Simulation in the LAAS Architecture

Sylvain Joyeux, Rachid Alami, Simon Lacroix, Alexandre Lampe

Simulation in the LAAS Architecture[1] (Extended abstract)

Sylvain Joyeux, Alexandre Lampe, Rachid Alami, Simon Lacroix
LAAS/CNRS,
7 avenue du Colonel Roche, F-31077 Toulouse Cedex 04
Email: firstname.lastname@laas.fr

Abstract

While software for robotics cannot avoid being tested for real, it is often difficult and time-consuming to do so. In multi-robot developments, one may be unable to fully experiment its software since putting an experiment into place for a few aerial and terrestrial vehicules can quickly become difficult to do. In these cases, being able to extensively test in simulation is a very important thing.

We present here a simulation environment adapted to the LAAS Architecture[1]. This architecture is designed for the control of autonomous robots. The simulation system has been designed to address the major issues in this kind of problems:

- being able to use the same code in both simulated and real environments.
- being able to reproduce the behavior in non-simulated systems: sensor accuracy, execution latencies in communication (inter-robot and hardware-to-software).
- scalability in terms of robots and world size for multi-robot simulations.

We present first a quick overview of the LAAS architecture, then the simulation system, and how we did achieve to address the three points outlined above.

I. The LAAS architecture

The LAAS Architecture[1] for Autonomous Systems is a now classical 3-layer architecture.

- the low-level layer is a functional layer based on the GenoM tool[2]. In GenoM modules, there are two means of communication: it is either message-based (mailboxes), or shared-memory based (posters). The former is used for commands and reports and the latter is used as a mean to export the state of each module and/or the result of the algorithms it encapsulates.
- the execution control layer has two parts: a safety-bag controls the behavior of the system and the executive itself does low-level control of the functional layer. The safety bag is generated using the ExoGen tool[3] and execution controllers are developped using Tcl for the simplest and OpenPRS[4] for the more complex ones.
- decisional layer implementations are more diverse. The simpler ones are written using OpenPRS as a way to react to events from the environment. More complex ones may use IxTeT, a temporal planner, coupled with eXeC, its associated execution controller. The IxTeT-eXeC software is able to repair and/or replan if needed.

All these tools are either currently available as free software on the LAAS OpenRobots site[2], or will be in a near future. The source code of the simulation system itself will be available[3].

II. The simulation architecture

There are essentially three parts in the simulation architecture:

- a low-level layer controlling the module execution, which makes possible to control the time flow. Basically, it allows to modify the execution time of one thread *as it is perceived* by the other threads while keeping the overall behavior of the system. This can be done without modifying the GenoM modules at all.
- a world server, which simulates the world physics (collision, forces) and sensing.
- a bridge. The bridge is the central part of this system:
  - it maps values in the set of modules and in the world server, which makes possible to interface with other simulation systems.
  - it federates the time management of every modules in one computer.
  - in distributed simulations, it synchronizes every machine used in the simulation.

[1] LAAS Architecture for Autonomous Systems
[2] `http://softs.laas.fr/openrobots`
[3] See `http://www.laas.fr/~sjoyeux/simufornow`

A. *Discrete event simulation model*

We will present shortly the simulation algorithm we use in our system. It is based on the PDES problem, common in the simulation world. Refer to [5], [6] for a more complete description of this problem and the existing solutions.

In the simulation litterature, a discrete event simulation is based on a model which changes its state at discrete points in simulated time. In turn, Parallel Discrete Event Simulation (PDES) refers to the execution of a single discrete event simulation program on a parallel computer. The heart of the PDES problem is to split the single event model into more than one Logical Process which exchanges timestamped messages. The problem is to ensure that during the execution, events are always executed in timestamp order. This constraint is known as *the causal constraint* in the simulation litterature.

We present now how we made GenoM-based software fit into this model and then what simulation algorithm we used to ensure that the causal constraint is satisfied.

Like most robotic control software, the GenoM modules highly rely on multi-threading. In this kind of software, a thread behavior is defined by the current state of all its variable and where it is currently in the execution flow. In a multithreaded program, or multiprocess with IPC, we can split each of these states into:

- an *internal* state, which is its current instruction and the data it does not share with others.
- an *external* state, made of the state of synchronization objects and the data it writes and/or reads through communication objects (pipe, sockets, shared memory, variables shared amongst multiple threads, ...). Since in general every thread has access to these objects, we can assume that multiple threads share at least parts of their external states.

