Robustness Testing of Robot Controller Software

Hoang-Nam Chu, Jean Arlat, Marc-Olivier Killijian, Benjamin Lussier, David Powell

Robustness Testing of Robot Controller Software

Hoang-Nam Chu*§, Jean Arlat*§, Marc-Olivier Killijian*§, Benjamin Lussier*§ and David Powell*§
*CNRS ; LAAS ; 7 avenue du Colonel Roche, F-31077 Toulouse, France
§Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
Email: firstname.lastname@laas.fr

***Abstract*—The LAAS architecture [1] is a three-layer software architecture for real-time control of mobile robots, that has been used successfully in several systems. To improve its robustness, the architecture is currently being restructured with BIP [2] (*Behavior - Interaction - Priority*). In this paper, we describe a fault injection approach for testing the architecture's robustness.**

***Keywords*—Robustness testing, fault injection, experimental evaluation, autonomous system.**

I. Introduction

Robustness is defined as "the degree to which a system or component can function *correctly* in the presence of *invalid inputs* or *stressful environmental conditions*" (IEEE Std. 610-12, 1990). There are two objectives of robustness testing: verification and evaluation. Robustness verification aims to determine if a system's behaviour conforms to its specification (including exceptional behaviour) and to identify deficiencies in the protection against external adverse situations, whereas robustness evaluation aims to measure the degree of protection against external adverse situations.

Our work is being carried out in the scope of project MARAE[1], which focuses on a design and validation method to construct robust software for autonomous space systems, such as satellites and rovers. The LAAS architecture was chosen as a prototype to experiment our approach. This architecture decomposes the controller system into three layers: a functional layer, an execution control layer and a decisional layer. The architecture has been successfully applied to several mobile robots, some of which have performed missions in real situations. To construct a system that is more robust against external adverse situations, the functional and execution control layers are being restructured using BIP [2] (*Behavior - Interaction - Priority*), a modeling environment for real time components.

As exploration missions are very expensive, it is necessary to avoid dangerous situations which could jeopardize the robot or its missions. To do this, a set of safety constraints is defined, e.g., a maximum robot speed, exclusion of simultanous execution of certain actions, absence of deadlock, etc. The functional layer thus has a protective role to ensure that these safety constraints will not be violated. The BIP runtime platform consists of a synchronization engine and inter-component "glue" code that together aim to enforce the safety constraints. The objective of this paper is to propose a method to assess this mechanism's effectiveness and to evaluate experimentally the architecture's robustness.

[1]MARAE is partially funded by the "Fondation Nationale pour la Recherche en Aéronautique et l'Espace". The project consortium is LAAS-CNRS, Verimag, and Astrium Espace

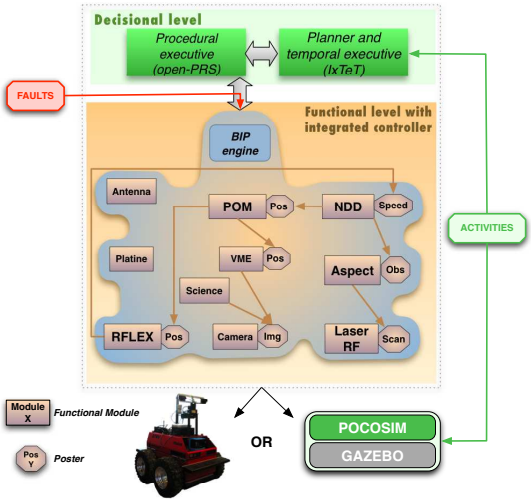

Fig. 1. Experiment environment with fault injection mechanism

II. Fault Injection Environment

Our testing environment is based on fault injection into real robot controller software with simulation of the physical robot. Simulating the physical robot instead of using a real robot is motivated by the two following important reasons:

- It is necessary to perform a large number of experiments in order to achieve a high statistical confidence, whereas experiments on a real system require a lot of time and equipment, and are almost impossible to automate.
- Using a real robot could lead to real hazards if faults are injected into its controller software.

A. Software Architecture

The simulation framework is presented in Figure 1. It incorporates three elements: an open source robot simulator named *Gazebo*, an interface library named *Pocosim*, and the components of the LAAS software architecture.

The *robot simulator Gazebo*[2] is used to simulate the physical world, the physical robot components and the actions of the autonomous system. It allows not only simulation of the rigid object's kinematics and typical robot sensors, but also 3D simulation of the robot environment.

