Recent developments of the G^{en}oM robotic component generator

Anthony Mallet, Matthieu Herrb
LAAS/CNRS
7 avenue du colonel Roche
F-31077 Toulouse, France.
{mallet,matthieu}@laas.fr.

***Abstract*— This paper presents recent developments of the LAAS architecture for autonomous systems: the G^{en}oM3 component generator for functional components design. The G^{en}oM tool was originally designed for autonomous and terrestrial mobile robots. This tool remains fairly general and is supported by a consistently integrated set of tools and methodologies, in order to properly design, integrate, test and validate a complex autonomous system.**

G^{en}oM3 intends to grant middleware independance to robotic software components so that a clear separation of concerns is achieved between highly reusable algorithmic parts and integration frameworks. Such a decoupling let middlewares be used interchangeably, while fully benefitting from their specific, individual features.

I. Historical perspective

From an historical perspective, G^{en}oM has been developed at LAAS since 1996. 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 middleware independence. G^{en}oM provides the programmer with some built-in system primitives such as inter-process communication, datapool capabilities with external access, a robust state machine for algorithms and automatic client code generation. The G^{en}oM tool is part of the Openrobots architecture [7] as a generator of software modules. Openrobots uses robotpkg as an infrastructure to automatically manage its compilation, installation, updates and dependencies. Robotpkg [9] provides an Open source repository with more than 200 packages from which around 120 are from LAAS.

II. GenoM3 overview

G^{en}oM [3] (Generator of Modules) is a code generation framework that allows the definition and the production of software components that encapsulate algorithms. A component is a standardized software entity that is able to offer services which are provided by a set of algorithms. Components 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. Also one might expect them to exchange data and communicate with other parts of the system.

Version 2 of G^{en}oM generates code using the LAAS pocolibs [8] middleware. This middleware is targeted toward embedded real-time systems and has a very small overhead. In some circumstances, it is interesting to easily target different middlewares for a robotic component, ideally without changing a single line of code in the source code of the component in question. This middleware independency is useful in the following cases:

- simulation where the real-time constraints may be relaxed,
- software verification and validation, where formal tools should be used in place of a regular middleware,
- targeting different robots running different systems, where the choice of the middleware is imposed.

Over the years, a constantly growing list of robotics middlewares have become available [11], [4], [10], [6]. All those middlewares are different, provide their own specificity and generally exhibit unique

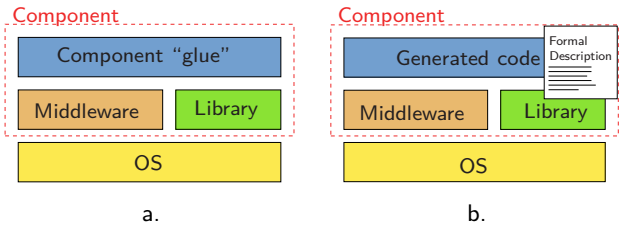

Fig. 1. *a.* A component architecture realizing a clear separation of concerns between a middleware and a library: glue code grants the decoupling. *b.* The GenoM3 tool generates the glue code from a component formal description and a skeleton (not shown on the figure) suited to the middleware.

qualities that make them better suited to a particular task or context. 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.

The version 3 of GenoM [5] generates fully middleware-independent components, due to the component architecture described in Figure 1. The algorithmic core (the set of *codels*[1] implemented by a "library" in Figure 1) 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 their input or output.

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 GenoM description 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.

The core component generation in GenoM3 is implemented with a generic code generation engine. The engine generates code by using the component's formal description file and a skeleton, dedicated to a particular middleware, that is instanstiated according to the description file. The skeleton is not part of the GenoM tool but is provided as an external package so that any kind of skeleton can be developed.

III. Component description language

A GenoM3 component is defined by *i)* its codels, organized in a standalone library and *ii)* a *specification file* that completely defines the component with respect to the GenoM component model. The full grammar is not described here: only important aspects are presented in Figure 2. The following paragraphs cover this example.

