2010 IEEE International Conference on Robotics and Automation
Anchorage Convention District
May 3-8, 2010, Anchorage, Alaska, USA

GenoM3: Building middleware-independent robotic components

Anthony Mallet, Cédric Pasteur**, Matthieu Herrb, Séverin Lemaignan, Félix Ingrand*

Abstract— **The topic of reusable software in robotics is now largely addressed. Components based architectures, where components are independent units that can be reused accross applications, have become more popular. As a consequence, a long list of middlewares and integration tools is available in the community, often in the form of open-source projects. However, these projects are generally self contained with little reuse between them. This paper presents a software engineering approach that 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. This work has been integrated into a new version of the open-source G$^{\text{en}}{}_{\text{o}}$M component generator tool: G$^{\text{en}}{}_{\text{o}}$M3.**

I. Introduction

Autonomous robots have to be endowed with a great amount of different software pieces. By nature, these pieces are heterogeneous: not only different kind of tasks have to be implemented (often categorized as "perception, decision and action") but also different pieces have to fulfil very different timing constraints, ranging from hard real-time requirements to offline data processing or loosely coupled reasoning activities. Taking into account the fact that all the pieces also have to interact with many of the other pieces, autonomous robots end up to be what is commonly described as a "complex system".

In order to master this complexity, component-based software architectures have proven to be a viable solution [13], [4]. Software components ease the system building task by focusing on the software reuse aspect and off-the-shelf software composition rather than programming a complete application from scratch. From the developer point of view, components are usually independent of each other, encapsulate internal details of algorithms and are not application specific.

While they are independant units of processing, software components are primarily meant to communicate with other components. One specificity of robotics (compared to traditional software engineering) resides in that the architecture formed by the set of running components is intrisically dynamic. This characteristic leads to strong requirements in terms of communication, controllabilty and synchronization of components, a task which is mostly performed by the software running *between* the components: the "robotic middleware".

Throughout this paper, "middleware" should be understood as the software that implements the communication and synchronization primitives and grants to components the access to the underlying operating system primitives and device drivers. Middlewares have a major influence on components design and are as such an important part of component-based architectures. The community now benefits from many developments in this area [22], [8], [21], [12]. All existing middlewares are different, provide their own specificity and generally exhibit unique qualities that make them better suited to a particular task or context.

Such a long list of available tools (be they explicitely designed for robotics or not) is a positive fact: it gives more freedom and leaves the choice of picking the tool that is best suited to a particular application. Consequently, projects focusing on providing reusable software that can be integrated in any framework, like GearBox [7] or `robotpkg` [19], are progressively emerging: this raises the issue of selecting the adequate middleware. Since this selection should be done early in the component design process, the choice is critical and can only be revoked at a high cost in terms of software (re)development. This issue is preeminent when different teams have to share software that was not initially developed from the same base choices. Component are, by design, long lived entities and ideal components should thus not be so much tied to any middleware; reusing components in different contexts or catching up with a new middleware contribution should be straightforward. This is however generally not possible as of today. While a lot of effort has been put in making software components modular, reusable and easily replaceable, the same remains to be done at the middleware level.

This paper presents a software engineering approach (and associated tools) that intends to tackle this problem at a meta-level without making any strong assumption on middleware software. The main idea is to decouple the algorithmic core of software components from their middleware encapsulation so that middlewares can be used interchangeably, while fully benefitting from their specific, individual features. This work has been integrated into a new version of the open-source G$^{\text{en}}{}_{\text{o}}$M component generator tool, named G$^{\text{en}}{}_{\text{o}}$M3. After briefly describing alternative approaches, Section II presents the

*The authors are with CNRS ; LAAS ; 7 avenue du colonel Roche, F-31077 Toulouse, France. CNRS ; LAAS is affiliated with the Université de Toulouse ; UPS ; INSA ; INP ; ISAE ; F-31077 Toulouse, France. `{mallet,matthieu,slemaign,felix}@laas.fr`.

**Cédric Pasteur is now with École Polytechnique, 91128 Palaiseau Cedex, France. `cedric.pasteur@polytechnique.edu`.

general approach implemented in G$^{en}{}_{o}$M3. Section III details important aspect of the G$^{en}{}_{o}$M component model and Section IV explains the component description language that grants the middleware independance to the component itself. Finally, Section V illustrates the paper with two concrete use cases that are under development.

