GenoM as a Robotics Framework for Planetary Rover Surface Operations*

Ceballos, A. (2); De Silva, Lavindra(1); Herrb, M. (1); Ingrand, F. (1); Mallet, A. (1); Medina, A. (2); Prieto, M. (2)
(1) LAAS/CNRS, 7, Av. du Colonel Roche 31077 Toulouse – France, {ldesilva, felix, mallet, matthieu}@laas.fr
(2) GMV, c/Isaac Newton, 1 28027 Tres Cantos – Spain, {aceballos, amedina, mprieto}@gmv.es

ABSTRACT

This paper presents some recent developments of the LAAS architecture for autonomous systems, and in particular its component to generate functional modules: G^{en}oM. The LAAS architecture was originally designed for autonomous and terrestrial mobile robots. This architecture remains fairly general and is supported by a consistently integrated set of tools and methodologies, in order to properly design, easily integrate, test and validate a complex autonomous system.

The growing complexity of the capabilities of these systems poses a major problem: how do we partition and manage the different system functionalities while making sure that the system never reaches dangerous states? A key requirement to guarantee such safety is through a state-machine able to control the step-by-step execution of core algorithms. The programmer must decompose all algorithmic procedures into the smallest entities that a module can handle, so as to achieve a high controllability of the system.

1. HISTORICAL PERSPECTIVE

From a historical perspective, G^{en}oM 1.0 was released in 1996 at LAAS. In 2004, G^{en}oM 2.0 was released to the public with an open source (BSD licence). G^{en}oM 3.0 is expected to be released by 2011, with a focus on being middleware independence.

G^{en}oM provides to the programmer some built-in system primitives such as inter-process communication, datapool capabilities with external access, robust state machine for algorithms and automatic telemetry generation. The G^{en}oM[1] tool is embedded in the Openrobots architecture (LAAS Open source software for robots)[2] as a generator of software modules. Openrobots uses robotpkg as an infrastructure that manages automatically the compilation, installation, updates and dependencies. Robotpkg[3] provides an Open source repository with more than 200 packages from which around 120 are from LAAS.

* List of authors in alphabetical order.
1 http://www.openrobots.org/wiki/genom
2 http://www.openrobots.org/wiki/OpenRobotsFrontPage
3 http://homepages.laas.fr/mallet/robotpkg

2. GENOM ARCHITECTURE OVERVIEW

2.1. Introduction

G^{en}oM (Generator of Modules) [Fleury 1997] is a development framework that allows the definition and the production of modules that encapsulate algorithms. A module is a standardized software entity that is able to offer services which are provided by a set of algorithms. Modules can start or stop the execution of these services, pass arguments to the algorithms and export the data produced.

The algorithms are intended to be embedded into a target machine such as a robotic system. The developer of the algorithms might not be aware of the way the machine works as a whole and, most importantly, the algorithms will be integrated into a more general software system that includes algorithms developed by others. Yet, all these algorithms share several common properties: they must be configured, started, and scheduled among them, and we might expect them to exchange data and communicate with other parts of the system.

2.2. Description language

The generator of modules provides a description language, and standard templates. The templates allow the developer to describe a module; the services it can offer, and for each service, the list of expected parameters, the algorithms (organized into elementary code called codels) that will be executed, the results along with their description, the failure messages, etc.

With the template and the codels, G^{en}oM produces:

- ✓ a *complete module* that can run on several flavours of Unix or VxWorks;
- ✓ *interface libraries* that lets the developer use the services of the module and get back their results;
- ✓ an *interactive test program* that lets the developer send requests to the module and trigger the execution of the corresponding services.

It also provides access to those services by means of a standardized interface. Produced data in "posters" can also be retrieved in a standard way.

2.3. Development process

The G^{en}oM development cycle starts with the creation of .gen files. A .gen file describes with its own syntax all the internal database structures and the different types of procedures or requests. The G^{en}oM tool converts this skeleton into a well defined structure of libraries, makefiles and server libraries as shown in *Figure 1*.

