation of distributed implementations from BIP models. It is parameterized by the partitioning of interactions, a committee coordination algorithm, and

the mapping of components. The performance of the resulting implementation strongly depends on the choice of these parameters (Jaber, 2010).

We plan to implement this work on some of our most recent robotic platforms (e.g., PR2 (Bohren et al. (2011))).

8.2. GenoM3

The most recent version of GenoM is GenoM3 (Mallet et al., 2010). This is a complete rewrite of GenoM that is template based.[10] One still provides the `.gen` specification and the associated *codels*, but the automatic synthesis of the module is based on templates that get properly instantiated for a particular instance. As a result, one can write templates for a Pocolib[11] version of the module or a ROS Comm version of the module. Similarly, client interface libraries can be generated using this powerful template mechanism. As a result we can for example synthesize ROS GenoM3 modules which can interface with ROS nodes.

This mechanism is perfectly adapted to generating a BIP model of the module. Instead of generating the C/C++ code of the classical module, we can generate the BIP version which implements the same behavior. We plan to write these templates in the near future. Note that this would open our approach to the ROS ecosystem (providing people write the associated `.gen` specifications for their ROS nodes/services/actions).

9. Discussion

Despite a growing interest in developing safe, robust, and verifiable robot software, such development still remains quite disconnected from the use of formal methods. While there is some work that uses formal methods for developing the decisional level of robotic systems, there is not much work that focuses on using them for developing the functional level.

In this work, we propose a novel approach for developing functional levels of robotic systems. Our approach is based on a combination of the BIP component-based design framework and the GenoM architectural tool for developing functional level modules. With our approach, one can synthesize a functional level that is correct-by-construction. Using the D-Finder verification tool, we can check

[10] `http://homepages.laas.fr/mallet/soft/architecture/genom3`

[11] Pocolib is the communication, data sharing, etc. library which we used at LAAS.

offline for properties such as deadlocks. In this way we have verified that a significant part of our functional level is deadlock free, in particular, each individual module in the functional level, as well as the four laser based navigation modules.

On a related note, another aspect on which we intend to improve the current system is to allow the user to check deadlock freedom on a particular subset of components, or more precisely on a particular chain of processing. For example, we may want to focus the deadlock search on the perception/navigation loop, and show that under operational conditions, the navigation algorithm will run properly without reaching a deadlock.

We were able to run experiments with a complete functional and decisional level on our Dala rover, and to demonstrate via fault injections that the BIP controller successfully enforces the constraints supplied, thereby preventing the rover from reaching undesirable/unsafe situations, and that it reports appropriately to the decisional level.

From a comparison of the performance of the GenoM functional level and the corresponding BIP one, we noticed that the BIP modules, resulting from our first implementation, used approximately ten times more CPU time on average than their GenoM counterparts. Motivated by such efficiency concerns, we worked on a real-time version of BIP, which takes into account execution time deadlines and is more efficient when evaluating the possible interactions from the model.

We are also working on a multi-CPU version of real-time BIP, which allows to exploit multi-core platforms by fully distributing the BIP engine and the model over multiple CPUs. Existing methods have been developed for distributing untimed BIP models (Bonakdarpour et al., 2010; Jaber, 2010). They involve correct-by-construction transformations of multi-party interactions of BIP into asynchronous message passing. The code generation for *Send/Receive* BIP models obtained from these transformations is parameterized by a mapping of the components on the target platform, that can be refined depending on performance analysis. A non trivial problem when trying to extend this approach to real-time execution is the presence of a non uniform measure of the real-time provided by the multiple clocks of the platform.

In light of issues related to the difficulty in directly using the BIP language to specify constraints on the functional level, we developed a user-friendly constraint language that roboticists can use to conveniently specify constraints in textual format, which are then automatically converted into BIP connectors and eventually enforced at runtime by the BIP based controller. Finally, we described some original work on using BIP as a temporal-plan execution controller, which controls the execution of a supplied temporal plan whose actions are executed through the

functional level.