II. Middleware-Independant Components

A. Background

One common solution to overcome the software independance issue is to build *abstraction layers* on top of the software expected to be replaceable (see Figure 1). However, building abstration layers is often cumbersome and can lead to performance loss. A more elegant solution is *normalization* (or standardization). For instance, the POSIX norm has, to a large extent, granted the hardware and operating system independance to regular software. However, when it comes to robotics middleware standardization, the state of the art is quite different.

The most mature contributions to the normalization of middlewares and components are OMG CORBA[5] and CORBA Component Model (CCM) [23]. These are not specifically robotics-oriented, though, and miss some important points that have to be addressed in robots architectures. First, the *Request Broking* architecture might not fit well in a dataflow oriented context. Then, CORBA does not provide, *per se*, tools to build dynamic architectures with programable control and data flows. It does not either provide design guarantees that a component fits within a given architecture model (like the layered architecture of T-Rex[10], amongst many others, where decision layer controls functional components). Finally, most of the existing implementations simply forget about real-time computing or are not lightweight as one would expect [9]. A new contribution was brought recently by the OMG RTC (Robotic Technology Component) [14] that provides a robotic-oriented component model compatible with CCM. OpenRTM-aist [15] is the freely available implementation prototype, developed by the AIST and based on CORBA.

Interestingly, many robotic middlewares are based on alternative technologies that are generally not normalized (or are custom extensions to standards). This is for instance the case of major contributions like ROS [20], Player [17] or Orocos [16] whose architecture fall into the case of Figure 1a. This suggests that current standards are not satisfactory to the robotics community. Indeed, normalization is usually achieved only for mastered and well understood issues — at the cost of taking the least common denominator of technologies encompassed by the norm — and robotic middleware might not be mature enough to pass this step. The cost required for switching from a middleware to another might also be an additional drawback to the emergence of a standard.

An alternative solution is *middleware interoperability.* It does not solve all the issues (a single component is still tied to one middleware) but components built

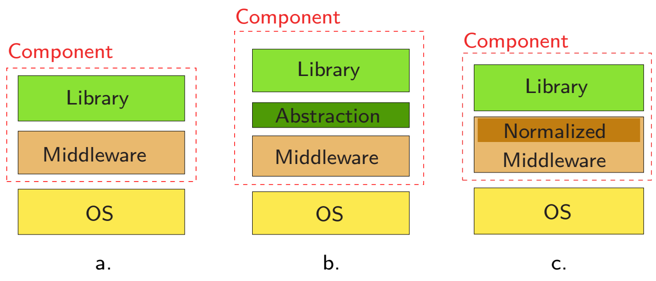

Fig. 1. *a*. Classical component architecture: a library uses a middleware by making direct calls to the middleware functions. The component is tied to the middleware in question. *b*. Abstraction: the middleware can be wrapped with an abstraction layer. Abstraction grants middleware independance at the cost of an extra layer. *c*. Normalization: middleware interface can be normalized. All middlewares have to conform to the norm for granting true middleware independance; which is not easily achieved. Often, the norm will also be the least common denominator of available functionalities.

with different middlewares are able to communicate with each other through interoperable middlewares, solving the issue of using off-the-shelf components for building a custom application. This remains however a cumbersome process, as for abstraction layers, since this involves bridging components, extra conversions or data (re)encoding. While this can be acceptable if only two middlewares are involved, this is hardly tractable in pratice if all the components of an application are built from different middlewares. Only CORBA achieved a real interoperability, thanks to normalization, and this is of course conditioned by individual implementations fully respecting the norm. Beyond CORBA, interoperability is often a matter of transients and *ad hoc* developments.

B. GenoM3: General Approach

In order to overcome the aforementioned issues and grant true middleware independance to robotics components, we propose the component architecture described on Figure 2*a*. The algorithmic core (the “library” on Figure 2) is made independant of middleware by using *glue* software linking the two pieces together. Instead of making direct calls to the middleware, component functions in the library simply have to formally describe the input or output objects they use (Section IV). 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. The immediate benefit of such an architecture is that the problem of middleware independance is deferred to the sole glue software and a clear separation of concerns between the algorithmic core and the middleware is achieved. Additionally, we propose to automatically generate the glue code so that if another middleware is to be used, the latter can be easily replaced (Figure 2*b*).

