# Testing the Input Timing Robustness of Real-time Control Software for Autonomous Systems

David Powell[1,2], Jean Arlat[1,2], Hoang Nam Chu[1,2], Félix Ingrand[1,2], Marc-Olivier Killijian[1,2]
[1]CNRS ; LAAS ; 7 avenue du Colonel Roche, F-31077 Toulouse, France
[2]Université de Toulouse ; UPS , INSA , INP, ISAE ; LAAS ; F-31077 Toulouse, France
{david.powell, jean.arlat, hoang-nam.chu, felix.ingrand, marc-olivier.killijian@laas.fr

***Abstract*—The functional layer of an autonomous system such as a robot is required to carry out multiple real-time control activities in parallel. These activities are launched by asynchronous calls from clients situated at higher layers, so there is a need for the functional layer to provide built-in protection to ensure that it is robust with respect to requests that are issued at instants that are incompatible with its current state and could therefore cause catastrophic system failure. This paper addresses the testing of the robustness provided by such protection mechanisms. A hybrid black-box robustness testing approach is considered by which test cases are generated by random mutation of a valid sequence of requests and test verdicts are obtained by a set of property-based robustness oracles applied to a logged trace of requests and responses. An application of the proposed framework to an experimental planetary explorer robot is described.**

***Index Terms*—software; robustness; testing; fault injection; robotics; dependability**

## I. Introduction

Autonomous systems cover a broad spectrum of applications, from robot pets and vacuum cleaners, to museum tour guides, planetary exploration rovers, deep space probes and, in the not-too-distant future, domestic service robots. As autonomous systems are deployed for increasingly critical and complex tasks, there is a need to demonstrate that they are sufficiently reliable and will operate safely in all situations that they may encounter.

By reliable, we mean that the autonomous system is able to fulfill its assigned goals or tasks with high probability, despite unfavorable endogenous and exogenous conditions, such as internal faults and adverse environmental situations. Reliability in the face of adverse environmental situations (sometimes referred to as (system-level) robustness) is a particularly important requirement for autonomous systems, which are intended to be capable of operating in partially unknown, unpredictable and possibly dynamic environments.

By safe, we mean that the autonomous system should neither cause harm to other agents (especially humans) in its environment nor cause irreversible damage to its own critical resources. Protection of its own integrity is in fact a pre-requisite for an autonomous system to be reliable: if critical resources are no longer usable, then no amount of automated reasoning will be able to find a course of actions enabling the system to fulfill its goals.

This paper addresses the assessment of a particular type of safety mechanism for an autonomous robot, implemented within its control software. The safety mechanism aims to enforce a set of *safety constraints* that specify inconsistent or dangerous behaviors that must be avoided. Examples of safety constraints are, for instance, that a mobile manipulator robot should not move at high speed if its arm is deployed, or that a robot planet observation satellite should not fire its thrusters unless its camera lens is protected.

The safety constraint enforcement mechanisms are implemented within the lowest layer of robot control software (called here the *functional layer*), which interfaces directly with the robot hardware. Typically, such a software layer contains built-in system functions that control the robot hardware and provides a programming interface to the next upper layer. Specifically, clients of the functional layer can issue requests to initialize modules, update their internal data structures, or start and stop various primitive behaviors or *activities*, such as: rotate the robot wheels at a given speed, move the robot to given coordinates whilst avoiding obstacles, etc.

Upper-layer clients can build more complex behaviors by issuing asynchronous requests to start and stop activities at the functional layer. We consider that the high-level safety constraints are expressed in terms of *safety properties* that place restrictions on when given functional layer activities can be executed. For example, a property might require mutual exclusion between activities $x$ and $y$. Thus, if a client issues a request for $x$ while $y$ is executing, enforcement of the property would require, for example, the request for $x$ to be rejected.

In this paper, we define a method for assessing the effectiveness of such property enforcement mechanisms. We address the problem from the perspective of *robustness testing*, where robustness if defined as the "degree to which a system or a component can function correctly in the presence of invalid inputs or stressful environmental conditions" [1]. From our perspective, an invalid input is an application-layer request that, if executed in the current state of the functional layer, would cause a safety property to be violated. We specifically address invalidity in the time domain (i.e., requests issued at the "wrong" moment), although our approach could easily be extended to embrace the more classic notion of invalidity in the value domain (i.e., incorrect request parameters).

We adopt a random testing approach based on fault injection, through which a large number of test cases are generated automatically by mutating a sequence of valid inputs. Our test approach thus allows robustness evaluation in the sense that

we can provide descriptive statistics of the robustness behavior of the system under test (SUT) (in our case, a functional layer implementation) with respect to the population of test cases. Moreover, we follow a black-box testing approach, which does not consider internal details of the SUT. Thus, a set of test cases generated using our approach can be applied as a robustness benchmark to compare different functional layer implementations.

The paper is structured as follows. In Section II, we discuss related work on robustness testing before describing, in Section III our system under test, which has several specificities that require us to adopt a new approach. Our testing framework is described in Section IV. In Section V, we apply our approach to a case study: the Dala experimental robot configured as a planet exploration rover. Section VI concludes the paper and suggests directions for future work.

## II. Related work

Based on the already-mentioned IEEEStd. 610-12 definition of robustness [1], robustness testing can be considered from two viewpoints: testing with respect to invalid inputs and testing with respect to environmental stress. The latter form of robustness testing may be concerned with either physical aspects of the environment (temperature, pressure, radiation, power fluctuations, etc.) or informational aspects such as the load placed on the system or the difficulties of the problems that the system needs to solve. Here, we focus on the former viewpoint: testing system robustness with respect to invalid inputs, which we refer to as *input robustness testing*. We can distinguish two sorts of invalid inputs:

- *inputs that are invalid in the value domain*, which are inputs that are not described in the system specification, or specified inputs that have one or more parameters taking out-of-range or otherwise incorrect values;
- *inputs that are invalid in the time domain*, which are specified inputs but whose occurrence is not expected or is inappropriate in the current state of the system.

There has been a considerable amount of work on input robustness testing, which can be roughly divided into two categories, according to whether test generation is guided by an input-domain model or a behavioral model.