This last piece of work is very promising as it paves the path to using a common modeling language from the functional level up to the decisional level (even if we start from a GenoM model for the former and from an Allen logic based temporal plan for the latter). This has always been a major concern for autonomous systems developers (Bernard et al., 2000): using a consistent model across the entire system, and the possibility of formally proving properties over the complete system (not just on particular modules or components). Of course, there are still issues and problems to be addressed to reach such an ambitious goal, but we think that BIP has the potential to provide such a common modeling framework.

Abdellatif, T., Combaz, J., Poulhìes, M., 2011. Correct implementation of open real-time systems. In: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11). pp. 57–64.

Abdellatif, T., Combaz, J., Sifakis, J., 2010. Model-based implementation of real-time applications. In: Proceedings of the tenth ACM international conference on Embedded software (EMSOFT'10). pp. 229–238.

Allen, J. F., November 1983. Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843.

Alur, R., Dill, D. L., April 1994. A theory of timed automata. Theor. Comput. Sci. 126, 183–235.
URL http://portal.acm.org/citation.cfm?id=180782.180519

Basu, A., Bozga, M., Sifakis, J., 2006. Modeling heterogeneous real-time components in BIP. In: Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM-06). Pune, India, pp. 3–12.

Bensalem, S., Bozga, M., Nguyen, T.-H., Sifakis, J., 2008a. Compositional verification for component-based systems and application. In: Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA-08). Seoul, pp. 64–79.

Bensalem, S., de Silva, L., Gallien, M., Ingrand, F., Yan, R., 2010a. "Rock Solid" Software: A Verifiable and Correct-by-Construction Controller for Rover and Spacecraft Functional Levels. In: International Symposium on Artificial Intelligence, Robotics and Automation for Space. Sapporo, Japan.

Bensalem, S., de Silva, L., Ingrand, F., Yan, R., 2011. A verifiable and correct-by-construction controller for robot functional levels. Journal of Software Engineering for Robotics 2 (1).

Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., Nguyen, T.-H., March 2009. Designing autonomous robots. IEEE Robotics and Automation Magazine 16 (1), 66–77.

Bensalem, S., Ingrand, F., Sifakis, J., May 2008b. Autonomous robot software design challenge. In: IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments. Pasadena, CA.

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

Bernard, D., Gamble, E., Rouquette, N., Smith, B., Tung, Y., Muscettola, N., Dorias, G., Kanefsky, B., Kurien, J., Millar, W., 2000. Remote agent experiment ds1 technology validation report. Tech. rep., NASA.

Bliudze, S., Sifakis, J., 2008. The algebra of connectors - structuring interaction in bip. IEEE Trans. Computers 57 (10), 1315–1330.

Bohren, J., Cousins, S., dec. 2010. The smach high-level executive [ros news]. Robotics Automation Magazine, IEEE 17 (4), 18 –20.

Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mösenlechner, L., Meeussen, W., Holzer, S., 2011. Towards autonomous robotic butlers: Lessons learned with the pr2. In: ICRA. pp. 5568–5575.

Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J., 2010. From high-level component-based models to distributed implementations. In: Proceedings of the tenth ACM international conference on Embedded software (EMSOFT'10). pp. 209–218.

Bordini, R. H., Fisher, M., Pardavila, C., Visser, W., Wooldridge, M., 2003. Model checking multi-agent programs with casp. In: CAV. pp. 110–113.

Boussinot, F., de Simone, R., September 1991. The ESTEREL Language. Proceeding of the IEEE, 1293–1304.

Broy, M., Jonsson, B., Katoen, J., Leucker, M., Pretschner, A., 2005. Model-based Testing of Reactive Systems. Lecture Notes in Computer Science 3472.

Bruyninckx, H., 2001. Open robot control software: the orocos project. In: ICRA. Seoul, Korea.

Espiau, B., Kapellos, K., Jourdan, M., October 1995. Formal verification in robotics: Why and how. In: The International Foundation for Robotics Research, editor, The Seventh International Symposium of Robotics Research. Cambridge Press, Munich, Germany, pp. 201 – 213.