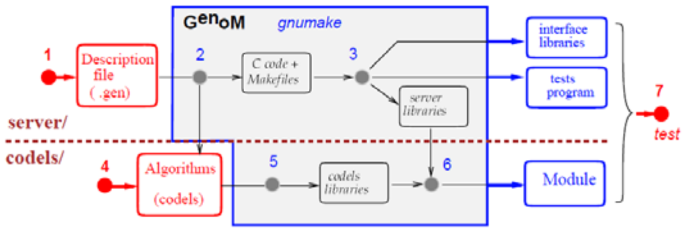

Figure 1. G^{en}oM development cycle

At this stage, the programmer is expected to incorporate the algorithms into the codels. Automatically generated code and codels are clearly separated from each other.

2.4. Tasks and requests

Each G^{en}oM module of the functional level is responsible for a functionality of the robot. For example, each basic sensor and effector is usually managed by its own module (one module for the camera pair, one module for the laser range finder, etc). More complex functionalities are encapsulated in higher level modules (e.g., a module doing stereo correlation will use the image taken by the camera module, and a module building an obstacle map will use the LRF scan). The most complex modules (such as navigation) can be obtained by having sets of modules "working" together under the control of a "supervisor." Each module can concurrently execute several services invoked by the supervisor. The module can send information regarding executed requests back to the supervisor, or share data with other modules using posters.

The services are managed by a control task responsible for launching corresponding activities within execution tasks.

The services are controlled (i.e., parameterized, started, or stopped) with requests, that represent the interface to the module. Requests will be sent by the operator using various interfaces (for instance, test programs generated by G^{en}oM, or the Tcl shell) or by a supervisor program (e.g., OpenPRS). Each request can have *input parameters* and *output parameters*. When a service ends, a reply is sent to the client who invoked the request. Each reply is associated with an execution report, which reports on the execution of the service and lets the client know about problems that might have occurred.

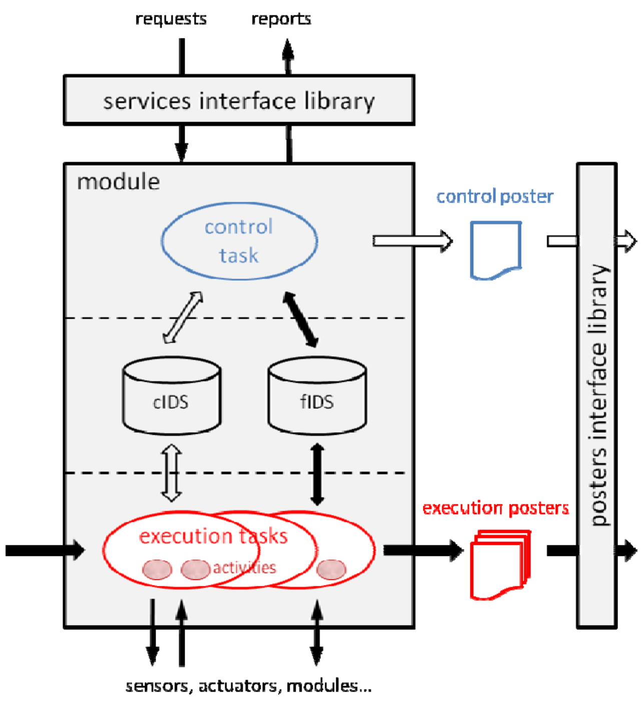

Figure 2. G^{en}oM module structure

There are two kinds of requests: execution requests and control requests (see *Figure 2*). Execution requests start an actual service, whereas control requests control the execution of the services. Control requests mainly allow to set parameters or interrupt services.

The execution time of a control request is negligible. Thus, they have only one ("final") reply, which is sent to the client upon completion. On the other hand, execution requests can last for an arbitrary amount of time (even indefinitely). Thus they send an "intermediate" reply, as soon as the service starts. The final reply is sent when the service terminates.

Besides this, the control task maintains a poster (the *control poster*) that contains information on the current state of the module (running services, activities, and so on).

Conflicts between services are handled in order to make the system as much reactive as possible, thus, if incompatible services are to be run at the same time, the last request received by the module has the highest priority and it interrupts the previous one.

Execution tasks are in charge of executing the user code. If this code is to be periodical (servoing, monitoring, filters, etc.), it is mandatory to use a periodical execution task and specify its period. It is also possible to use a-periodical tasks and a sequential scheduling. In this approach, tasks are given priorities (supported under real-time operating systems only) depending on their constraints in terms of resource and CPU requirements.