*Methods based on input domain models* generate invalid inputs based on an analysis of the system input specification or by mutating valid inputs. These methods are often based on random injection of faults or errors, and directed more towards *robustness evaluation* rather than robustness verification (discovery of robustness defects), which is nevertheless achieved as a side effect. This category of methods includes: fuzz testing (e.g., [2], [3]), which submits random data inputs to a program and thus blindly tests both value and time domains robustness; syntax testing [4], which is a value domain technique aimed at testing the robustness of string lexing and parsing systems; domain testing [4], which is a heuristic method for testing extreme values of inputs; and parameter corruption, in which correct input parameters are selectively replaced by either out-of-range values or in-range but incorrect values, as used in Ballista [5], [6], Mafalda [7], [8] and DBench [9].

We briefly present the latter DBench approach as an example of this category of methods. The goal of dependability benchmarking is to a provide generic and reproducible way of characterizing the behavior of a computer system in the presence of faults. Kanoun *et al.* [9] define a dependability benchmark for general-purpose operating systems and apply it to six versions of Windows OS and four versions of Linux OS. This study aimed to measure the robustness of Windows and Linux in the presence of faults by injecting erroneous inputs to the OS via its API. The benchmark execution profile consisted of a PostMark workload together with fault injection into parameters of system calls. PostMark creates a large pool of continually changing files and measures the transaction rates (a workload approximating a large Internet electronic mail server). Fault injection was carried out using the parameter corruption technique relying on analysis of system call parameters to define substitution values to be applied to these parameters. During runtime, the workload system calls are intercepted, corrupted and re-inserted.

Test results are classified according to the following observation categories:

- **SEr**: An error code is returned.
- **SXp**: An exception is raised.
- **SPc**: The system is in panic state, and is no longer servicing the application.
- **SHg**: Hang state, from which it is necessary to do a hard reboot.
- **SNS**: No-signaling state.

The robustness of the various operating systems is evaluated and compared based on statistics on the proportions of test results in each of these categories. Such a system-level characterization of test results is typical of robustness testing approaches based on input domain models.

*Methods based on behavioral models* use a formal model of the SUT to define both the test cases and an oracle enabling a decision to be made regarding system robustness. These methods generally concentrate on *robustness verification* rather than evaluation. If the available behavior model covers both nominal behavior (reaction to valid inputs) and robust behavior (reaction to invalid inputs), robustness testing can be expressed in the same way as conformance testing. If, on the contrary, the available behavior model only includes the nominal behavior, the model must be extended in some way to include a specification of the expected robust behavior [10], [11].

We briefly present the approach of Saad-Khorchef *et al.* [11], [12] as an example of this category of methods. The proposed method is aimed at testing the robustness of communication protocols and is one of few works that we know of that takes explicitly into account input invalidity in both the value and time domains. Starting from a *nominal specification* given in the form of an IOLTS (Input Output Labelled Transition System) model, which describes the system by a set of states, a set of transition relations between states and an alphabet of transition actions, the test method comprises two phases: (i) the

construction of an *augmented specification*, which describes the acceptable behaviors in the presence of invalid inputs, by integrating so-called "hazards" (events that are not expected in the nominal specification); (ii) the generation of conformance tests based on the latter augmented specification. To the best of our knowledge, these model-based methods have only been applied to a few toy examples of limited size.

A third category of method, called *hybrid robustness testing* has been proposed by Cavalli *et al.* [13], that contains aspects of both input-domain model-based methods and behavior model-based methods. The proposed method combines fault-injection and passive testing [14]. It is aimed at characterizing the robustness of communication protocol implementations in the presence of communication faults, which are categorized as either omission faults (lost messages), so called "arbitrary" faults (corrupted messages) and performance faults (delayed messages). Robust behavior is expressed as a set of *invariant* properties, which specify the allowable input and output sequences that a system can produce. The proposed approach consists of the following main steps: (i) build a formal model of the system behavior; (ii) define the invariant properties and check them against the formal behavior model; (iii) define the fault model and the faults to inject; (iv) instrument the SUT for fault injection and monitoring purposes; (v) execute the tests by activating the SUT while injecting the faults and monitoring the SUT interactions; (vi) analyze the results based on the defined invariants and produce a robustness verdict.

The method proposed in this paper can also be considered a hybrid robustness testing approach that combines random fault-injection and passive testing. Our focus is on injecting system inputs that are invalid in the time domain in order to assess the effectiveness of a SUT's built-in capabilities to defend itself against asynchronous inputs that cannot be processed in its current state. We follow a passive testing approach in that we define property-based robustness oracles that allow the analysis of system traces obtained in the presence of invalidly-timed inputs. We also categorize system traces according to two simple DBench-like categories: *no-hang* and *hang*.

## III. System under test

The control software for many practical autonomous systems [15]–[19] is structured as a hierarchy, typically with 3 layers [20]: a functional layer, an executive layer, and a decisional layer.

We present here the LAAS architecture [16], [17] as a typical example of a hierarchical architecture for autonomous systems (Figure 1).

1) The *decisional layer*: At the top of the hierarchy, this layer is responsible for the deliberative aspects of the autonomous system. It is in charge of generating task plans from goals given by the operator and supervising their execution via the executive layer, while taking into account reports and errors notified by the lower layers. In the LAAS 3-layer architecture, the decisional layer uses the IxTeT planner [21] to produce the task plan. IxTeT is a temporal constraint planner, combining high level actions to build plans, and capable of carrying out temporal execution control, plan repair and re-planning.
2) The *executive layer*: This layer carries out the task plans (provided by the decisional layer) by choosing the elementary activities that the functional layer must carry out, and supervises their execution. In the LAAS architecture, OpenPRS (Open Procedural Reasoning System) [22] is used to execute the task plan sent from the decisional layer, while being at the same time reactive to events from the functional layer. The procedural executive OpenPRS is in charge of decomposing and refining plan actions into lower-level actions executable by functional components, and executing them. During execution, OpenPRS reports any action failures to the planner, in order to re-plan or repair the plan.
3) The *functional layer*: This is the layer that controls the basic hardware and provides the functional interface for higher-level components. It includes the basic built-in system functions and perception capabilities (obstacle avoidance, trajectory calculation, communication, etc.). In the LAAS architecture, these functions are encapsulated into $G^{en}oM$ modules [23]. Each module can be in charge of controlling a hardware component (e.g., a camera, a laser sensor, etc.) or accomplishing a particular functionality (e.g., navigation). Modules provide services that can be activated by requests from the executive layer. Services can return data to the caller and export data for use by other modules.