During its execution, each thread has its own Local Virtual Time (LVT) [6], which depends only on its own execution flow. On the one hand, by definition, there is always a match between one thread's internal state and its LVT. On the other hand, the external state is - in general - shared and thus we should ensure that it remains coherent. For that, any modification of the external state shall be controlled through a set of well-defined events. We will then be able to schedule the execution of these events using PDES algorithms.

Fortunately, in well-formed multithreaded software, any modification of global state of each thread shall be protected by use of a few inter-thread primitives:

- synchronization objects (mutexes, signals, semaphores).
- communication objects (pipes, sockets). Note that since an access to shared memory has to be protected, we can manage it *via* the synchronization objects the access will need.

A successful simulation is one where we can ensure that the modifications of the global states are made in LVT order. This is the same as ensuring that all uses of the inter-thread primitives are made in timestamp order. So, in our simulation model, the events are simply the use of these primitives.

Since there is in general no constraint on each thread's local virtual time, it becomes easy to simulate the latency of a simulated access to hardware by offseting the logical time of the thread doing the access. Moreover, we can test algorithms as if they were faster and/or slower than they are in reality. Under some limitations, we can also use debugging tools on the software like source-based debuggers without changing the overall behavior of the system.

B. *Event management in a GenoM module*

The GenoM modules[2] are based on one command task and one or more execution tasks. The command thread interprets the requests coming from the outside of the module and sends replies and reports to inform on the ongoing execution of the module services. The execution tasks interpret the command sent internally by the command task and reacts accordingly. Mostly, it consists in the execution of C functions implementing the various services provided by the module.

In a module, the low-level layer wraps the function calls related to the inter-thread primitives and adds a waitpoint just before they are used. The waitpoint is an event which blocks the thread creating it until the event management algorithm allowed its execution. Since the events can be generated anytime during the tasks execution, we need a function which provides the local time of the thread at *any time*. Time functions currently in use are:

- one where the progression of local time and wall-clock time are proportionnal.
- one where a task loop execution is done in a fixed amount of time. It would be equivalent to a hard real-time (RT) model: in hard RT systems, the specification provides an upper bound for the execution time and the software is supposed to guarantee the behavior of the system as long as the execution time obeys this specification.
- one based on the real CPU usage of the calling thread.

Moreover, a few things had to be taken care of in order to fully integrate the software:

- the standard calls for getting time (`gettimeofday` for instance) are wrapped in order to return the logical time instead of the machine time.
- time is also shared implicitely by use of *timers*. The timer implementation we use in GenoM is semaphore-based. In order to manage it in the event algorithm, we use a recurrent event using the period of the timer.

Events are then managed by the algorithm 1. This algorithm is based on the centralized event list approach of PDES. It ensures that when an event is executed at a global time t:

- each other thread has a local time greater than t.
- for each other thread, the last modification of its external state has been done before t.

```
Data:
  $t$ := $0$, the global time
  $events$ := $\emptyset$, the set of events: wait points, timers
  tasks := the set of tasks on the system
begin
    while true do
        wait(For all tasks to have put one event in $events$)
        $t_{min}$ := min(t($events$))
        advance_time($t_{min}$)
        execute_event(events_at($t_{min}$))
    end
end

Where:
  • t($i$) returns the logical time associated with $i$
  • advance_time(new_t) is defined as
    function advance_time(new_t)
    begin
        t := new_t
        wait_authorization(new_t)
    end
```

Algorithm 1: Event management algorithm

```
Data: $t_l$, $t_g$ the local and global time
function wrapped_primitive(arguments)
begin
    wp = wait_point($t_l$)
    wait(wp)
    call(the original implementation)
    if infinite timeout then $t_l$ := $t_g$
end
```

Algorithm 2: Modified call to an primitive `wrapped_primitive` which needs synchronization

C. Managing blocking calls

Since some primitives can block, their execution time is non-null (i.e. the calling threads spend time sleeping during the call). Therefore, we have to compute the local time *after* the call given the global time and the local time before it.

There are two cases: on the one hand, the call does not block and the function call behaves like a non-blocking primitive. On the other hand, it does block, and the thread returns in running state when another thread uses the corresponding functions (`pthread_mutex_unlock`, `pthread_cond_signal`, ...).

In the blocking case, when the two threads interact, they share the same local time. Because of the way we have modified the function calls, it happens that this shared logical time is also the global time. Therefore, we can say that *just after the call*, the sleeping thread shall initialize its logical time to the global time.

The full waiting procedure is outlined in algorithm 2.