In basic G^{en}oM modules, we may only require one execution task (e.g., “MotionTask”). This is usually sufficient since a single execution task can manage several activities in parallel. However, if several activities require different priorities or time periods, we need to declare several execution tasks.

In order to associate user-supplied code with requests, the programmer must indicate which functions must be executed to handle requests. These user-supplied algorithms must be split into several parts: initialisation, body, termination, interruption, etc. Each of these elementary pieces of code is called a *codel*. At present, codels are C/C++ functions. A module is obtained by linking the G^{en}oM generated code and the codel libraries.

Finally, the G^{en}oM framework can produce two standard interface libraries in various programming languages (C, Tcl, XML, and OpenPRS as of writing), in particular,

- ✓ a service library, to send requests and manage their reports; and
- ✓ a poster library, which contains the necessary functions to read posters.

2.5. Activity state machine

A running service is called an activity. Some functions, such as monitoring services, can start several activities and execute them simultaneously. Other kinds of functions, such as servoing functions, cannot handle parallelism and can only start one activity at a time. This constraint has to be indicated by the developer.

The different states an activity can go through are shown in *Figure 3*. The external ring corresponds to the normal sequencing. START and END states are optional. On any transition, one can go into the INTER state. In case of a problem, one can go into the FAIL state, or even directly into the ZOMBIE state. The module is then frozen.

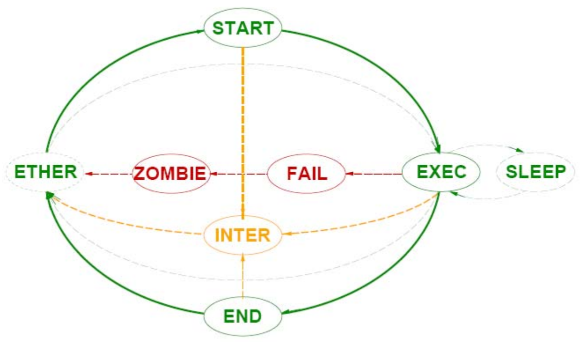

Figure 3. G^{en}oM codel state machine

Activities can control a physical device (e.g., sensors and actuators), read data produced by other modules (by means of posters) or produce data. The data can be transferred at the end of the execution through the final reply, or at any time by means of posters.

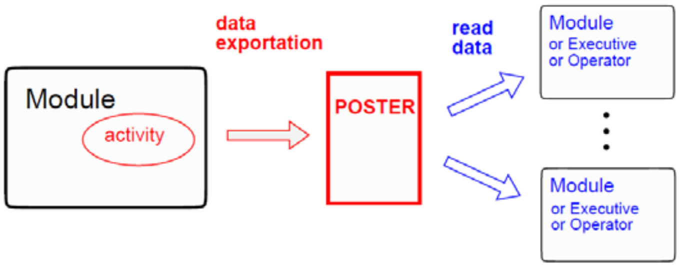

Figure 4. Data exportation through the use of posters

A **poster** is a data structure that is updated by an activity and shared in the global system. It can be read by any other component of the system (a module, a supervisor, etc.) at any time. All internal data are stored in an internal structure, called the *Functional Internal Data Structure* (fIDS), which is defined by the user at the beginning of the .gen file.

There is also a *Control Internal Data Structure* (cIDS), which is internally used by G^{en}oM to record the status of activities, request execution, etc. This structure is defined automatically by G^{en}oM, but can be accessed by the user if needed.

3. GENOM3

As of today, G^{en}oM generates code for the pocolibs middleware [Posix COmmunication LIBraries[4]]. This middleware is targeted toward embedded real-time systems and has a very small overhead. In some circumstances, it is interesting to easily target another middleware, while reusing the same components and ideally without changing a single line of code in the components’ codels.

This middleware independency is useful in the following cases:

- ✓ simulation where the real-time constraints may be relaxed.
- ✓ software verification and validation, where the middleware is replaced by formal tools.
- ✓ targeting another robot running with a different system.