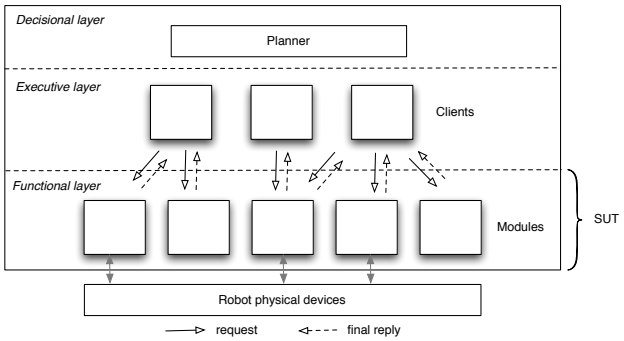


Fig. 1. 3-layer autonomous system software architecture

It is this latter layer that is of particular concern to us in this paper.

Clients situated at the executive layer can issue *requests* to modules of the functional layer to initialize them, to control them (e.g., to make changes to their internal data structures, and to start and stop *activities*). Execution requests are non-blocking. For each request, the called module will return a *final reply* that indicates the outcome of the activity whose execution was requested[1]. Two cases can be distinguished:

1) The request is accepted by the called module, and

[1]In some cases, and intermediate reply is also returned, to indicate to the client that the request is being processed.

the final reply indicates the outcome of the requested activity, as specified by the module code. Nominally, the final reply returns $ok$, but the programmer can specify other, application-specific reasons for termination (e.g., interruption due to the reception of a request of higher priority).

2) The request is rejected by the called module, and the final reply indicates the cause of the rejection (e.g., incorrect call parameters).

Clients can issue executions requests at any time. This allows them to use abstract representations of robot behavior and operate in different timeframes to that of the functional layer, thus favoring tractability of the computations to be carried out at the upper layer. The downside to this higher level of abstraction is that it is possible for the functional layer to receive execution requests that are inconsistent with its current state and that, if executed, could cause the robot to fail catastrophically, causing harm to itself or to other agents in its vicinity. We say that such requests are *untimely*. The functional layer should protect itself against untimely requests by either rejecting them, or queueing them until an appropriate state has been reached, or forcing a state change to one in which they can be accepted.

The specification of what constitutes a timely request and what reaction there should be in the face of an untimely request is given a by a set of *safety properties*. We have identified five basic types of safety property [24], which we summarize below.

- **Precondition property** $PC[x, C_{PRE}]$: an activity of type $x$ may only be started if a specified precondition $C_{PRE}$ is true at the instant that $x$ is requested.[2]
  The precondition itself may correspond to, for example, the SUT being in a specified state, or having successfully executed a specific set of activities.
  Property $PC[x, C_{PRE}]$ can be enforced:
  - by rejecting or queuing[3] requests for $x$ while $C_{PRE}$ is false, or
  - by forcing $C_{PRE}$ to be true (if that is possible) in order to serve requests for $x$.
- **Excluded start property** $ES[x, y]$: an activity of type $x$ may only be started if there is no ongoing activity of type $y$ at the instant that $x$ is requested.
  Property $ES[x, y]$ can be enforced by rejecting or queuing requests for an activity of type $x$ while there is an ongoing activity of type $y$.
- **Excluded execution property** $EE[x, y]$: an activity of type $x$ may only execute in the absence of requests for activities of type $y$.
  Property $EE[x, y]$ can (only) be enforced by interrupting any ongoing activity of type $x$ in the event of a request for an activity of type $y$.
- **Exclusion property** $EX[x, y]$: an activity of type $x$ is excluded by an activity of type $y$.
  Property $EX[x, y]$ can be enforced by rejecting or queuing requests for an activity of type $x$ while there is an ongoing activity of type $y$, *and* interrupting any ongoing activity of type $x$ in the event of a request for an activity of type $y$. We note that $EX[x, y] \equiv ES[x, y] \wedge EE[x, y]$. We also observe that $EX[x, y]$ defines an *asymmetric* exclusion between activities $x$ and $y$, giving priority to $y$.
- **Mutual exclusion property** $MX[x, y]$: activities of types $x$ and $y$ cannot be executed at the same time.
  Property $MX[x, y]$ can be enforced by rejecting or queuing requests for an activity of type $x$ (respectively $y$) while there is an ongoing activity of type $y$ (respectively $x$), *or* by interrupting execution of an activity of type $x$ (respectively $y$) in the event of a request for an activity of type $y$ (respectively $x$).
  We consider two enforcement policies for $MX[x, y]$:
  - *mutual rejection*, noted $MX_R[x, y]$, which favours the currently executing activity by rejecting the latest request;
  - *mutual interruption*, noted $MX_I[x, y]$, which favours the latest request by interrupting the currently executing activity.

## IV. Testing framework

To assess the robustness of the system under test, we adopt a passive testing approach similar to that employed by *Cavalli et al.* [13] in the context of communication protocols (cf. Section II). Testing is passive in that the system under test is observed and assessed in a way that is totally independent of system activation and test generation, at the level of the trace of requests and replies crossing its API. Given a trace of requests and responses, our aim is to define a set of oracles that can classify the behaviour of the SUT with respect to a set of safety properties. Each property oracle characterizes the robustness behaviour of the SUT for each request that is relevant to a given property $P$, where $P$ is any of the properties defined in Section III.

### A. Notation

We define the following notation for requests and responses to the functional layer:

- $x(i)$ (respectively $y(i)$, $q(i)$): a request for activity $x$ (respectively $y$, $q$), with request identifier $i$;
- $F_x$: set of final reply values defined for request $x$ with $F_x = \{R_x, Z_x, T_x\}$, such that:
  - $R_x$: set of final reply values indicating request rejection;
  - $Z_x$: set of final reply values indicating activity interruption;
  - $T_x$: set of final reply values indicating activity termination other than by interruption (in particular, $T_x$ contains the final reply value $ok$, which indicates correct termination of the activity);

[2] In the following, we will often refer simply to "$x$" rather than saying "activity of type $x$"

[3] Queueing is not considered to be viable mechanism in practice due it its potentially negative impact on real-time performance.

- $f_x(i) \in F_x$: the final reply to $x(i)$;
- $r_P \in R_x$ : value of $f_x(i)$ signaling rejection of request $x(i)$ to enforce property $P$;
- $z_P \in Z_x$ : value of $f_x(i)$ signaling interruption[4] of request $x(i)$ to enforce property $P$.