This approach has been implemented as an evolution of G$^{en}{}_{o}$M [6]. Historically, G$^{en}{}_{o}$M was generating the code of components from a single, generic source code intimately tied with its companion middleware “pocolibs” [18]. The

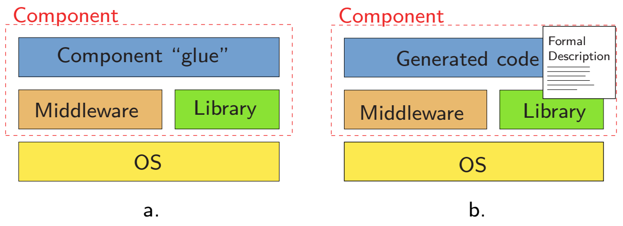

Fig. 2. *a.* A component architecture realizing a clear separation of concerns between a middleware and a library: glue code grants the decoupling. *b.* The G$^{en}{}_{o}$M3 tool generates the glue code from a component formal description and a skeleton (not shown on the figure) suited to the middleware.

major evolution in this version 3 resides in the definition of "Component Templates".

A component template is a generic skeleton, organized as a set of source files. 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 call or data marshalling. In short, a template contains all the source code that is not part of the algorithmic core of any specific component. Thus, only one template is required for a given middleware: it will be *reused* among all components of an application.

Yet, developing different templates offers an opportunity to switch between alternative components architectures (for instance, a threaded versus a non-threaded implementation) or developing *different* component models. Testing such alternative architecture design is a matter of recompiling existing components with a new template. This approach is more versatile than using a standard middleware API since virtually *any* strategy can be implemented in the templates while remaining transparent to the users implementing the core of components. The template-based approach grants middleware independance without the cost of a potentially too specific component model. Additionally, there is no restriction on the language in which component are written, and it is possible to have C, C++, Java or Python templates, provided the core library is written using the same (or a compatible) language.

Since templates are generic, they have to be instanciated for each individual component. This process is done by G$^{en}{}_{o}$M thanks to code generation (see Figure 3). From a *Component Description File* that contains all the necessary information to describe the component (see section IV), the G$^{en}{}_{o}$M parser builds an abstract syntax tree and converts it into a suitable representation for the scripting language of the template interpreter[1]. Then, every file of the component template is read by G$^{en}{}_{o}$M and interpreted. Special markers in the file are detected and their content replaced in a manner similar to how a PHP script is embedded into an HTML page (see Figure 4). The scripted code has access to all the information

[1]Currently TCL and Python are supported, but any language is possible.

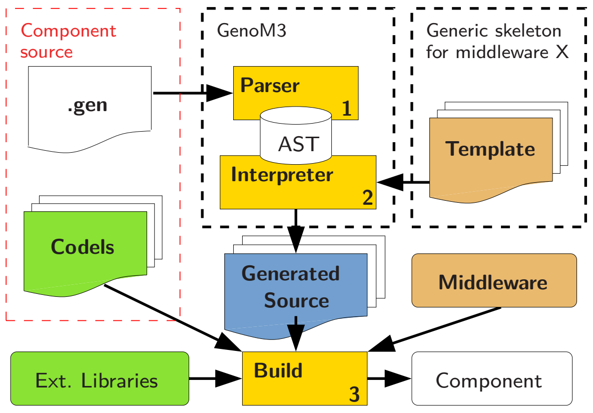

Fig. 3. Overview of the G$^{en}{}_{o}$M3 workflow. The "Component source" represents the algorithmic core of the component and implements the interface (services and data types). The description file (.gen) is parsed by G$^{en}{}_{o}$M and converted into an internal syntax tree. A template, selected amongst available templates, is chosen and instanciated by the template interpreter according to the syntax tree. The generated source code of the component can then be compiled as a regular software in order to produce the component binary.

```
#include <stdio.h>
int main() {
  printf("Running component '%s'!\n",
     "<?tcl $component name ?>");
  return 0;
}
```

Fig. 4. A sample G$^{en}{}_{o}$M3 template in C: the special `<?tcl` and `?>` markers are interpreted as TCL code and the result replaces the markers. The TCL code has access to the component description file (here, it accesses only the component name).

of the component description file. 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.