Adding non-zero finite timeouts is not so simple. We have thought of a few options, but none looks really good:

- First, add a waitpoint at the timeout end and make the calling thread wait for it. Second, make a different thread wait for the synchronization object. If the timeout event occurs, the separate thread will force the waitpoint release itself. Otherwise, the waitpoint will release the calling thread at the logical time of the timeout. In the second case, we will have to clean up when the synchronization object is actually released (if ever). The main drawback is that this solution needs a new thread for each call with timeout.
- use the real timeout but check if the global time matches. If the timeout released the thread too early, call back with a modified timeout. Note that there is a potential race-condition here.

Since there is no such timeouts in GenoM, no support for them has been implemented.

D. Plugging the layer on a GenoM module

The execution and control task are made of automatically generated code (GenoM stands for Generator of Modules) from a high-level specification file. The only part not generated is the functions implementing the module services (which are called *codels*). The fact that the control part of the module is automatically generated helps us to ensure one of the design goals of the simulator: since we can rely on some behaviour and common data structures, we are able to plug and configure the simulation layer without even recompiling the module. The low-level layer is only a shared library which is preloaded at the module execution. It is then able to plug itself in known points during the module initialization.

III. The bridge and the world server

Three components are used to provide the missing parts of this simulation system:

- inter-module synchronization.
- multi-robot functionalities.
- simulation of the physical world.

A. Putting the modules together

A robotic system is - of course - made of more than one GenoM module. In the simulation system, another event algorithm is put inside the bridge in order to synchronize a set of modules. In the algorithm 1 we have put a `wait_advance` call which ensures that the bridge has control on each module global time advance. The bridge simply allows the execution of the module with least global time. We can notice that corresponds to the naive approach in PDES. But it is interesting in two ways in our case:

- it is simple.
- at first, we did not intend to split a mono-CPU software system across multiple CPUs in simulation to speed up the simulation. In that case, even this algorithm does not waste computing power.

B. Simulation of the physical world

The world server has to provide the physical world simulation: vehicule control, collision checking, sensing, ... We are currently using the Gazebo simulator [7]. However, the simulation architecture tries to be as world-server independent as possible.

One of the constraints for this simulator to be useful is that its integration step can be as small as the needed grain in the real world dynamics. In general, the time step should be less than the smallest period of the simulated software. It is not always needed to have very high precision in this component. We may want to simulate the world roughly when testing the behavior of some decision layers, and in that case an approximate simulation of the world physics, with bigger time steps, is more than enough.

As stated earlier, one of the design constraints for the architecture was to be as much world-server independent as possible. Because of that, the GenoM modules do not access the world server directly. They read and write posters - a structure used in GenoM for inter-module communication - which are then mapped from and to the world server by the bridge. The bridge does the convertion between the data structures used on the genom side and on the world-server side. If the world server is changed, we do not have to change the genom modules, only the mapping in the bridge. The example presented later in IV-A illustrates this.

C. Multi-robot simulation

There are essentially two issues for multi-robot simulation:

- the physical world has to be shared amongst the robots.
- robots can communicate.

Even if we did not consider the use of this system to split a mono-CPU robotic architecture across multiple hosts to speed up the simulation, we wanted to use multiple hosts to run multi-robot simulation since it is necessary in order to be able to achieve scalability in terms of simulated robots in the world. The problem here is the synchronization of a shared world on multiple hosts. There are two solutions:

- use a world simulator that supports distributed simulations.
- use a mono-host world simulator like Gazebo and externally synchronize the state for each robot. Synchronization takes place at each simulation step. To speed up the process, one can consider only attributes that can be perceived by other robots. For instance, while the position shall be synchronized, we do not have to do it for a range finder state.

For now, it is the second solution that has been implemented. The multihost synchronization is done by CERTI [8], a free software HLA RTI, in which the RTI services take care of both data distribution and time synchronization.