### B. Behaviour categories

For a given property $P$, we can classify the system behaviour for each relevant request according to the following basic outcomes:

- **true negative** (TN): execution of the requested activity is authorized since it does not endanger the property $P$; no invocation of property-enforcement (correct behaviour);
- **true positive** (TP): execution of the requested activity is forbidden since it endangers the property $P$; property-enforcement is invoked (correct behaviour);
- **false negative** (FN): execution of the requested activity is forbidden since it endangers the property $P$; however, there is no invocation of property-enforcement (incorrect behaviour);
- **false positive** (FP): execution of the requested activity is authorized since it does not endanger the property $P$; however, property-enforcement is invoked (incorrect behaviour);

In addition, we consider the following specific outcomes to cover special cases:

- **other positive** (op): the considered request was rejected to enforce some other property $P' \neq P$;
- **not applicable** (na): $P$ is no longer applicable to the requested activity since apparently conflicting requests were rejected to enforce some other property $P' \neq P$;
- **truncated trace** ($\omega$): the end of the trace is reached before any conclusion can be reached.

### C. Property oracles

Figure 2 illustrates the ideal true positive and true negative behaviors with respect to the five basic property types defined in Section III. The arrows represent pertinent events observable at the SUT interface, with downwards arrows representing requests, and upwards arrows representing final replies. The identifiers of requests are omitted to avoid cluttering the figure.

Due to space restrictions, we illustrate our approach by discussing only the simple precondition property $PC$. The illustration at the top of Figure 2 considers a very simple precondition: successful prior execution of an activity of type $q$. The oracle can test for satisfaction of the precondition by evaluating the formula:

$$C_{PRE}(x(i)) = \exists k, (f_q(k) = ok) \wedge (t(f_q(k)) < t(x(i))) \quad (1)$$

where $C_{PRE}(x(i))$ denotes the truth value of the precondition for the request $x(i)$ and $t(event)$ is the time of occurrence of $event$, with $event \in \{f_q(k), x(i)\}$.

[4]We use "interruption of request $x(i)$" as shorthand for "interruption of the activity corresponding to request x(i)"

When the precondition is true (left-hand figure), the request for $x$ should be accepted. When the precondition is false (right-hand figure), the request for $x$ must be rejected.

Note that when $C_{PRE}(x(i)) = true$, the property $PC[x, C_{PRE}]$ is respected for $x(i)$ without any need for explicit enforcement. Enforcement (in this case, by rejection) is only needed when $C_{PRE}(x(i)) = false$. Formally, we can express this as follows:

$PC[x, C_{PRE}] \models behaviour_{SUT} \mid_{x(i)} \Leftrightarrow C_{PRE}(x(i)) \vee reject_{x(i)}$

which reads: the behaviour of the SUT with respect to request $x(i)$ satisfies the property $PC[x, C_{PRE}]$ if and only if either $C_{PRE}(x(i))$ is true or the request $x(i)$ is rejected.

Of course, the property $PC[x, C_{PRE}]$ is trivially satisfied (for any $x(i)$) if all requests of type $x$ are systematically rejected. However, from a robustness evaluation viewpoint, it is important to distinguish whether $x(i)$ is accepted or rejected for the right reason. This can be determined by examining the value of the final reply $f_x(i)$. A request $x(i)$ is judged to be:

- *accepted*, if its final reply $f_x(i)$ indicates that the activity has been executed, either completely ($f_x(i) \in T_x$) or partially ($f_x(i) \in Z_x$);
- *rejected* by the property enforcement mechanism, if its final reply $f_x(i)$ is equal to $r_P$, the specific rejection message defined for the considered property (here, we have $r_P = r_{PC[x,C_{PRE}]}$);
- *rejected for some other reason*, if its final reply $f_x(i)$ is in the set $R_x \backslash r_P$ (i.e., an "other positive" outcome, due to the enforcement of a property other than the considered property).

If no final reply is received before the end of the analyzed trace then no conclusion can be drawn as to whether $x(i)$ was accepted or rejected. We refer to this case as a *truncated trace*.

Considering now both the correct behaviours of the SUT (as depicted on Figure 2) and possible incorrect behaviours (i.e., incorrect acceptance and incorrect rejection), Table I enumerates all the possible observable behaviours of the SUT for a request $x(i)$ with respect to the precondition property $PC[x, C_{PRE}]$ and indicates the corresponding robustness test verdict, including the "truncated trace" case (no final reply $f_x(i)$ is ever received) for which the test verdict is classified as $\omega$.

TABLE I
TEST VERDICTS FOR $PC[x, C_{PRE}]$

| | $f_x(i)$ | | | |
|---|---|---|---|---|
| $C_{PRE}(x(i))$ | $\in \{Z_x, T_x\}$ | $r_P$ | $\in R_x \backslash r_P$ | $\varnothing$ |
| $true$ | TN | FP | op | $\omega$ |
| $false$ | FN | TP | op | $\omega$ |

The oracles for the other basic property types can be found in [24]. For each property type, there is a "$P$-condition" $C_P(x(i))$, where $P \in \{PC, ES, ES, EE, EX, MX_R, MX_I)$ such that, when true for request $x(i)$, indicates that no property enforcement is necessary, and that, when false, indicates