Feiler, P. H., Lewis, B. A., Vestal, S., 2006. The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems. In: IEEE International Symposium on Computer-Aided Control Systems Design. pp. 1206–1211.

Fleury, S., Herrb, M., Chatila, R., 1997. GenoM: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In: IROS-97. pp. 842–848.

Goldman, R. P., Musliner, D. J., Pelican, M. J., 2000. Using model checking to plan hard real-time controllers. In: Proceedings of the AIPS Workshop on Model-Theoretic Approaches to Planning.

Halbwachs, N., 1992. Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, Norwell, MA, USA.

Ingrand, F., Chatila, R., Alami, R., Robert, F., apr 1996. Prs: a high level supervision and control language for autonomous mobile robots. In: Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on. Vol. 1. pp. 43 –49 vol.1.

Ingrand, F., Lacroix, S., Lemai, S., Py, F., 2007. Decisional autonomy of planetary rovers. Journal of Field Robotics 24 (7), 559–580.

Jaber, M., 2010. Centralized and distributed implementations of correct-by-construction component-based systems by using source-to-source transformations in bip. Ph.D. thesis, Grenoble Universités.

Jackson, J., 2007. Microsoft robotics studio: A technical introduction. IEEE RAM 14 (4), 82–87.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. The unified software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kowalski, R., Sergot, M., January 1986. A logic-based calculus of events. New Generation Computing 4, 67–95.

Kramer, J., Scheutz, M., Jan 2007. Development environments for autonomous mobile robots: A survey. Auton Robot.

Kress-Gazit, H., Pappas, G., May 2010. Automatic synthesis of robot controllers for tasks with locative prepositions. In: Robotics and Automation (ICRA), 2010 IEEE International Conference on. pp. 3215 –3220.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., Ingrand, F., 2010. GenoM3: Building middleware-independent robotic components. In: Robotics and Automation (ICRA), 2010 IEEE International Conference on. pp. 4627–4632.

Montemerlo, M., Roy, N., Thrun, S., 2003. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In: Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems. Las Vegas, NV, pp. 2436–2441.

Nesnas, I. A., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., Oct 2003. Claraty and challenges of developing interoperable robotic software. In: IROS. Las Vegas, NV, invited paper.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A., 2009. Ros: an open-source robot operating system. In: International Conference on Robotics and Automation. Kobe, Japan.

Rosu, G., Bensalem, S., 2006. Allen linear (interval) temporal logic - translation to ltl and monitor synthesis. In: Ball, T., Jones, R. B. (Eds.), CAV. Vol. 4144 of Lecture Notes in Computer Science. Springer, pp. 263–277.

Shakhimardanov, A., Prassler, E., Sep 2007. Comparative evaluation of robotic software integration systems: A case study. In: IROS. San Diego, CA, p. 7.

Shanahan, M., 2000. An abductive event calculus planner. Journal of Logic Programming 44, 207–239.

Sifakis, J., 2005. A framework for component-based construction extended abstract. In: Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM-05). IEEE Computer Society, Washington, DC, USA, pp. 293–300.

Simmons, R., Pecheur, C., Srinivasan, G., 2000. Towards automatic verification of autonomous systems. In: IEEE/RSJ International conference on Intelligent Robots & Systems.

Vaughan, R., Gerkey, B., 2007. Reusable robot software and the player/stage project. Software Engineering for Experimental Robotics, 267–289.

Williams, B. C., Ingham, M. D., Chung, S., Elliott, P., Hofbaur, M., Sullivan, G. T., winter 2003. Model-Based Programming of Fault-Aware Systems. Artificial Intelligence, 61–75.

Wongpiromsarn, T., Topcu, U., Murray, R. M., 2010. Receding horizon control for temporal logic specifications. In: Proceedings of the 13th ACM international conference on Hybrid systems: computation and control. HSCC '10. ACM, New York, NY, USA, pp. 101–110.