[1] *Code el*ements: elementary code fragments managed by GenoM

```
1
2  #include "demo-types.idl"
3
4  component demo {
5     language:    "c";
6  };
7
8  inport long parameter;
9  outport double position;
10
11 ids demo_state {
12    double goal;
13    double velocity;
14 };
15
16 task main {
17    period:            5ms;
18    priority:          20;
19 };
20
21 service setspeed(inout velocity) {
22    validate:
23       demo_chk_velocity(inout velocity);
24 }
25
26 service goto(in goal) {
27    task:         main;
28    throw:        SERVO_ERROR;
29    codel start:
30       demo_init_velocity(out velocity)
31       yield main;
32    codel main:
33       demo_servo(in goal, out velocity,
34       outport position) yield main, stop;
35    codel stop, emergency:
36       demo_stop() yield ether;
37 };
```

Fig. 2. A sample component description file.

a) Data types: A component description always start with the definition of data types used in the interface (line 2). Types are typically defined in separate files and `#include`d in the description, so that the definitions can be shared amongst other components.

The syntax used is the subset of the OMG-IDL language related to data type definitions. Using IDL guarantees the programming language independance and offers a standardized approach.

b) Data ports: Data ports are defined via the `inport` or `outport` keyword, followed by an IDL

type and the name of the port (lines 8-9). Those communication ports correspond to data made publicly available in the system, with the single-writer multiple-readers model. The implementation of data ports is provided by the component template.

c) Execution tasks: Execution contexts are defined by the `task` keyword and a name (lines 16-19). Properties like execution period or priority are optional. A task defines execution timing properties. They may be implemented as separate threads in a typical component template.

d) Services: Services (lines 21-24 and 26-37) may have input arguments or produce output, defined as an element of the IDS (lines 11-14). A component may provide as many services as needed.

An important part of the service definition is the mapping with user codels (lines 29-36). Each codel associated to the service is defined by the `codel` keyword followed by an event name. Some names are reserved: `start` and `stop` correspond to the service invocation or interruption. Other events may be generated by running codels (*e.g.* `main`, line 32).

Codels themselves are described with an IDL-like syntax: the name of the function is followed by its arguments that can be of type `input`, `output` and taken from the IDS or the name of an `inport` or `outport`. The signature of the codel function must of course match this definition. Codels return a value which can be either an error, terminating the service and raising one of the exceptions defined (line 28), or a transition of the service. The list of valid transitions for a codel are indicated via the *yield* keyword.

```
#include <stdio.h>

int main() {
  <'set c [dotgen component]'>
  printf("Services of %s component:\n",
         "<"[$c name]">");
  /* loop on services */
  <'foreach s [$c services] {'>
     printf("- <"[$s name]">\n");
  <'}'>
  return 0;
}
```

Fig. 3. A sample dummy G^{en}oM3 template in C: the special `<"` and `">` markers are interpreted as TCL code and the result of the evaluation replaces the markers. `<'` and `'>` markers also evaluate TCL code, but they do not produce output in the generated file. The TCL code has access to the whole component description file. In this example, it only retrieves the component via the `dotgen component` command, the component name from the component object stored in the `$c` variable and the list of services returned by the `$c services` command.

```
#include <stdio.h>

int main() {

  printf("Services of %s component:\n",
         "demo");
  /* loop on services */
     printf("- setspeed\n");
     printf("- goto\n");
  return 0;
}
```

Fig. 4. The result of the instanciation of the template shown on Figure 3 with the description file shown on Figure 2. This code can now be compiled into a working component.

IV. Component templates

Component templates implement the G^{en}oM component model and instanciate one description file according to this model. They are organized as a set of regular source files, using primitives such as threads or semaphores and the specific middleware API to take care of the communication aspects such as remote procedure call or data marshalling. In short, a component template contains all the software that is not part of the algorithmic core of any specific component. Only one template should be required for a given middleware: the template should be generic so that it can be used for all the individual component of an application.