[2]http://playerstage.sourceforge.net/index.php?src=gazebo

The *Pocosim* library [3] is a software bridge between the simulated robot (executed on Gazebo) and the robot functional layer; it processes the requests and replies between Gazebo and the modules, and works as a synchronization center to bridge the temporal gap resulting from using the modules (which work in real time) in a simulated environment (which works in simulated time).

B. Fault Injection and Observation Mechanism

Figure 1 presents our proposed architecture for robustness testing based on the FARM methodology [4]. The fault injection occurs at the interaction level between the functional layer (the target component) and the decisional layer.

1) Workload: We consider the case of a planetary exploration robot for which a basic activity is defined by a mission in a certain environment. The mission is defined by the goals to be completed: scientific photos to be taken, the displacement of the robot to a determined position, etc. The robot's working environment is composed of a group of known or unknown obstables that can prevent it from accomplishing its goals. The experimentation campaign thus will include a group of activities with different difficulties and adversity degrees to take into account the possible variability of multiple missions in an open environment.

2) Faultload: As the functional layer communicates with the decisional layer via messages (requests and replies) transferred using a *Mail Box*, robustness testing can be implemented by injecting faults at the level of these messages. The targetted fault types are: (a) unforeseen messages, (b) message transmission disruptions and (c) message parameter corruptions. These techniques aim to test the robustness of the functional layer under overload conditions and with invalid inputs at the message flow level.

a) Unforeseen messages: Faults can be implemented without modifying the *Mail Box* but by generating and injecting additional messages into it. By bombarding the functional layer with many such messages, we expect to push the robot towards situations which can threaten its safety constraints.

b) Message transmission disruption: This technique requires "opening" the *Mail Box* to modify message-handling routines in order to manipulate message transmission. Message disruption can include: (a) removal of messages, (b) delaying messages, (c) inversion of message order, and (d) repetition of messages.

c) Message parameter corruption: Message parameter corruption is similar to the technique used in BALLISTA [5], MAFALDA [6] and DBench [7]. By analysing message parameters, we define selective substitutions appropriate to these parameters. Message corruption is then carried out by intercepting messages and replacing their parameters on the fly according to these substitutions. To implement this fault type, we again have to "open" the *Mail Box* to manipulate the messages.

There are two types of value substitutions:

- Valid value: the corrupted value is in the validity range of the parameter. For example, an *int* valid value is in the range of [MIN_INT, MAX_INT].
- Invalid value: this is an out-of-range value.

3) Observation: Since the faults injected affect not only the functional layer but also the whole system, we propose to observe the system's behaviour at three levels: *the functional (interface) level*, where we observe any error messages that are returned to the upper layer; *the system level*, where we are interested in the observing various sorts of crashes; and *the application level*, where we observe whether the mission is accomplished and whether the safety constraints are violated.

Table I presents our proposed observation scale to characterize the test results, which is inspired from BALLISTA, MAFALDA and DBench, and adapted to our context.

TABLE I
OBSERVATION SCALE

Interface (Error messages)	E_None	- No error message returned.
	E_C	- Correct error message.
	E_I	- Incorrect error message.
System (Crashes)	C_None	- Nothing observed.
	C_M	- Module crash: One or several modules crash.
	C_B	- BIP engine crash: The whole functional layer blocks.
	C_OS	- OS crash: The simulated robot's OS collapses.
Application (Safety)	S_ok	- Safety respected.
	S_fail	- A safety condition is violated.
Application (Missions)	M_ok	- Mission fulfilled.
	M_fail	- Mission failure: One or several mission objectives are not completed.

III. Conclusion

The proposed fault injection environment allows us to observe the system's response in the face of external adverse situations. We are currently working on a refinement of the observations in terms of a *vector* of outcomes and the definition of a *scale* for measuring robustness. A parallelisation on a computing grid to perform a large number of experiments is also being envisaged.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, pp. 315–337, 1998.

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in BIP," in *Software Engineering and Formal Methods SEFM*, 2006, pp. 3–12.

[3] S. Joyeux, R. Alami, S. Lacroix, and A. Lampe, "Simulation in the LAAS architecture," in *Principles and Practice of Software Development in Robotics (SDIR2005)*, Barcelona, Spain, 2005.

[4] 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, pp. 166–182, 1990.

[5] P. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Fault-Tolerant Computing*, 1999, pp. 30–37.

[6] J. Arlat, J. Fabre, M. Rodriguez, and F. Salles, "Dependability of COTS Microkernel-Based systems," *IEEE Transactions on Computers*, vol. 51, pp. 138–163, 2002.

[7] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau, "Benchmarking the dependability of Windows and Linux using PostMark^{TM} workloads," in *Proc. 16th IEEE Int. Symp. on Software Reliability Engineering*, 2005, pp. 11–20.