A specification of generic robotics software components: future evolutions of G\textsuperscript{en}M in the Orocos context

Anthony Mallet, Sara Fleury

LAAS - CNRS
7, avenue du Colonel Roche
F-31077 Toulouse Cedex 4 - France
mallet@laas.fr, sara@laas.fr

Herman Bruyninckx

KU-Leuven
Celestijnenlaan 300B
B-3001 Leuven (Heverlee) - Belgium
Herman.Bruyninckx@mech.kuleuven.ac.be

Abstract

Robotics systems and software are becoming more and more complex: the need of standard specifications is certainly a key issue in the near future. Indeed, to be able to share, distribute or reuse robotics software components among laboratories or companies we have to define, from the software point of view, a generic robot. By generic, we mean that components should have a standard behavior, be interoperable and, most important, make the assumption that they will controlled and parametrized by external entities, be they human or software.

The Orocos project \cite{orocos} was started to address this problem. It aims at developing a set of robotics software in particular domains and, as a first step, will define a software framework to do that. This paper presents the future evolutions of G\textsuperscript{en}M \cite{genom} (component generator) which we propose as a programming framework in the context of the project. G\textsuperscript{en}M proposes the definition of generic components that are used to implement robotics functionalities (vision, control, motion planning, ...), and the paper presents the definition of those components. They have been designed so that they can be connected together and externally controlled to form a modular functional layer on robots. We have defined three main entities of which components are made: a set of algorithms, an execution engine, and a communication library. The paper explains their role and presents a formal description language that let us achieves a real decoupling between code of the components (the algorithms) and the execution engine.

1 Introduction

For about three decades, particular aspects of robotics systems have been studied: perception, locomotion, trajectory planning, environment modeling ... and, of course, reasoning and all AI aspects. The state of the art contains a really huge amount of contributions in those domains and some real robots are progressively appearing and making their way toward the general public \cite{ABB, iRobot, Saphira}. However, such systems remain mostly of the time laboratory prototypes. If the feasibility of certain tasks is proven, it is still very difficult to share between labs or reuse the software developments that have been done on those robots. One of the reason for that is certainly the lack of standard specification for robotics software developments. Indeed, robots are still very far from the standardization of traditional computer science engineering — workstations and personal computers — that has undoubtedly permitted their development: although the same programs won’t run as-is on different architectures, or file formats change in almost every new application, UNIX or POSIX have done much regarding portability and the API and behavior of a large amount of kernels is well known and well defined.

Hence, an issue for robotic research is not only to develop particular functionalities, but also to conceive and define a generic robot, i.e. a robot that not only has a standard (preferably real-time) kernel, but also a well defined model of components that will implement functionalities (section 2 defines this notion of components). This will not only permit the conception of complex robots, but also to share, reuse and capitalize existing and well known functionalities.

To tackle this problem, some robotics companies have proposed software frameworks. Notable contributions are ABB and Adept, iRobot \cite{iRobot} and their Mobility framework or ActiveMedia \cite{Activa} and Saphira. However, these frameworks are not much opened (for instance the code is often proprietary) nor extensible, in the sense that they were made for the few robots sold by those companies. These frameworks can still let developers conceive very complex autonomous machines, but they would usually be fixed and predefined in terms of sensors and missions. Something which does not appear that often within the proposed frameworks is that autonomous machines must embed numerous and heterogeneous algorithms, each of them being dedicated to a particular functional-
ity. Real-time processes that control actuators have to live together with interactive processes that let users asynchronously specify missions. Decisional processes have to supervise other processes and control all the executions on the machine. The needs in terms of modularity, configurability, communication and control are thus critical.

In the context of the Orocos project [4, 9, 10], we are defining a software framework that will address the aforementioned problems with a generic approach. Reusing the concept of a functional layer [9], which includes all the robot capabilities, we redefine the notion of components of G2RM [9, 10] (previously called modules, see next section). The paper focuses only on important aspect of the specification of those components. More information on the general philosophy or technical aspects can be found in [9]. Finally, section 3 presents very briefly some parts of the components description language which is the key for portability and reusability.

2 Components definition

Components are software building blocks, each with a specific functionality, that the user can assemble to achieve the global functionality of the robot; they have a standardized way to exchange informations between each other so that they can be connected together. Components are executables: they represent an encapsulation of the notion of processes and threads. They will typically be dedicated to a particular sensor or actuator (and handle the data of that sensor or control that actuator) but this is not mandatory: they can also provide abstract and high-level services and rely on data produced by other components.

Components execute their functionality (or “deliver their services”) independently of the context in which they are used, i.e. without having to know during design and implementation who is going to use its services, and how they are going to be used (e.g. unknown data sources). The independence of a component is a guide to decide what functionality should be put in it (the “grain size”), but it is not absolute: this choice is left to the developer.