Since the year 2000, a lot of robotic middlewares have been developed, such as Orocos, YARP, and the latest ROS. This reinforces our conviction that generating components that are truly middleware independent is important for being able to track down and quickly adapt to the latest evolutions in this domain.

[4] https://softs.laas.fr/openrobots/wiki/pocolibs

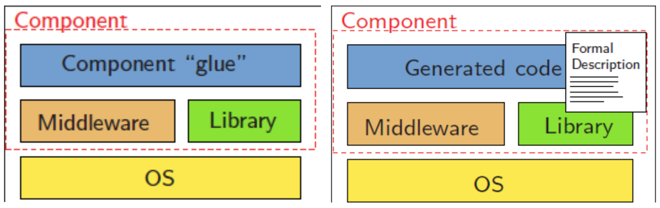

Figure 5. A component architecture realizing a clear separation of concerns between a middleware and a library; the ``glue'' code grants the decoupling (left). The G^{en}oM3 tool generates the glue code from a component's formal description and a skeleton (not shown in the figure) suited to the middleware (right).

The upcoming version 3 of G^{en}oM [Mallet 2010] will be able to generate fully middleware-independent components, due to the component architecture described in Figure 5. The algorithmic core (the set of codels implemented in a "library" in *Figure 5*) is made independent of middleware by using glue software linking the two pieces together. Instead of making direct calls to the middleware, component functions in the library formally describe the input or output objects they use.

The "glue" code is responsible for making the necessary calls to the middleware and passing (or retrieving) the desired objects to (or from) the library's functions. With the help of the formal description provided by the .gen files, the glue code can be generated from generic templates that can be easily replaced for every middleware that is used. Therefore, the libraries do not contain any references to any specific middleware. To achieve this goal, a few things had to be changed in the .gen syntax. First, data types must now be described with a subset of the OMG IDL language, to provide programming language independence. The mappings from IDL to the actual programming language are strongly inspired from the OMG standard but slightly adapted to provide type definitions that do not require memory management, and to keep the ability to generate real-time code. Second, codels had to be more precisely described so that the corresponding functions do not have to use specific middleware functions to retrieve external data such as posters or access the IDS transparently. All data is now passed as parameters to the codels, and a component's generic template is responsible for the codel invocation with the proper set of parameters.

The core component generation in G^{en}oM3 is implemented with a generic code generation engine. The engine generates code by using the .gen description file and a skeleton, dedicated to a particular middleware, that is instantiated according to the .gen description file. The skeleton is not part of the G^{en}oM tool anymore but provided as an external package so that any kind of skeleton can be easily developed. Such skeletons are called "Component Templates."

A component template is organized as a regular set of source files, with special markers that are replaced (in a manner similar to how a PHP script is embedded into an HTML page) according to the .gen description file. It implements the internals of a component with classical primitives such as threads or semaphores and takes care of the communication aspects such as remote procedure calls and data marshalling. In short, a template contains all the source code that is not part of the algorithmic core of any specific component. The key for middleware independence is thus to develop different templates that provide an implementation for different middlewares or even different component architectures. Since all templates will implement the same .gen component model, testing alternative templates is simply a matter of recompiling existing components with a new template.

The G^{en}oM3 tool is still a work in progress. As of today, the tool itself is almost functional and a pocolibs template has also been developed. This template provides the same functionality as what G^{en}oM2 used to provide, so that G^{en}oM2 and G^{en}oM3 components can coexist. During the next months, new templates will be developed. One option is to provide a ROS template, which will generate ROS-compatible components. Another direction is to develop a BIP template (see section 4) to allow software verification and validation.

4. GENOM AND THE BIP FRAMEWORK

Apart from extending G^{en}oM to be middleware independent, we have also looked at making the functional layer developed with G^{en}oM more robust and verifiable [Bensalem 2010]. In particular, we have recently started an effort to integrate G^{en}oM with a component based framework called BIP, used for implementing embedded real-time systems. We address the problem of using formal methods for developing modules of the functional level of robots and satellites.