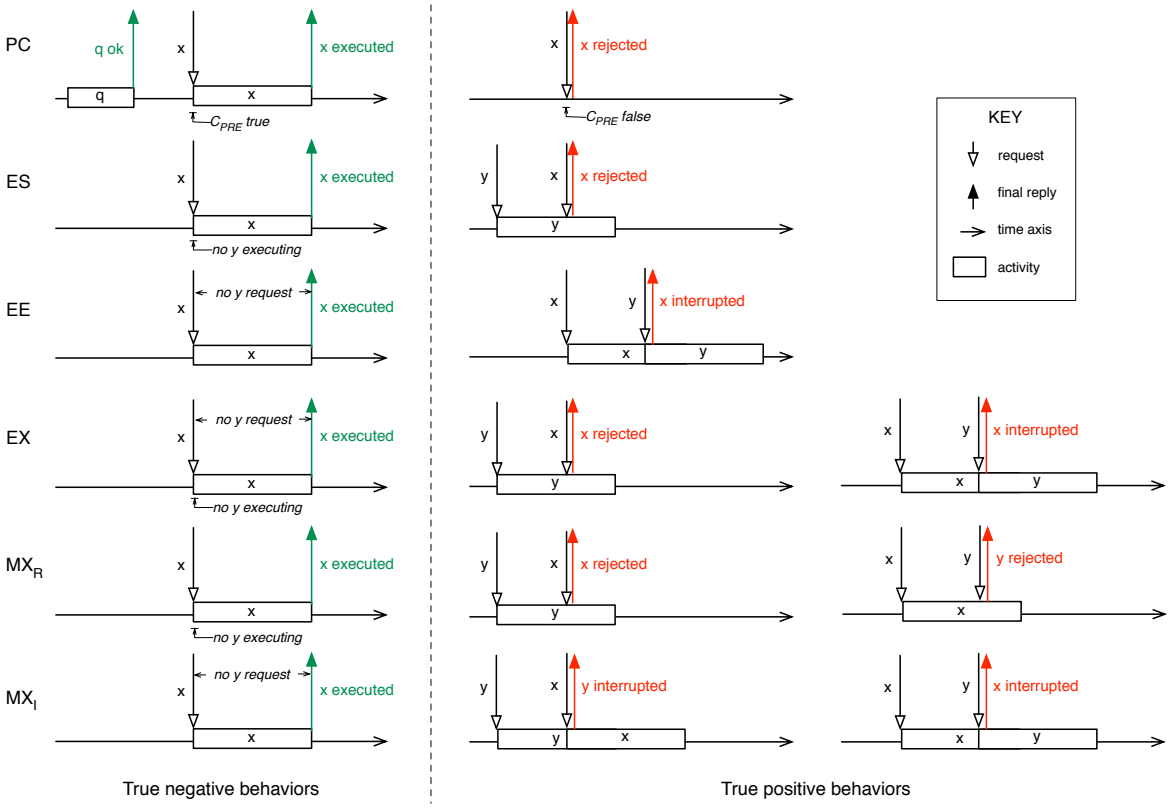


Fig. 2. Illustration of correct behaviors for the various timing robustness properties

that property enforcement (by rejection or by interruption, according to the considered property) is required.

## V. Case Study

Our input timing robustness testing framework has been applied to a case study: the functional layer of the Dala rover, a robot that is currently used in LAAS for navigation experiments. The functional layer is required to respect a number of safety properties to protect the Dala rover from combinations of activities that could lead to inconsistent or dangerous behaviour. Our robustness testing framework exercises the property-enforcing mechanisms of the Dala rover functional layer by means of mutated exploration mission scripts containing potentially *temporally invalid* test inputs that may endanger the safety properties. The execution traces, consisting of the requests and responses intercepted at the functional layer interface, are then processed by a trace analyzer tool to assess the robustness of the functional layer.

We first describe the functional layer of Dala rover (our "system under test"), and the safety properties that it must enforce. Then we present our robustness testing environment and the characteristics of the test campaign that has been carried out. Finally, we present and analyze the results of the case study.

### A. System under test

The system under test is a functional layer comprising five modules, some of which communicate directly with the robot hardware (cf. Figure 3):

- Rflex: odometry and wheel control;
- Sick: laser range finder;
- Aspect: 2D environment map;
- Ndd: navigation and obstacle avoidance;
- Antenna: communication (to/from overhead orbiter) (simulated).

With the exception of Antenna, each module maintains a "poster" (a shared data area) that can be read by the other modules:

- Robot: current position of the robot as computed using the Rflex odometric function;
- Ref: reference velocity of the rover computed by Ndd;
- CartA: 2D environment map computed by Aspect;
- SCart: orientation and range of obstacles currently within view of the Sick laser range finder.

The functional layer receives requests from, and returns replies to, clients at the executive layer, which is here implemented in Open-PRS [22]). There are three varieties of requests: *initialization* requests (to set up a module), *control* requests (to make changes to a module's internal data structures) and *execution* requests (to start or stop activities).

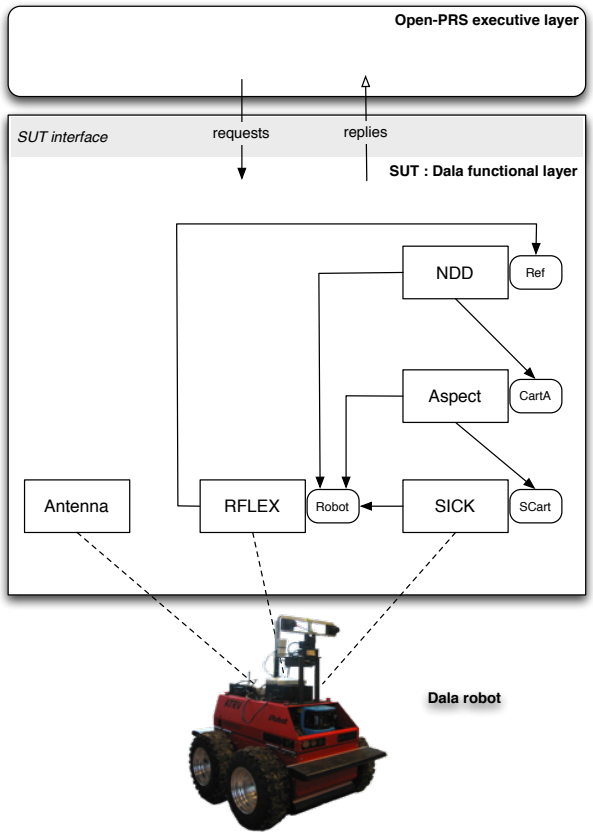


Fig. 3. System under test: Dala robot functional layer

A total of 21 safety properties are defined for this configuration of the Dala Robot [24]. They can be grouped into four families:

- PEX$(module)$ - *Precondition for EXec request*: For each module, there must be at least one successfully executed initialization request before the module can process an execution request.
  Property oracle: Pre-condition $PC[x, C_{PRE}]$
- AIB$(x)$ - *Activity $x$ Interrupted By*: Activities of type $x$ must be inactive or be interrupted if any request of a type "dominating" $x$ is received. Domination relationships between requests are statically defined in the module description code.
  Property oracle: Exclusive execution $EE[x, y]$
- PRE$(x, Q)$ - *activity $x$ PREceded by*: An activity of type $x$ cannot be executed until a specified set of activities $Q$ has been successfully terminated (in any order).
  Property oracle: Pre-condition $PC[x, C_{PRE}]$
- EXC$(x, y)$ - *mutual EXClusion between activites $x$ and $y$*: Activities of type $x$ and $y$ cannot be executed at the same time; priority to the most recent request.
  Property oracle: Mutual exclusion by interruption $MX_I[x, y]$

The proposed robustness testing approach has been applied to three implementations of the Dala functional layer, which we designate as follows:

- *GenoM*: a well-established implementation using the standard *GenoM* environment [23], which provides built-in protection to ensure properties of the families $PEX$ and $AIB$ only;
- *BIP-A*: a preliminary implementation using the BIP formal development framework [25]–[27], with a large proportion of BIP code generated automatically from the GenoM module descriptions, together with additional protection mechanisms generated from BIP inter-component connectors;
- *BIP-B*: a more mature implementation using the BIP framework with, in particular, several corrections resulting from the experiments carried out on *BIP-A*.