In order to accomplish tasks, the services provided by components are activated by external entities according to the task requirements and the execution context (e.g. device availability, contingent events, ...). These sequences of services can be directly activated by a human operator, planned and triggered by decisional components, or more generally by events generated elsewhere in the system. It is important to stress that this structuring does not presuppose any decisional architecture: the way components are connected together is not forced nor specified. Hence, these controllable components can be used for implementing any decisional architecture, from behavior based to supervised and planned approaches. The

\[\text{Codels, execution engine and communication library (depicted by arrows). Codels and the description file are independent of the rest of the system and make the component portable.}\]

same set of components can serve different purposes, when re-used in different architectures.

In order to make the components implementation independent, and hence portable either from one robot to another or between different decisional architectures, we have defined three main entities that compose them (figure 1): a set of algorithms, an execution engine, and a communication library. Algorithms are structured into individual steps called codels (which stands for code element): they are described in the next section. The execution engine sequences the execution of codels, manages the internal state of the component and handles data and control flow (Section 2.2). Last, the communication library provides the means to exchange messages between components (Section 2.3). We also identified three class of users who participate in the building of a component (figure 1): the Developer, who proposes the present framework and maintain it (that’s a computer scientist), the Builder, who develops a particular component (he has the technical knowl- edge to do that, and is not necesserally a computer scientist, although some basic programming skills are required), and finally the User, who set up the functional layer of a robot by connecting the components needed to achieve particular missions.

2.1 Codels

A codel is a piece of code and variables local to it, that corresponds to an uninterruptable step (e.g. start, end, main loop, interrupt, recovery, ...) of an algorithm (servo-loop, monitoring, computing, ...) and which is designated for execution by an execution engine (Section 2.2). The return code (state)
of the codel enables the execution engine to make decisions as to the designation of the next codel for execution. A codel’s execution must never be interrupted by the execution engine and is, in this sense, atomic.

Codels belong to the completely portable part of the framework: they are typically grouped into a codels library, which represent the core of a component. For instance, only this library (along with a description file, Section 3) has to be shared between parties who would like to exchange components. The other parts of the components (execution engine and a communication library) are standard and independent of the codels. To be completely independent of the robot, codels are linked to the sensors and effectors through an hardware abstraction layer, which is out of the scope of this paper and not described here (a working document is available on the Orocos web site [9]).

What code must codels contain. To achieve the maximum portability, we want to insist on the fact that codels have to be totally independent of the communication libraries. Codels must thus be written as simple functions that take some input and return some output: data handling and communication between components is done by the execution engine. This is a very important property that will allow the definition of several strategies for the data flow definition (between components) and achieve a true decoupling between components and the way they are used by the system. Note that this is much stronger than simply relying on standard libraries and defining a standard API for communication.

The interface specification presented in Section 4 exhibit those ideas: all the data the codel wants to manipulate must be declared in a description file. Thus, a function prototype for each codel is defined, and all data can be directly passed to the function by the execution engine. The way these data are retrieved does not influence neither the description of the component, nor the code of the codel itself.

Interruption strategies. As already stated, codels cannot be interrupted in order to guarantee the coherency of components. However, in some situations, it is necessary to stop the execution of a service (e.g. when another service that is incompatible must be started or when a decisional level wants to stop it). Thus, every service is interruptible, and this interruption takes place upon a transition between two codels. If a service is stopped, a special codel will be invoked (the inter codel, Section 3) to cleanly stop the service.

---

2.2 Execution Engine

The execution engine is the infrastructure software in a component that takes care of the properly sequenced execution of services and codels associated to it. It paces the algorithms (codels) using its own threads or processes and can for instance implement periodic executions (servo-controls, monitoring, ...). The execution engine also provides an internal data structure to represent the global state of a component (e.g. current value of services parameters, state of running services, ...).

The sequencing is implemented as an automaton (Finite State Automaton; Petri Net; Hierarchical State Machine; StateCharts; ...) that executes the codels that belong to services. Each state has a codel that contains the functional part of the service in that state; the return value of codels is used to influence the transition to another state. This mechanism naturally endows the component with a great adaptability to the current execution context (the sequencing of the codels within a particular service is not hard-coded).

Execution engine is a generic software library, provided by our framework, that is linked to the codels library to form the final executable component. The engine is designed to be reused for different components and, indeed, we only have one as of now. It was designed to have the best time properties and guarantee bounded execution time. However, it would be fully possible to develop several execution engines, best suited for such or such required properties, or dedicated to particular OS (rt-linux, VxWorks, ...).