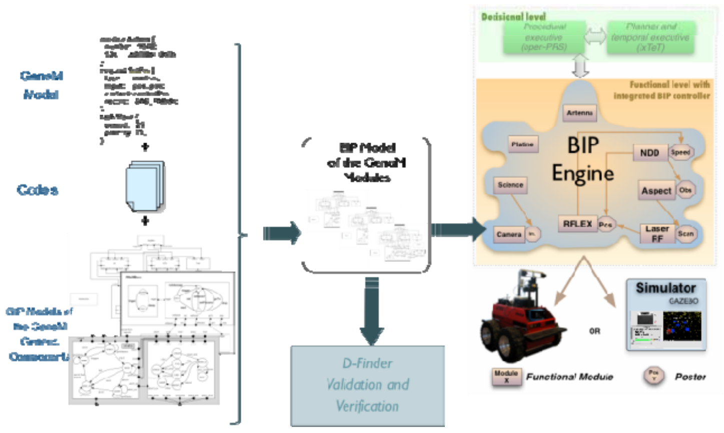
GenoM Model
Codes
BIP Models of the GenoM Generic Components
BIP Model of the GenoM Modules
D-Finder Validation and Verification
Decisional level
Procedural executive (oper-PRS)
Planner and temporal executive (IxTeT)
Functional level with integrated BIP controller
Antenna
Platine
BIP Engine
NDD
Speed
Science
Aspect
Obs
Camera
RFLEX
Pos
Laser RF
Scan
OR
Simulator
GAZEBO
Module X
Functional Module
Poster

Figure 6. BIP engine over G^{en}oM framework

To this end, we have successfully developed the BIP/GenoM component based design approach and applied it on the functional level of a complex exploration rover (see Figure 6). Using this approach,

- ✓ we produce a very fine grained formal computational model of the robot's functional level;
- ✓ we run the BIP engine on the real robot, which executes and enforces the model at runtime; and
- ✓ we are able check the model offline for deadlock freedom, as well as other safety properties using formal tools such as D-Finder.

5. TCL AND OPENPRS

Modules written in G^{en}oM provide a set of services and posters. Yet, modules need to be driven by some supervisor. To this end, the Openrobots framework provides a number of ways to write a supervisor controlling a set of G^{en}oM modules. Indeed, you can write a supervisor in C/C++ and link it to the various automatically generated server libraries of the different modules. However, most applications require some higher level language to perform this control. To this end, G^{en}oM provides an API to send requests from Tcl to G^{en}oM modules. Using this API, one can write a supervisor with Tcl scripts and control the various G^{en}oM modules. For applications requiring supervisors that incorporate certain AI techniques (e.g. Prolog like logic programming), G^{en}oM provides also a binding with OpenPRS. OpenPRS [Ingrand 2007] is an open source version of PRS (Procedural Reasoning System), used as a decisional layer component in the LAAS architecture. PRS provides the system with the ability to reason in complex ways about dynamic processes while still maintaining the reactivity required to ensure appropriate responsiveness and control.

6. APPLICATION TO LAAS ROVERS

Since 1996, the G^{en}oM modules have been present on all the robots at LAAS, such as Rackham, an autonomous interactive mobile robot, Jido (see *Figure 7*), a fully equipped mobile robot with a manipulator, HRP-2, a full-size humanoid robot, and Dala and Mana, two rough terrain mobile rovers. Some of the resulting functional layers are quite complex, with as many as 30 modules (easily reusable on other robots), and may span over 2 or 3 CPUs.

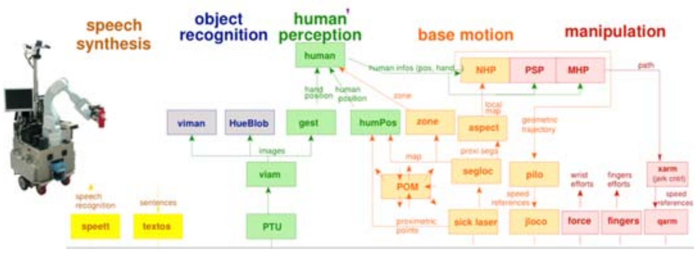

Figure 7. GenoM modules of the Jido robot

7. APPLICATION TO A SATELLITE

G^{en}oM has also been used to implement the functional layer of an observation satellite similar to PROBA (see Figure 8).

This was initially started in the Sydre Project with Astrium in the late 90s, and has been revived in the MARAE project. The two projects were intended to show that one can use a tool such as G^{en}oM to control a satellite, and by extending G^{en}oM with BIP (see next section), to show that the resulting functional layer could enforce some safety constraints on the behaviour of the modules.