We can notice that sharing of the physical world takes care of implicit communication between robots. Still, explicit communication shall be solved by other means. This can be solved by wrapping communication primitives like sockets, pipes, ... which is not done yet. On the one hand, their integration in the time control algorithm should be painless. On the other hand, we will need media models in order to emulate the real communication behaviour. These models could provide:

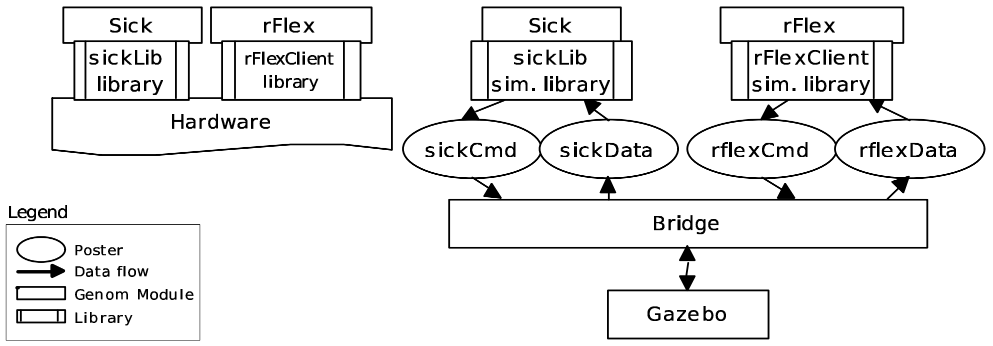

Fig. 1. Modules that access hardware in the real robot (left) and in the simulation system (right). See IV-A for a detailed description of each module role

- a communication latency model. In order to simulate latency, we will need information on how long the packets take to travel. If the operating system does not provide this information, we may have to add in the data the information regarding when it was sent.
- a packet drop model for datagram-based networking. For obvious reasons, we cannot do that if it is stream-based. The drop will take place when the data is sent, not when it is received, to avoid sending useless data on the network.

Note that we do not plan to do full network simulations though - which is an area of research in both the simulation and communication tools communities. We only want to get a rough estimate of the contingencies regarding communication that the robots will have to cope with.

D. Interactive simulations

Of course, the simulation user is able to interact with both the world server and the simulated software. However, for simulations that are not nearly real-time, a human-software interaction should be difficult since it is possible to have gaps in the simulation time. Currently, the simulation layer performance makes it possible to have the logical time following the real-time, making such interactions possible.

We can notice that more generally this remark applies to any interaction between simulated systems and non-simulated ones, for instance a simulation where real robots interact with simulated robots.

IV. Integration and testing

A. Current implementation and simulation example

An implementation is currently in use at LAAS. While multi robot simulations have only been tested with simple simulations for validation and regression testing, mono-robot has been more extensively tested. In particular, the integration of modules running on real robots has been successful. We present now the instanciation of the architecture in the experiment and the way it has been integrated in simulation. You might want to refer to I for a quick presentation of GenoM-related vocabulary.

A comparison of the low-level module architecture in the real experiment and in the simulation system is detailed in figure 1. In this experiment, only two modules are connected to the hardware:

- the `rFlex` module is connected via a serial line to the microcontroller which controls the robot motors and sonars, and reads data like odometry and battery level.
- the `sick` module controls a laser range finder, writes its readings in a separate poster and can optionally do segment detection and put these segments in a poster. These segments are used later in the `segLoc` module for localization and mapping.

Of course, these modules had to be adapted. Since both use separate libraries for hardware access, we replaced these libraries by implementations which communicate with the world server through the bridge. For that, we defined two posters by device: one for sending commands to the device and one for reading data back from it. One these two modules have been adapted, it has been straightforward to get the other modules working.

The robot position is generated by the collaboration of a few modules:

- laser range-based SLAM algorithms are put in the `segLoc` module. The module does SLAM in a segment-based 2D map.
- position integrated from the robot odometry is produced by the `rFlex` module.
- the `POM` module does the fusion and filtering of these positions and produce the true position of the robot.

Finally, the robot position is used by the `NDD` module for navigation. This module uses both the laser ranges and the filtered position for navigation and obstacle avoidance.

All modules but `rFlex` and `sick` have not been modified at all. These last two modules have been slightly modified because of their links with the hardware. These modifications are limited to the hardware libraries though.

This experiment will allow us to compare the behaviors of the simulated and real robot.

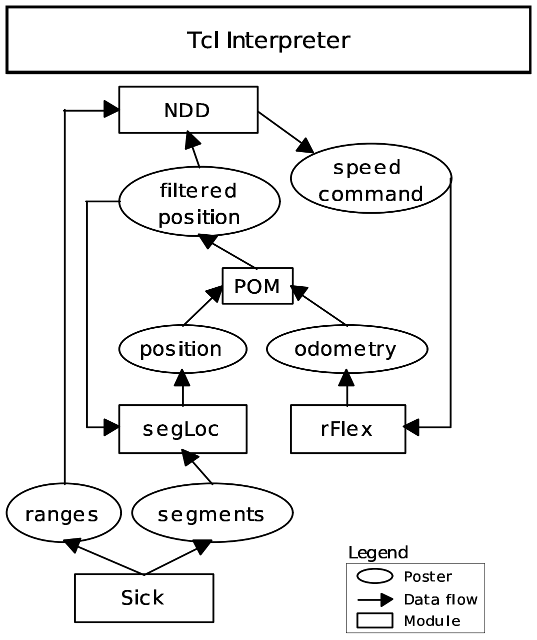

Fig. 2. Module architecture for navigation. See IV-A for a detailed description of each module role