2.3 Communication

The specification of components allows the communication to be completely decoupled of the components code (i.e. the codels). As a consequence, both the control flow (to start, stop or parameterize services) and the data flow (to read or produce shared data) has to be established outside the components themselves. This is a major advantage since it allows the definition of any communication strategy: different patterns can be used with the same set of components. Of course, communication libraries and execution engine will be tightly coupled. But this has no impact on modularity, since both are standard libraries, provided by the framework.

At least two basic aspects should be present in the proposed communication libraries, in order to be successfully used by execution engines: there must be one strategy for serving and requesting services, and another for implementing data exchanges between components.

Requests and control flow. Services will be started (or stopped) upon reception of a request sent to the component. Communication libraries should thus provide such a message-passing mechanism, to
be used by the execution engine. One property which seems important is that both the sending of requests and the reception of replies must be done in non-blocking mode. This mechanism allows asynchronous processing of services.

Since services are activities that can last for a long time, the communication library should also be able to generate more than one answer per message. The first one, called acknowledgment, would be sent to the client as soon as the request has been successfully received. It indicates that the request has been accepted, and that it will be honored. Successive answers can then be generated, the last of them being sent when the execution of the request is terminated. This scheme allows the clients of a component to have a control on the services they start, and authorize the construction of a global state of running activities.

Shared data and data flow. Communication between components also requires data transfers during services execution. This is decoupled from the aforementioned control flow and requires a specific mechanism. Currently, a library based on shared memory and named data structured is implemented (posterLib in 
[2]).

3 Component description language

The purpose of the component description language is to propose a formal description of the services offered by components. It will be used as a mean to link an execution engine with the codels, and produce the communication interface (the structure of the messages for service invocation), by parsing the file at compilation time.

The description doesn’t presuppose a particular implementation: any execution engine or communication library (although we have only one as of now) can be used to execute the codels. Hence, distributing a component consists in distributing the codels library plus the description file, and only that. Porting components to different architectures requires at most recompilation of the codels for another execution engine (if a different one is wanted).

The description language contains structures that encapsulate attribute / value pairs. Figure 2 presents an uncomplete example that defines a component that controls a simple mobile. Only important attributes and values pairs are detailed in the subsections below; aspects such as a formal grammar are not relevant in this paper and will not be discussed.

3.1 General information

The component structure (top of figure 2) gathers general attributes that globally describe the compo-

2 Only one in the current version of GenoM.

3 This language is not a new programming language. Its purpose is to provide a description only. We proposed the notion of codels (Section 2.1) to handle the programming aspects.

```c
/* Current state of a mobile device */
typedef struct DEMO_STATE_STR {
    double position; /* current position */
    double speed;   /* current speed */
} DEMO_STATE_STR;

component pilot {
    category: default;
    lang: C;
}

include "demoStruct.idl" /* structure above */
ids demoIds {
    double distRef; /* Distance reference */
}

export demoExport {
    DEMO_STATE_STR state {
        position:= 0.0 "Current position";
        speed:= 0.0 "Current speed";
    }
}

thread motionTask {
    priority: 0;        /* normal priority */
    period: 1s;         /* 1 second */
}

service moveDistance {
    doc: "Translates of a given distance";
    thread: motionTask;
    report: TOO_FAR_AWAY;
    stops: moveDistance;
    input: demoIds.distRef:=0.0 "Distance to move";
}

control codel {
    /* parameters control */
    exec: demoControlDistance(demoIds.distRef);
}

start codel {
    /* start the mobile */
    exec: demoStartMobile();
    on-inter: stop;
    next: goto, stop;
}

exec codel goto {
    /* main servo-loop */
    exec: demoGotoPosition(demoIds.distRef, demoExport.state);
    update: demoIds.distRef, demoExport.state;
    on-inter: stop;
    next: goto, stop;
}

inter codel stop {
    /* stop or interrupt */
    exec: demoStopMobile(demoExport.state);
    update: demoExport.state;
    next: ;
}
```

Figure 2: Example of a component description file. The box on top is an IDL structure definition, which would be defined in a header.idl file.
nent. It is not fully specified yet and can be extended. However, the fields included in the example (figure 4) illustrate the kind of information that can be defined: a name (pilot in the example) that identifies this component and a category (default in the example) that will select an execution engine among those that are available. For instance, the category real-time could be defined so that the components of this category use a real-time execution engine (e.g. constant-time, no dynamic memory allocation, ...).

3.2 Data structures

This part defines the data structures that are used by the codels and will allow the generation of the message structures of the request\(^5\). Header inclusion is allowed within the description file (#include statement in the example). Such headers are written in IDL language. This allows to share them with the code of the codels, or with other libraries (and in particular with clients of the component), whatever be the programming language used.