### B. Test environment

Testing is carried out with a simulated robot instead of the real Dala robot since (a) testing needs to be automated in order to allow a large number of tests to be carried out, and (b) robustness testing implies the system under test be subjected to aggressive test sequences, which could cause a real robot to behave very dangerously.

The test sequences are obtained by mutating a valid sequence of requests so as to cause some requests to be invalid in the time domain. Of course, there is a high risk that the mutated sequence of requests does not correspond to a meaningful mission at the application level, so it is not of much interest to observe whether the robot achieves its original goals or not. However, it is of concern to us whether the safety properties of the functional layer (the system under test) are satisfied.

Figure 4 illustrates our test environment, which results from an application of the FARM framework [28]. The main steps of the testing process are:

1) Create manually a *golden script* that defines a typical mission of a planetary explorer (the **A**ctivity set).
2) Generate a database of *mutated scripts* by applying a *mutation* procedure to the golden script (the **F**ault set).
3) Submit the set of mutated scripts to OpenPRS for execution in order to exercise the SUT robustness features.
4) Save the resulting execution traces, each consisting of the sequence of requests sent to and replies received from the SUT, in a Trace Database (the raw **R**eadout set).
5) Use the *Trace Analyzer* tool to parse and process the execution trace database to obtain SUT robustness verdicts for each property $\times$ request combination and for each trace (an extension of the **R**eadout set).
6) Evaluate the robustness statistics (the **M**easure set).

### C. Results

Our *golden script* defines a typical exploration mission of the Dala rover, involving navigation to predefined coordinates to take science photos and communicating with a planetary orbiter. A semi-automated mutation procedure was applied

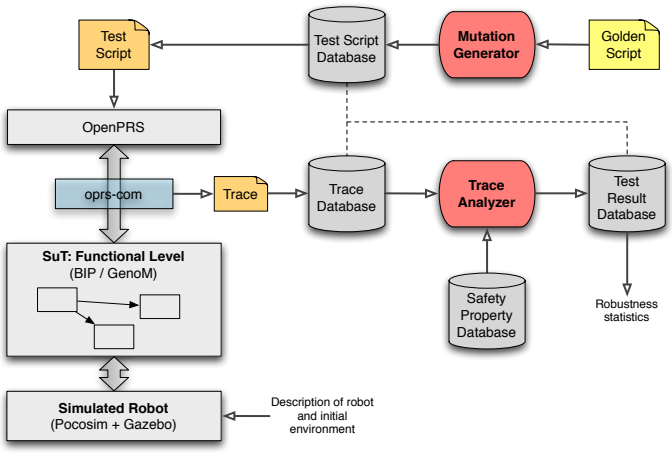


Fig. 4. Robustness testing environment

to the golden script to generate a total 300 mutated scripts. Mutations were applied randomly and consisted of: deletion of a module request; insertion of an additional request; re-ordering of a pair of module requests. 7 mutated scripts could not be used as test scripts since they were systematically rejected as manifestly faulty by the Open-PRS execution environment. The remaining 293 test scripts were applied to all three SUT implementations.

The results of the executions of the test scripts were logged and analyzed after execution by a trace analyzer written as a set of SQL requests. Tables II through IV summarize the main results (see [24] for the detailed results).

Table II summarizes the results obtained per trace for each of the considered implementations. In addition to the number of hung traces, the table reports the number of traces containing at least one falsenegative or at least one false positive (for any property). The column "total bad traces" indicates the number of traces that either hung, or contained a false negative or positive. The final column gives the corresponding measure of *trace robustness* ($T_{ROB}$), which we define as the proportion of good traces.

We note that the *BIP-B* implementation exhibits an appreciable higher degree of robustness than the reference $G^{en}oM$ implementation. There was, however, 1 trace that was categorized as hung, and 11 traces in which false negatives were identified. These traces were thus analyzed manually.

We concluded that the hung trace was due to an uncorrected defect in code generator used for the *BIP-B* implementation. Thus, the actual implementation was not a faithful translation of the formally-proven BIP model. We note, however, that there was a considerable improvement in this respect compared to the immature *BIP-A* implementation, which exhibited 42 hung traces.

Our analysis regarding the observed false negatives led us to the conclusion that these were in fact incorrect verdicts due to an observability problem that is inherent to our black-box testing approach. Indeed, we deduce the internal behavior of the SUT from an observation of the trace of requests and responses visible at its call interface. Since there is an inevitable delay between events observable at the interface and corresponding internal events, it is possible to observe events in a different order on the external (observable) timeline to that in which they occur on the internal (unobservable) timeline. Due to this, it is possible for the values of P-conditions to be inverted, thus leading to incorrect robustness verdicts. We note, however, that this can only occur for *near-coincident* events, i.e., those that occur in an interval of time of the same order of magnitude as the event propagation delay between the internal and external timelines.

Tables III and IV give the true and false positive rates at the level of the four Dala property families and globally, with all properties considered simultaneously. The true positive rate is the proportion of correct reactions of the system under test when the P-condition evaluates to false, i.e., when it is necessary to enforce the considered property. Conversely, the false positive rate is the proportion of incorrect reactions of the system under test when the P-condition evaluates to true, i.e., when no enforcement is necessary.

On Table III, we observe a growth in the true positive rate over the successive implementations, due to the face that the BIP technology allows additional protections to be implemented over and above the limited protections included within the reference $G^{en}oM$ implementation. Indeed, this implementation had no built-in protection to enforce the $PRE$ and $EXC$ property families, thus leading to a null true positive rate in the corresponding cells of Table III. Surprisingly, however, there is also a null true positive rate for the $PRE$ property family in the *BIP-A* implementation, which was supposed to offer the corresponding protection. However, the corresponding connector had been accidentally omitted from the corresponding BIP model. This omission was uncovered by our robustness tests. The true positive rates in the *BIP-B* column are not quite 100%. In fact, we concluded that this column would read 100% after manual correction of the already-mentioned incorrect false negative verdicts due to the false observation problem.