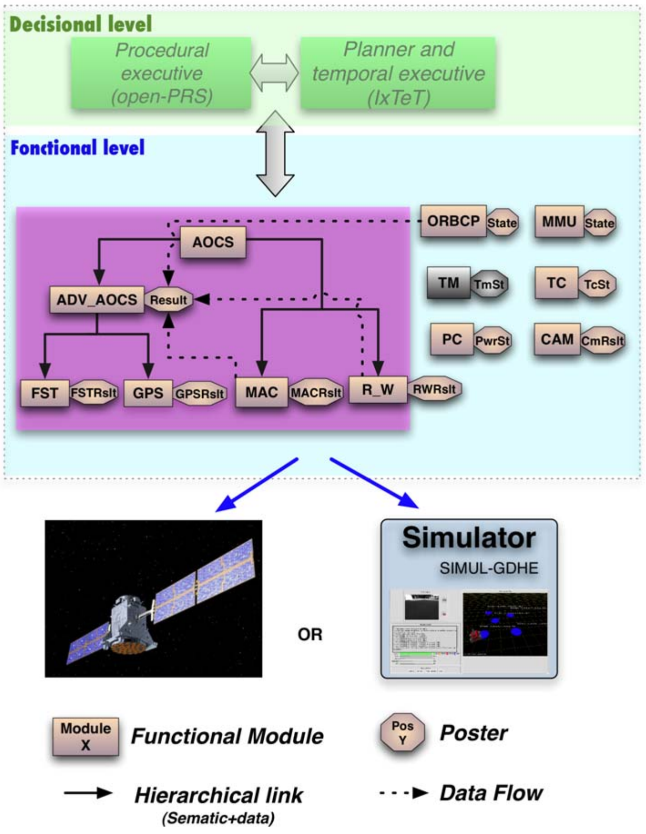

Figure 8. G^{en}oM modules of the satellite simulator

8. APPLICATION BY GMV TO SPACE RELATED SYSTEMS

Peculiarities of space operation pose additional challenges to system design and operation. Space systems have to work on harsh environmental conditions requiring conservative procedures to preserve safety. They typically can count on limited computational and energy resources. There is then a need for high efficiency and scalability. Other aspects typical of the space environment are the presence of low bandwidth communication channels and time-lagged operation.

G^{en}oM is ideally suited to meet space requirements because of its limited overhead, efficient communications and complexity partitioning characteristics. Although G^{en}oM has been used by several institutions external to LAAS in the past, it is only recently that other industry partners have begun to adopt it as a framework for space robotic systems.

Several prototypes specifically designed for a scenario of planetary surface exploration are being engineered by GMV using G^{en}oM software components. The MoonHound rover, a 3DROV based ExoMars-like virtual rover within the ESA GOAC project, and LRM (as a recently initiated ESA activity) are examples of this.

The MoonHound rover is a 4-wheel differential rover for research in space mobility and perception with a special focus on navigation, localisation and mapping algorithms. The functional layer of the MoonHound rover as shown in *Figure 9* has been completely built using G^{en}oM components to obtain a high performance, low level control and perception system.

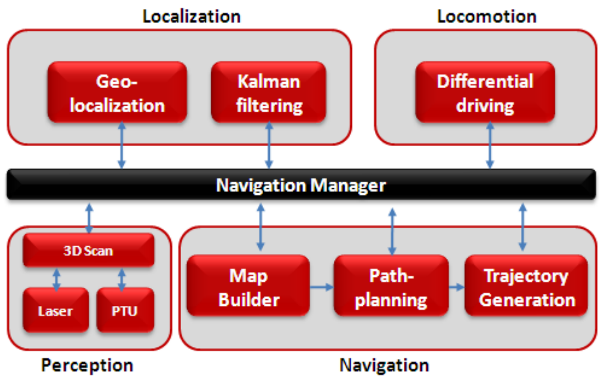

Figure 9. MoonHound rover G^{en}oM modules

Another ongoing project using G^{en}oM components in a rover controller is GOAC. It aims at building a Goal Oriented Autonomous Controller for a variety of space robotic systems. The GOAC executive and decision algorithms will be concept-proofed on a real rover and a simulated rover.