From the component developer point of view, the major concern brought by this approach is that the algorithmic core of a component has to be written as a set of individual functions, performing only elementary actions. These functions are not responsible for the retrieval and disposal of their inputs and outputs, nor for the implementation of architecture related aspects: this is a template concern. This is however no different from the way any regular library is written and is not a problem in practice. These kind of functions are called *codels* [6] (*code elements*) in G$^{en}{}_{o}$M's vocabulary and the rest of the paper will use this term to refer to such functions. Note that codels are nothing more than *callback* functions, bound to each services and states of a component.

III. Component Model

G$^{en}{}_{o}$M3 components follow a generic model defining the concepts that a component template has to implement. The model does not impose anything on the im-

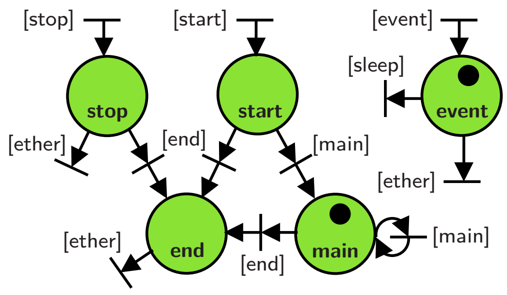

Fig. 5. A sample Petri net of a G$^{en}{}_{o}$M3 activity. Places correspond to codels and transitions are triggered by events. The termination of a codel generates an event activating the next transition. Two active places are shown (`main` and `event`). The Petri net of activites are defined in the component description file.

plementation itself, since implementation is the responsibility of the component templates. After sketching a few general properties, this section details two important aspects of the component model that are services and data ports.

A. General properties

Components are executable programs. They define an interface made of a set of services and data ports. Components must provide a service invocation facility as well as unidirectional (input and output) data ports. They must be able to send and receive events. They have to define an internal data structure, named the *IDS*, that is used by codels to store global and permanent objects. Each codel has access to the IDS objects and use them to communicate with other codels within the component. Finally, components have to define one or more "execution tasks" that represent the execution context (*e.g.* thread) in which services are run. Those tasks may be periodic or aperiodic.

B. Services, activities and Petri nets

A service is an interface for running codels. It can be invoked via a request on the component service port. Services have optional input and output data and a list of failure reports. Input data is stored in the IDS (and output read from there), so that codels can access it. A service might be incompatible with other services of the same component or can be started multiple time, provided it is compatible with itself. It always interrupts other incompatible services when starting.

A service invocation triggers an *activity* that manages codel execution. An activity is described by a Petri net in which places correspond to codels execution and transitions are events generated either externally or implicitely by the return value of codels (see Figure 5). When an activity starts, the `start` event is generated and the corresponding codel executed. Similarly, the activity is interrupted whenever the `stop` event is generated. Asynchronous events trigger the execution of the corresponding codel (if any). A special `sleep` transition is defined so that an activity can be put in a sleeping state, waiting for external events or a `stop` to trigger a new transition. The activity stops when all active places in the Petri net have returned the special `ether` event.

If the execution task of a service is periodic, transitions are executed at each period. They are otherwise executed as soon as all the codels corresponding to active places have returned. The codel execution order is undefined.

It should be noted that no direct remote procedure call (RPC) for service invocation between components is allowed. RPC should be performed by external applications that take care of setting up the architecture of components. While this differs from traditional approaches, this guarantees that components can be controlled and will not interfere with the system. This also grants an increased reusability since no component explicitely depends on a particular set of services implemented by other components.

C. Data ports and events

Components define input and output data ports, used to implement dataflow connections in parallel to services. Event sinks, aimed at receiving external asynchronous events, are supported as well. Components templates should provide the necessary infrastructure to connect ports and event sinks, at run time, to symetric ports. Different categories of data ports can be provided by the templates, such as buffered or unbuffered, ring buffers, etc. This is a run-time parameter and should be selected when the port is connected to another port. Similarly to RPC, the configuration of data port connections shall be done *only* by external applications, so that dynamic architectures can be set up. Data ports are typically of *fixed* size. Variable size data ports are permitted, but templates might not support them. Such ports typically involve dynamic memory allocation and could be problematic with certain middleware or architectures.