Table IV presents a surprising singularity for the AIB property family in the supposedly mature $G^{en}oM$ implementation. After analysis, we discovered that this implementation possesses a non-documented feature: all initialization requests dominate all activities. Since our property oracle was based on the $G^{en}oM$ documentation, this feature was not included. Thus, behaviors corresponding to activities interrupted by an initialization request were declared to be false positives. Note that, if we were to correct the corresponding property oracle, the would be a corresponding decrease in the true positive rates for AIB in the *BIP-A* and *BIP-B* implementations, since the implementors had, quite naturally, ignored this undocumented feature.

## VI. Conclusion

We have proposed a method and a platform for testing the robustness of the safety-property enforcing mechanisms of the functional layer of a hierarchically-structured autonomous

TABLE II
SUMMARY OF PER TRACE RESULTS

| | Total traces | Hung traces | Traces with $\geqslant 1$ FN | Traces with $\geqslant 1$ FP | Total bad traces | Trace robustness ($T_{ROB}$) |
|---|---|---|---|---|---|---|
| *GenoM* | 293 | | 74 | 5 | 76 | 74.1% |
| *BIP-A* | 293 | 42 | 40 | | 80 | 72.7% |
| *BIP-B* | 293 | 1 | 11 | | 12 | 95.9% |

TABLE III
TRUE POSITIVE RATES (%)

| | *GenoM* | *BIP-A* | *BIP-B* |
|---|---|---|---|
| PEX | 100.0 | 95.7 | 99. 7 |
| AIB | 99.7 | 99.8 | 99.7 |
| PRE | 0 | 0 | 99.4 |
| EXC | 0 | 100.0 | 100.0 |
| All | 93.1 | 96.3 | 99.7 |

TABLE IV
FALSE POSITIVE RATES (%)

| | *GenoM* | *BIP-A* | *BIP-B* |
|---|---|---|---|
| PEX | 0 | 0 | 0 |
| AIB | 0.3 | 0 | 0 |
| PRE | 0 | 0 | 0 |
| EXC | 0 | 0 | 0 |
| All | 0.1 | 0 | 0 |

system. The application to the Dala rover shows several advantages of our method. The adoption of a black-box testing approach allowed us to carry out the testing campaign without any formal behavior specification of the system under test (SUT). With little or no information about the internal activities or states of the SUT, we were still able to compare the effectiveness of the safety enforcement mechanisms of two different implementations of the functional layer based on a behavior categorisation with respect to 21 safety properties, grouped into 4 families. We think that this approach is appropriate for testing off-the-shelf software, for which a formal behavior specification is usually not available.

Using the passive testing technique enabled us to evaluate the robustness of the SUT based on offline observation of logged test execution traces. In our case study, the whole testing process is composed of two phases: exercising the SUT with 293 test cases (which takes around 25 hours) and examining the execution results with the trace analyzer (which takes about 30 minutes). The passive testing technique applied to logged execution traces separates the observation process from the system activation process, and thus avoids us having to re-run the whole test set (25 hours) each time we want to refine the definition of the trace analyzer, for example, to add an additional property.

We have attempted to define properties that are as generic as possible. To this end, we defined five basic safety properties, along with their enforcement policies, that can be instantiated as timing robustness requirements of the functional layer of an autonomous system: pre-condition, excluded start, excluded execution, (asymmetric) exclusion, and (symmetric) mutual exclusion. We believe that they are sufficiently general to allow their application to other systems. For each property type, we have defined the corresponding input timing robustness testing oracle.

We also developed and presented a testing environment that allowed us to evaluate the timing robustness of the functional layer of the Dala planetary exploration rover by subjecting it to invalid inputs in the time domain. Starting from a workload (a typical mission of a planetary explorer) described in a golden script, we stress the SUT by creating mutated scripts containing inputs submitted at the "wrong time". Simulation of the physical hardware of the robot and its environment facilitated our intensive testing process and ensured that injected faults could only cause "virtual" damage. However, we note that simulation cannot totally replace testing (albeit without injected faults) on the real platform, which can reveal phenomena that are hard to simulate faithfully (e.g., due to real-time issues, or hard-to-model sensor and actuator inaccuracies).

The implementation of property-based oracles as a sets of SQL queries proved to be very flexible and easy to maintain. The evaluation environment showed its efficacy in comparing and evaluating different systems. Indeed, thanks to the ability to explore thoroughly the SUT's reaction to untimely inputs, our approach to robustness testing allows both fault removal (debugging) by studying the consequences of fault injection, and fault forecasting (evaluation) through statistical measures of system behavior with respect to fault occurrence.

However, our approach does present certain limitations.

The absence of a formal behavior specification of the SUT may lead to an inaccurate oracle, and testers thus have to progressively improve it manually. The question is how? In the hybrid robustness testing approach proposed in [13], the authors verified the correctness of their invariant properties by checking them against a formal model of the system behavior before using them as a robustness testing oracle. This approach could not be applied in our context since we did not have any document that could serve as an authoritative specification of the implementations being tested. We thus had to manually analyse the results produced by the oracle to identify singularities (e.g., too many *False Positives* or *False Negatives*), and then examine the execution trace to diagnose the source of the problem (oracle inaccuracy or SUT misbehavior). In effect, the oracle and the SUT are tested back-to-back and iteratively corrected. We were aided in this respect by the fact that we had several SUTs, one of which (*GenoM*) was a mature implementation (at least with respect to a subset of the required safety properties).

Another limitation is the possibility of incorrect test verdicts due to false observations of *P-conditions*, which are inevitable due to the fact that we cannot control the propagation time of events in the SUT. In the Dala rover case study, our analysis concluded that all false negatives observed on the *BIP-C* implementation in fact corresponded to incorrect verdicts, i.e., true negative situations that were declared erroneously to be false negatives. In each case there was a plausible explanation of how correct behavior of the SUT (i.e., *true positive* or *true negative* behavior) could be wrongly interpreted as incorrect behavior due to uncertain propagation delays.

The inverse is also possible, i.e., misinterpretation of incorrect behavior as correct behavior. Unfortunately, such misinterpretations cannot, by essence, be identified since the observed behavior presents no singularities (it is the expected behavior), so there is no reason to bring it into doubt. This is especially problematic in the case of misinterpreting a false negative as a true negative, as that would be optimistic from a safety viewpoint. Thus, some finer degree of SUT observation is likely to be necessary for testing the robustness of extremely critical systems.

Several directions for future research can be considered.