The GOAC system is being implemented in two different scenarios:

- ✓ Outdoor differential rover ATRV-based named DALA at LAAS premises using a stereovision system and a Sick laser as perception sensors. The GOAC controller interleaves planning with execution while performing activities such navigation 2D/3D, images acquisition, communications and continous monitoring (environment, platform heating and power comsumption). The BIP engine enforces constraints already in place (laser-based navigation, moving/comunicating) and new ones at system level (temperature and power/energy check, data freshness, dead-lock freedom , etc.).

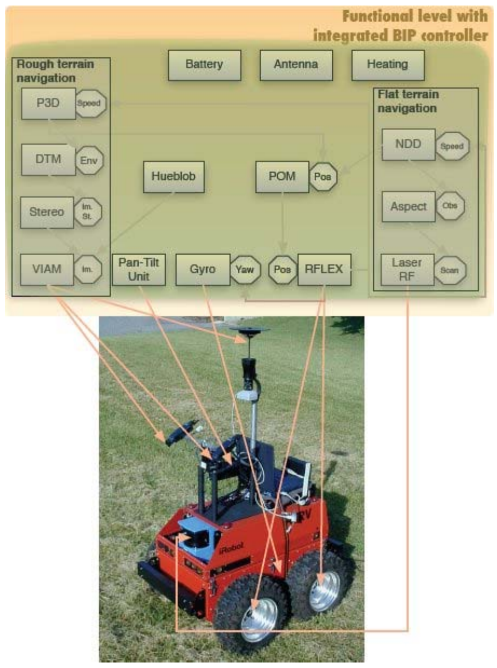

Figure 10. GOAC scenario using the DALA rover at LAAS premises.

- ✓ Mars scenario navigation based on a 3DROV modelled ExoMars-like rover (see *Figure 11*). Both rovers share a similar G^{en}oM functional layer capable of performing stereovision based navigation.

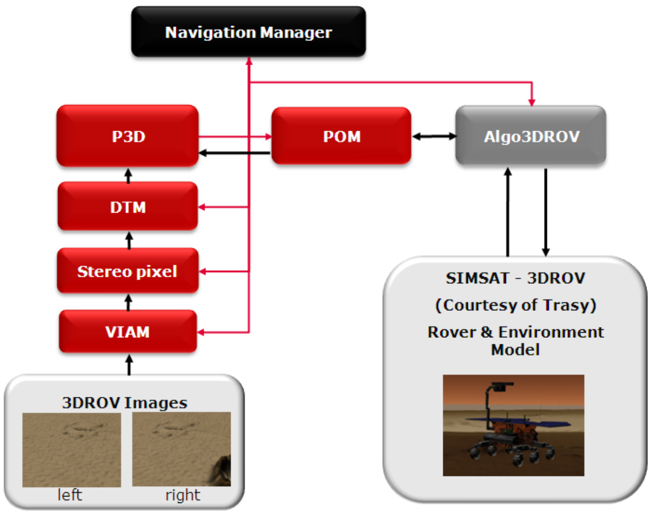

Figure 11. GOAC Stereovision tool-chain.

3DROV [Poulakis2008] is a simulation environment for planetary exploration rovers. It includes models of the planetary environment, the mechanical, electrical and thermal subsystems of the rover, a generic on-board controller, and a ground control station module. The simulation framework relies on ESA's SIMSAT 4.0 and the models are compliant with SMP 2.0. The framework offers the ability to attach on-board algorithms for testing

The integration of GOAC in the 3DROV involves modifying the Generic Controller of the 3DROV and interfacing it with the G^{en}oM functional layer. In order to do that, a software library making available an interface to the algorithms of the Generic Controller has been developed.

Recently an activity aiming at renovating ESA's Lunar Rover Mockup rover into an autonomy investigation platform has been started.

A G^{en}oM based functional layer will be in charge of the control of the LRM rover (see *Figure 12*). It will be built from distributed, specialised components in charge of performing tasks related to perception, maps generation, path-planning localisation, locomotion and safety monitoring.