IV. Component Description Language

A G$^{en}{}_{o}$M3 component is defined by *i)* its codels, typically organized in a standalone library and *ii)* a *specification file* that completely defines the component with respect to the component model previously presented. The full grammar is not described here: only important aspects are presented in Figure 6. The following paragraphs cover this example and detail some aspects.

Data types: A component description always start with the definition of the data types used in the interface (lines 1-4). Types are typically defined in separate files and `#included` in the description, so that the definitions can be shared amongst other components.

The syntax used is the subset of the IDL language (part of the OMG CORBA [5] specification) related to data type definitions. Using IDL guarantees the programming language independance and offers a standardized approach. Although IDL is part of CORBA, it should be

```
 1  struct demo_state {
 2     double position;
 3     double velocity;
 4  };
 5
 6  inport long parameter;
 7  outport double position;
 8  event emergency;
 9
10  component demo {
11     language:   "c";
12     ids:        demo_state;
13  };
14
15  service goto {
16     task:        main;
17     input:       position;
18     output:      position;
19     throws:      SERVO_ERROR;
20     codel start:
21        demo_init_velocity(out velocity)
22        yield main;
23     codel main:
24        demo_servo(in position, out velocity,
25        outport position) yield main, stop;
26     codel stop, emergency:
27        demo_stop() yield ether;
28  };
29
30  task main {
31     period:           5ms;
32     priority:         20;
33  };
```

Fig. 6. A sample component description file.

clear that components are not tied anyhow to CORBA[2].

Generic information: A component defines the programming language used by the codels (line 11) and the data type of the Internal Data Structure (IDS, line 12).

Data ports and event sinks: Data ports are defined via the `inport` or `outport` keyword, followed by an IDL type and the name of the port (lines 6-7). An event sink in simply defined by its name (line 8).

Execution tasks: Execution contexts are defined by the `task` keyword and a name (lines 30-33). Properties like execution period or priority are optional.

Services: Services (lines 15-28) may have input arguments or produce output, defined as an element of the IDS (lines 17-18).

An important part of the service definition is the mapping with user codels (lines 20-27). 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 are either generated by running codels (*e.g.* `main`, line 23) or via an event (*e.g.* `emergency`, line 26).

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.

[2]Of course, templates are free to use a CORBA-based middleware but this is completely unrelated to the description file.

Codels return a value which can be either an error, terminating the service and raising one of the exception defined (line 19), or a transition in the Petri net of the service. The list of valid transitions for a codel are indicated via the *yield* keyword.

V. Component Templates: use cases

The first G^{en}$_{o}$M3 template developed corresponds to the previous version of G^{en}$_{o}$M. The main purpose of this template is to achieve a soft transition between older and newer components, but it also served as a validation of the new concepts. It has been written using the Pocolibs [18] middleware, and is especially well suited for real-time systems.

For demonstration purposes, two other templates have been developed with two other middlewares: YARP [11] and OpenRTM-aist [15]. Three components dedicated to stereo-vision were ported to G^{en}$_{o}$M3 and have been successfully compiled and run with all these templates.

The two next paragraphs describe two other concrete use cases of components templates that are currently under active development.

A. *The openrobots simulator project*

The OpenRobots Simulator is intended to be a versatile, open-source simulator for mobile robotic applications. The development is driven by several European and French research projects with a wide range of requirements: from large simulation of multi-robot cooperation (up to twelve simultaneous robots) in open outdoor environments to human-robot interaction scenarii in indoor environment. Another strong requirement is to achieve compatibility with the largest set of middlewares.

The simulator uses the Blender Game Engine [3] and relies on a set of reusable components to build the simulation scenarii. Simulation components are independant entities that comprise *i)* a geometrical and physical representation (3D meshes with physical properties); *ii)* a formal description of data they use and produce (a G^{en}$_{o}$M file); and *iii)* Python code (*codels*) that implement the algorithmic core. Since the components are to be run in the Blender Game Engine, the template has to be written in Python (possibly wrapping underlying C or C++ libraries). Different templates can be written for the middlewares that are to be supported, like YARP and Pocolibs. As a result of the code generation, components consist in a Python script that is run in the Blender realtime 3D engine.

Simulation components are expected to produce data (or use inputs) at different level of abstraction; each level of abstraction can be specified as a data port in the components. For instance, a stereo camera component could output two video streams (left and right images) or a single 3D depth map depending if the user wants to simulate the reconstruction process or not.