One area for improvement would be to reduce the number of incorrect test verdicts raised by black-box robustness testing, and the associated observability issues. At least two complementary directions can be considered:

1) Include explicit consideration of real-time in the property oracles to flag verdicts on closely-separated events as "suspicious". In our case study, only in the *BIP-B* implementation were there sufficiently few false negatives in order to justify a tedious manual inspection of execution traces.
2) Study possible modifications or extensions to the interface protocol to facilitate robustness testing (for example, by requiring "intermediate replies" to be sent systematically).

Another area for improvement is in test generation. In particular, the generation of test inputs could definitely be improved in order to make it more automatic. For example, it should be possible to adapt an automatic program mutation tool, such as SESAME [29], to automatically inject faults into a golden test script written in the Open-PRS format. Alternatively, a more deterministic generation of test scripts could be envisaged, focussing on the falsification of the considered P-conditions.

It would also be interesting (and relatively straightforward) to extend the proposed hybrid input timing robustness testing approach to include classic value domain robustness properties.

## Acknowledgment

This work was partially financed by the *Fondation Nationale de Recherche pour l'Aéronautique et l'Espace* (FNRAE) through project MARAE (*Méthode et Architecture Robuste pour l'Autonomie dans l'Espace*), which was carried out by a consortium consisting of LAAS-CNRS, Vérimag and EADS Astrium.

## References

[1] IEEE729, "Standard glossary of software engineering terminology," *IEEE Standard*, 1982.

[2] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Comm. ACM*, vol. 31, no. 10, 1990.

[3] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macOS applications using random testing," in *1st International Workshop on Random Testing (RT '06)*. Portland, Maine New York, NY, USA: ACM, 2006, pp. 46–54.

[4] B. Beizer, *Black-box Testing : Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[5] P. Koopman, "Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security," in *Computer Security, Dependability and Assurance: From Needs to Solutions*, 1998, pp. 103 –1131.

[6] N. Kropp, P. Koopman, and D. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *28th IEEE International Symposium on Fault-Tolerant Computing (FTCS-28)*, 1998, pp. 230 – 2239.

[7] J. Arlat, J.-C. Fabre, and M. Rodriguez, "Dependability of COTS microkernel-based systems," *Transactions on Computers*, vol. 51, no. 2, pp. 138–163, 2002.

[8] J. Arlat, J.-C. Fabre, M. Rodriguez, and F. Salles, "MAFALDA: a series of prototype tools for the assessment of real time COTS microkernel-based systems," in *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003.

[9] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau, "Benchmarking the dependability of windows and linux using postmark workloads," *16th IEEE International Symposium on Software Reliability Engineering (ISSRE05)*, pp. 11–20, 2005.

[10] A. Tarhini, A. Rollet, and H. Fouchal, "A pragmatic approach for testing robustness on real-time component based systems," in *3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005, pp. 143–.

[11] F. Saad-Khorchef, A. Rollet, and R. Castanet, "A framework and a tool for robustness testing of communicating software," in *SAC '07*, vol. Proceedings of the 2007 ACM symposium on Applied computing. Seoul, Korea New York, NY, USA: ACM, 2007, pp. 1461–1466.

[12] F. Saad Khorchef, I. Berrada, A. Rollet, and R. Castanet, "Cadre formel pour le test de robustesse - application au protocole SSL," in *Colloque Francophone sur l'Ingnierie des Protocoles - CFIP 2006*, Tunisie, 2006.

[13] A. Cavalli, E. Martins, and A. Morais, "Use of invariant properties to evaluate the results of fault-injection-based robustness testing of protocol implementations," in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, 2008, pp. 21–30.

[14] B. T. Ladani, B. Alcalde, and A. Cavalli, "Passive testing - a constrained invariant checking approach," *Testing of Communicating Systems*, pp. 9–22, 2005.

[15] D. Musliner, E. Durfee, and K. Shine, "CIRCA: a cooperative intelligent real-time control architecture," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 6, pp. 1561–1574, 1993.

[16] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *International Journal of Robotic Research*, vol. 17, no. 4, pp. 315–337, 1998.

[17] F. Ingrand, R. Chatila, and R. Alami, "An architecture for dependable autonomous robots," in *1st IARP - IEEE/RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments*, Seoul, Korea, 2001.

[18] T. Estlin, R. Volpe, I. Nesnas, D. Mutz, F. Fisher, B. Engelhardt, and S. Chien, "Decision-making in a robotic architecture for autonomy," in *6th Int. Symp. on Artificial Intelligence, Robotics and Automation for Space (i-SAIRAS 2001)*, Montreal, CA, USA, 2001.

[19] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *IEEE Aerospace Conference*, Big Sky, Montana, USA, 2001.

[20] E. Gat, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*, D. Kortenkamp, R. Bonnasso, and R. Murphy, Eds. MIT/AAAI Press, 1997, pp. 195–210.

[21] M. Ghallab and H. Laruelle, "Representation and control in IxTeT, a temporal planner," in *2nd Int. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*. Chicago, IL, USA: AIAA Press, 1994, pp. 61–67.

[22] F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A high level supervision and control language for autonomous mobile robots," in *IEEE Int. Conf. on Robotics and Automation*, Minneapolis, MN, USA, 1996, pp. 43–49.

[23] S. Fleury, M. Herrb, and R. Chatila, "Genom: a tool for the specification and the implementation of operating modules in a distributed robot architecture," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'97)*, vol. 2, Grenoble, France, 1997, pp. 842–848.

[24] H. Chu, "Test and evaluation of the robustness of the functional layer of an autonomous robot," Ph.D. dissertation, Institut Polytechnique de Toulouse, Univerity of Toulouse, 2011.

[25] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *4th IEEE Int'l Conf. on Software Engineering and Formal Methods (SEFM'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 3–12.

[26] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen, "Designing autonomous robots: Toward a more dependable software architecture," *Robotics & Automation Magazine, IEEE*, vol. 16, no. 1, pp. 67–77, 2009.

[27] S. Bensalem, L. de Silva, M. Gallien, F. Ingrand, and R. Yan, "'Rock Solid' software: A verifiable and correct-by-construction controller for rover and spacecraft functional level," in *i-SAIRAS 2010, The 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Sapporo, Japan, 2010.

[28] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation - a methodology and some applications," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[29] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell, "The SESAME experience: from assembly languages to declarative models," in *Mutation 2006 - The Second Workshop on Mutation Analysis, 17th IEEE Int. Symp. on Software Reliability Engineering (ISSRE 2006)*. Raleigh, NC, USA: IEEE, 2006.