Given the number of tasks and elements interacting in the functional layer, the powerful encapsulation and data interchange mechanisms provided by G^{en}oM will be used. By specifying standardized ways to interpret posters information as well as specifying default requests for certain types of modules, interchangeability and relocation of modules will be supported.

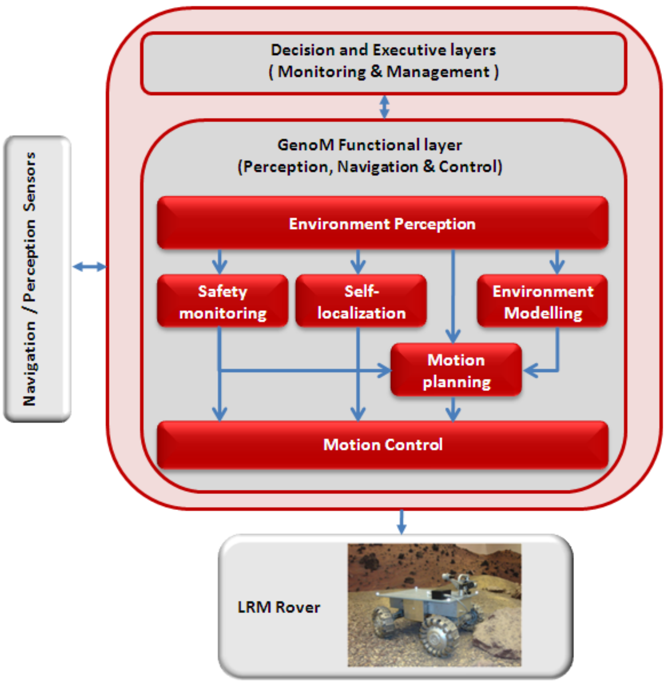

Figure 12. LRM Functional Layer

9. CONCLUSION

Space systems such as planetary rovers and satellites have to meet stringent safety and performance requirements. The need for more on-board autonomy in a context of growing complexity poses significant challenges to system designers.

The G^{en}oM framework described in this paper provides a number of capabilities to address those needs efficiently. It is thus ideally suited for space robotics systems development. With the upcoming G^{en}oM3, independence of the used middleware will be a reality. Some capabilities worth highlighting include:

- Modular design with proper management of component dependencies.
- Built-in support for supervision and inter-module communication using publicly accessible service requests and data posters.
- Focus on algorithmic development, and automated generation of code for management of low-level software details.
- Support for methodological correctness through state machine based algorithms and control procedures.
- It also enforces engineers to design modules in a specific way, which helps to guarantee key characteristics of a controller.
- Enables integration with BIP for implementation of correct-by-construction software systems.
- Automatic generation of libraries for interfacing a module from a program written in C, Tcl or OpenPRS, as well as an automatically generated test tool.

Finally, the framework has been extensively tested in a variety of research robotic prototypes. Lately, it is being used in several ESA projects for the next generation of autonomous rovers and other robotic systems.

ACKNOWLEDGEMENT

Part of this work has been funded by the ESA GSTP GOAC and RAT projects and the FNRAE/ MARAE project.

REFERENCES

[Mallet 2010] GenoM3: Building middleware-independent robotic components. A. Mallet et al. *IEEE ICRA, Anchorage, AK (USA), 2010*.

[Fleury 1997] S. Fleury, M. Herrb and R. Chatila. GenoM: A Tool for the Specification and the Implementation of Operating Modules in Distributed Robot Architecture. *IEEE-IROS 1997*. Grenoble, France.

[Bensalem 2010] Saddek Bensalem, Lavindra de Silva, Matthieu Gallien, Félix Ingrand, and Rongjie Yan. "Rock Solid" Software: A Verifiable and Correct-by-Construction Controller for Rover and Spacecraft Functional Levels. *i-SAIRAS 2010*, Sapporo, Japan.

[Ingrand 2007] Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier and Fredéric Py. *Decisional Autonomy of Planetary Rovers, Journal of Field Robotics, Volume 24, Issue 7, July 2007.*

[Poulakis2008] P. Poulakis, L. Joudrier, S. Wailliez, K. Kapellos. 3DROV: A Planetary Rover System Design, Simulation and Verification Tool. *iSAIRAS 2008.*