Each component has an internal data structure (IDS, see 4), which is shared among codels and represents the instantaneous state of the component. It will be composed of all the controllable parameters that are visible to and can be changed by clients. There can be several such data structures, when concurrent accesses are needed (these structures will be typically locked when accessed).

3.3 Code execution

This part allows the declaration of the threads (provided by the execution engine) that will execute the services. A single thread can parallelize the execution of several services only if they share the same time and priority properties: priority and period. The period is the rate of the underlying sequencing machine: an a-periodical thread will execute code as fast a possible and a periodical thread will sequence the code execution at the specified rate. Two hooks (functions written in the language of the component) can be defined and would be invoked upon thread creation (start) and deletion (stop).

\footnote{We still do not have defined a strategy to share these definitions (this should appear later in the Orocos lifetime).}

```
thread <name> {
  priority: <number>;
  period: <seconds>;
  stack: <size>;
  start: <function>;
  stop: <function>;
}
```

3.4 Services

Services are the most complex part: we must describe their input, their output, the shared data they work on and produce, and the codels they are made of, so that codels contain no code regarding communication. Services description begins with the following structure:

```
[init|auto] service <name> {
  doc: "short description";
  
}
```

The init or auto attribute can be used to distinguish between three types of services. Basically, init indicates a service that has to be run before any other service to perform some initializations. auto indicates an permanently running service. Any other service will be triggered on demand (by the reception of a request).

```
thread: <name>;
report: <name> [, <name>, ...];
max-time: <seconds>;
credential: admin | ... | none;
```

This important declaration defines compatibilities (or incompatibilities) between services and allows the component to handle conflicts. stops indicates that if this service is requested, it should interrupt any instance of the listed services. delays indicates that if this service is running, any service listed should be delayed. forbids would prevent any listed service to execute when this service is running.

```
/* input/output parameters */
input|output: <type> <variable>|<idsref> [:=<default-val>] ["short description"], ...
```

This part defines the input and output parameters of the service. They are typically small data, which are required to parameterize the service. Such parameters can be defined with just a type and a name (such as double speed), or can be a reference into a particular IDS (such as <ids name>..<member>).

\footnote{In such a case, the input would be automatically stored}
The short description could be used when invoking the service interactively and prompting the user for the required input.

```c
/* imported/exported data */
import|export: <type> <variable>;
```

import and export tokens define shared data (their type and a name) imported or exported by the service. The difference with parameters is that shared data are permanent, public, and do not belong to the component once they have been produced. They will typically be the result of most services (maps, positions, trajectories, ...).

```c
[start|inter|control|exec] codel <name> {
  exec: <function>([const] <variable>, ... );
  update: <ids-ref><import|<export>;
  on-inter: <codel>;
  max-time: <seconds>;
  next: <codel> [1, <codel>, ...];
}
/* end of service declaration */
```

This part defines the core of the service: the codels (Section 2). A service can be made of as many codels as necessary and there are four types of codels:

- **control**: this is a particular codel, executed by the control thread. It is used to check the input parameters before the service starts (and also before the data is put in the databases, if so requested).
- **start**: this is the first codel to be executed when the service starts. The sequencing of the other codels will be done by the codels themselves (by returning a particular value).
- **inter**: this defines a codel that can be executed when the service is interrupted. There can be as many inter codel as needed, and each codel tells with the on-inter token which interrupt routine it needs.
- **exec**: any other codels are execution codels, and implement the core of the FSM of the service. The next attribute defines the authorized transitions between codels.

The exec links the codel to a function or method. It looks much like a C function call, where the input and output parameters that the codel uses are listed, and must match the prototype of the underlying function. The update keyword indicates that the data (internal, imported or exported) used by the codels has to be refreshed when the codel executes. This allows the execution engine to take the appropriate actions.

4 Conclusion

We have presented an overview of the specification of generic robotics components, and a component description language that allows the connection of specific code (the codels) with a generic execution engine and communication libraries.

We paid much attention on obtaining a high decoupling between the component specification and the usage of it: communication, cooperation with other components and underlying system (robot, OS) have been abstracted thanks to the description language we propose and the notion of codels.

One important aspect that has not been addressed in this paper concerns the shared data structure definitions. Indeed, shared data has to be defined outside components and the specification of an “object factory” is needed. For this aspect, CORBA and IDL might be an appropriate candidate and will be considered.

This framework is still under development and has not been experimented yet (ideas and discussions in the Orocos mailing lists are still welcomed!). However, the current framework developed at LAAS (G*M) is intensively used and has greatly demonstrated the interest of generic components, standard interfaces and automated integration.

References