B. *Integration of the upper layers*

The activity of the upper layers of the LAAS architecture is mostly based on two things:

- reactions to events coming from the lower layer.
- progression of time.

Since the GenoM layer events are already managed by modules' time control, the events are not a problem. The timeouts could be one. However, since all of the tools we use always read the time explicitely after a timeout, wrapping the library calls related to time (`gettimeofday` for instance) should be sufficient.

Notice that a full test of these principles has not yet been done.

V. Related work

In the simulation community, discrete event simulations, and more particularly PDES, is a very active field of research [5], [6], [9], [10]. However, our problem is much simpler than the general PDES problem since we can exploit the native parallelism of the multithreaded software and the associated synchronization points.

A few standardization efforts exist for communication in distributed simulation applications. In one hand, the Distributed Interactive Simulation standard has been mostly used in military training simulation. It provides information distribution but no time control service. On the other hand, the High Level Architecture (HLA, [11], [12]) provides a generic framework for simulations and allows the integration. Both DIS and HLA are IEEE standards (standards number 1278 and 1516 respectively). While other efforts have been put into developping generic simulation frameworks, this in our knowledge the only common standard in this field.

Another aspect of the simulation research is adding real-time constraints [13] to the simulation systems. Mainly, the goal here is to provide simulation with hardware-in-the-loop, or complex simulations which will be able to interact tightly with non-simulated parts. While in our work it was not a basic requirement, we may want to test mixed real-robots/simulated-robots environments in future simulations.

In the field of robotic research, the Player/Stage project [7], [14] aims at providing a package for both robot control and simulation of the control software. The Player software is a hardware abstraction layer with a communication model that allows to control a robot without caring about the real hardware. Moreover, the server nature of a Player program makes easy the distributed control of the hardware. In particular, this allows to replace hardware with a simulator. In particular, Stage provides a 2D simulator while Gazebo is a 3D simulator. However, there is no effort in this project to provide a time control layer, and the Player project does not consist of a full architecture like the LAAS Architecture does.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," 1998. [Online]. Available: citeseer.ist.psu.edu/alami98architecture.html

[2] 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 *Proceedings of the International Conference on Intelligent Robots and Systems IROS’97*, vol. 2, pp. 842–848.

[3] F. Ingrand and F. Py, “An execution control system for autonomous robots,” in *IEEE International Conference on Robotics and Automation*, Washington DC (USA), May 2002.

[4] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert, “PRS: A high level supervision and control language for autonomous mobile robots.”

[5] R. M. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[6] Yi-Bing Lin and P.A. Fishwick, P.A., “Asynchronous parallel discrete event simulation,” in *IEEE Transactions on Systems, Man and Cybernetics, Part A*, Jul 1996, vol. 26, no. 4, pp. 397–412.

[7] “The player/stage project,” http://playerstage.sourceforge.net/.

[8] B. Bréholée and P. Siron, “CERTI: Evolutions of the ONERA RTI protoype.”

[9] D. Nicol and R. Fujimoto, “Parallel simulation today,” in *Annals of Operations Research*, 1994.

[10] D. R. Jefferson, “Virtual time,” in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1985, vol. 7, pp. 404 – 425.

[11] S. I. S. Committee, “Ieee standards for modeling and simulation high level architecture,” 2000, iEEE Standard 1516.

[12] R. M. Fujimoto, “Zero lookahead and repeatability in the high level architecture,” 1997 Spring Simulation Interoperability Workshop, March 1997.

[13] M. J. Patrick, “Data communication infrastructure for distributed real-time simulations,” Master’s thesis, Univeriteit van Amsterdam, August 2004.

[14] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard, “The player/stage project: Tools for multi-robot and distributed sensor systems,” in *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, June 2003, pp. 317–323.

[15] S. Joyeux, “Architecture distribuée de simulation multi-robots, rapport de dea,” 2004, http://www.laas.fr/~sjoyeux/doc/sim-multi-robots.pdf.