B. Controller Synthesis and Software Validation

Recent developments, in particular service robots, put robots in close vicinity and in direct physical interaction with humans. In order to address the issue of safety and dependability of robot software in such demanding contexts, it is critical to provide a formal approach which guarantees that safety rules and properties related to robot software components interactions are enforced.

The formal specification of $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M}$ components and the use of a common component model for all components was particularly well fitting the BIP[1] (Behaviour, Interaction, Priorities) methodology. Consequently, an approach that combines BIP component based design with $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M}$ components was recently carried out [2]. A BIP model of the generic $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M}$ component template was developed and "BIP modules" were synthesized for the complete functional layer of a rover. $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M3}$ template mechanism let us go further since a $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M}$-BIP component template can directly (and automatically) generate the BIP modules from each component description file. This results in a rover controller which is correct by construction and that can 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[2].

VI. Conclusions and Future Works

This paper has presented a software engineering approach that aims at fostering software reuse, focusing on the middleware level. The central idea is to use components templates, designed for a specific middleware, but easily replaceable thanks to code generation. Our main objective was to propose a solution that enables efficient integration of robotic software withing different frameworks without restraining the possibilities offered by each of the available middleware. A prototype implementation is available and has successfully proven, with three different middlewares, that the approach is effective in handling real components.

Although the proposed component model was thought as generic as possible, it imposes a restriction on the application architecture since direct remote procedure calls between components are not allowed. While this is generally acceptable, this can become a real problem in some situation. Solutions to work around this restriction are currently under study.

Beyond the software reuse problem, we believe that the $\mathrm{G}^{\mathrm{en}}{}_{\mathrm{o}}\mathrm{M3}$ approach can also help in comparing and evaluating available robotics middlewares. Performance analysis, for instance, is rarely tackled because it would require a real application, running many components compiled with different frameworks. Such an experimental setup simply does not exist as of today, but could be made more easily available if switching from a middleware to another would be only a matter of recompiling an application's software.

The component template approach is also potentially powerful as templates can implement virtually anything. An interesting extension would be to implement component composition, that is, producing a single binary program from a set of individual components. This can be an efficient approach for embedded computing or, for instance, if communication performance is a concern.

References

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Pune (India), September 2006.

[2] 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.

[3] Blender. `http://www.blender.org`.

[4] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback. Towards component-based robotics. In *IEEE International Conference on Intelligent Robots and Systems*, pages 163–168, Tsukuba (Japan), 2005.

[5] OMG CORBA 3.1. `http://www.omg.org/spec/CORBA/3.1`.

[6] 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.

[7] GearBox: Peer-reviewed open-source libraries for robotics. `http://gearbox.sourceforge.net`.

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

[9] A. Makarenko, A. Brooks, and T. Kaupp. On the benefits of making robotic software frameworks thin. In *IEEE International Conference on Intelligent Robots and Systems*, San Diego, CA (USA), 2007.

[10] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen. A deliberative architecture for auv control. In *IEEE International Conference on Robotics and Automation*, Pasadena, CA (USA), May 2008.

[11] G. Metta, P. Fitzpatrick, and L. Natale. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems*, 3(1):43–48, 2006.

[12] 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.

[13] I.A.D. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras, and D. Mutz. Toward Developing Reusable Software Components for Robotic Applications. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2375–2383, Maui, HI (USA), October 2001.

[14] OMG RTC Robotic Technology Component 1.0 specification. `http://www.omg.org/spec/RTC/1.0`.

[15] OpenRTM. `http://www.is.aist.go.jp/rt/OpenRTM-aist`.

[16] The Orocos Project. `http://www.orocos.org`.

[17] The Player Project. `http://playerstage.sourceforge.net`.

[18] Pocolibs: POsix COmmunication LIbrary. `https://softs.laas.fr/openrobots/wiki/pocolibs`.

[19] robotpkg. `http://www.laas.fr/~mallet/robotpkg`.

[20] ROS: Robot Operating System. `http://www.ros.org`.

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

[22] 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.

[23] N. Wang, D. Schmidt, and C. O'Ryan. *Component-based software engineering: putting the pieces together*, chapter Overview of the CORBA component model, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., 2001.