Templates must be instanciated to a particular description file: this is done by G^{en}oM thanks to code generation. From a component description file, the G^{en}oM parser builds an abstract syntax tree and converts it into a suitable representation. Then, each source file of the component template is read by G^{en}oM and interpreted specially. `<" ">` markers in the file are detected and their content replaced by the result of the evaluation according to the TCL scripting language. To draw a parallel, this is exactly how a PHP script is embedded into an HTML page (see Figure 3). The scripted code (between markers) has access to all the information of the component description file. So a typical template will consist of regular code, mixed with scripted loops on *e.g.* services that generate calls to functions of the core libraries. Since the interpreter relies on a complete scripting language, there is virtually no restriction on what a template can express.

V. Future Work

A number of GenoM3 templates are beeing currently developed.

A. GenoM3 and the pocolibs template

The pocolibs template of GenoM3 implements the same functionaly as what GenoM version 2 used to provide. This let us mix newer components with older, existing ones and ease the transition between the two GenoM versions.

This template, which is almost ready for general use, was used to validate the GenoM3 approach during its development. Being able to implement a set of exiting functional modules which exercised various aspects of GenoM 2.0 and pocolibs functionalities demonstrated the soundness of the approach its ability to further abstract the algorithms from the actual middleware used.

Furthermore, since the pocolibs middleware is truly real-time and can run on small hardware, we will continue using it for embedded computing, such as UAVs (ref manta) or custom devices (ref Bidule).

B. GenoM3 and BIP

One of the benefits of making GenoM middleware independent is the possibility to make the functional layer more robust and verifiable [2]. In particular, a recent effort to integrate GenoM with a component based framework called BIP, used for implementing embedded real-time systems, was carried out.

A BIP model of the generic GenoM+pocolibs component template was developed and "BIP modules" were synthesized for the complete functional layer of a rover. GenoM3 template mechanism will let us go further since a GenoM-BIP component template can directly (and automatically) generate the BIP modules from each component description file. This would result in a rover controller correct by construction and that could be run by the BIP engine. It is also possible to check if the model satisfies particular properties and constraints, by using Verification and Validation (V&V) tools such as D-Finder [1].

C. GenoM3 and ROS

There is a large community effort driven by the Willow Garage company to develop and integrate software for the PR2 robot using the ROS framework. A ROS template for GenoM3 would let us directly mix GenoM3 components with native ROS components, or even better encapsulate algorithms provided as ROS components, in order to comply with a software architecture which can have different constraints compared to the exiting ROS system.

Conclusion

A number of good middleware tools for robotics have been developed recently, but they all lack the ability to propose a standard behavior of the components that are implemented. Our experience with GenoM has shown that this is a key feature for anyone building a complex architecture with several components which may be developed by independant teams. GenoM3, through its template mechanism, provides a way to define a standard behaviour of components and then generate the code implementing this behaviour using the middleware of choice of the user.

References

[1] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and N. Thanh-Hung. Designing autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1):67–77, 2009.

[2] 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 levels. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Sapporo (Japan), September 2010.

[3] 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 International Conference on Intelligent Robots and Systems*, volume 2, pages 842–848, Grenoble (France), September 1997.

[4] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots Journal*, 22:101–132, 2007.

[5] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand. Genom3 : Building middleware-independent robotic components. In *IEEE International Conference on Robotics and Automation*, pages 4627–4632, Anchorage, AK (USA), May 2010.

[6] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *International Cconference on Robotics, Automation and Mechatronics*, pages 736–742, September 2008.

[7] OpenRobots: Software for Autonomous Systems. `http://www.openrobots.org/wiki/`.

[8] Pocolibs: POsix COmmunication LIbrary. `http://www.openrobots.org/wiki/pocolibs`.

[9] robotpkg. `http://homepages.laas.fr/mallet/robotpkg`.

[10] RoSta: Robot Standards and Reference Architectures. `http://www.robot-standards.org/`.

[11] A. Shakhimardanov and E. Prassler. Comparative evaluation of robotic software integration systems: A case study. In *IEEE International Conference on Intelligent Robots and Systems*, pages 3031–3037, San Diego, CA (USA), 2007.