Tentative specification of components' interface

A working document for the Orocos project

Anthony Mallet, Sara Fleury

1 Introduction

The ideas presented in this document are part of a more general reflection on software architecture for robots that LAAS is currently carrying out. As of today, the "LAAS" architecture comprises into three main layers (see [1] and figure 1):

- **a functional layer.** It includes all the basic built-in robot action and perception capacities. These processing functions and control loops (image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating components. In order to make this level as implementation independent as possible, and hence portable from a robot to another, its is linked to the sensors and effectors through a *logical robot* interface.

  In order to accomplish tasks, the components are activated by the next layer.

- **an execution control layer,** or *Executive*. It controls and coordinates the execution of the functions distributed in the components of the functional layer, according to the task requirements. The tasks are sequences of services, planned and triggered by the next layer.

- **a decision layer.** This layer includes the capacities of producing the task plan and supervising its execution, while being at the same time reactive to events from the previous layer. It may be decomposed into two or more levels, based on the same conceptual design, but using different representation abstractions or different algorithmic tools, and having different temporal properties. This choice is mainly application dependent.

This draft document focuses on the functional layer, since Orocos focuses only on it as of today. Ideas presented here are strongly based on the experience gathered with G$^{en}_{o}$M [2, 3], although significant improvements have been made to the original G$^{en}_{o}$M specifications.

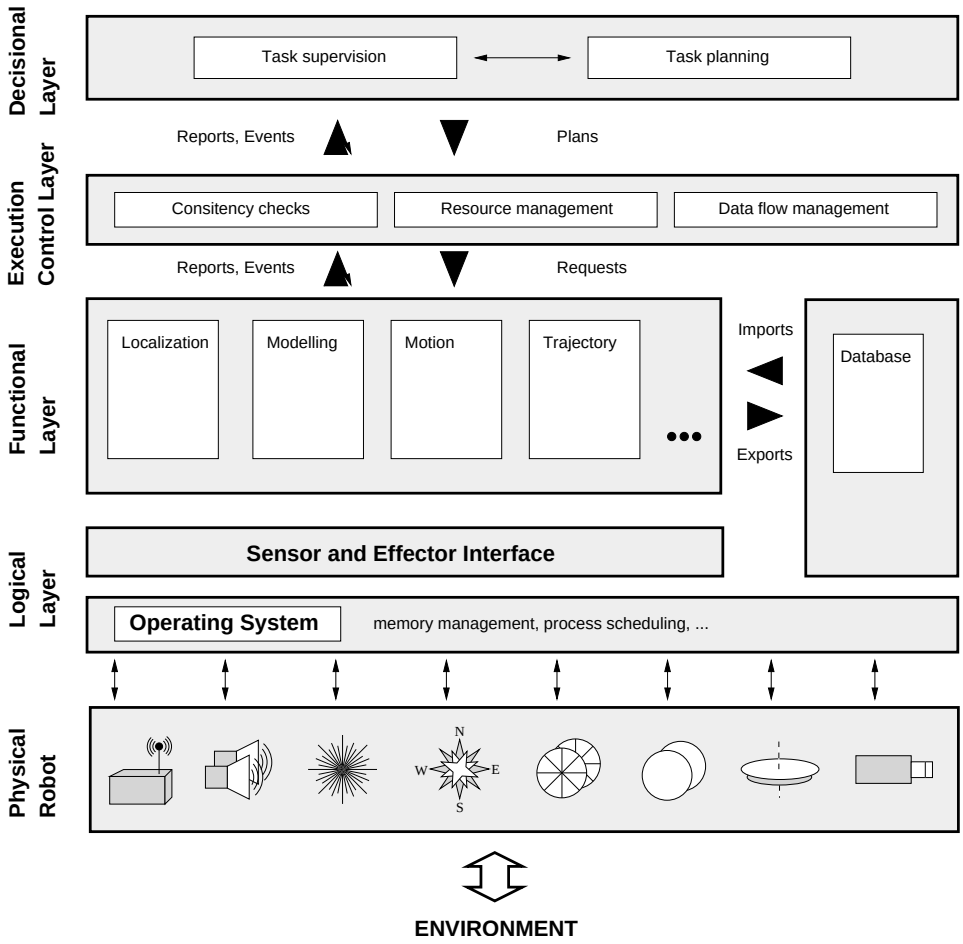

Figure 1: Generic software architecture.

The next section describes some properties of the functional layer. These considerations have lead to the specification of a generic interface for components, presented in the section 4. This specification is intended to be completely implementation independent. However, components must have several properties: section 3 presents the notion of *codel* (code element), which is a way to *structure* algorithms, and section 5 presents some communication schemes between components.

2 Functional layer

The functional layer is made up of a set of *components*. Components are *active* entities that provide both algorithms and also the means to execute them. Components are typically asynchronous.

Each component will offer a specific set of services and thus provide specific functionalities to the robot system. Components are typically dedicated to a particular sensor or actuator (and handle the data of that sensor or control that actuator) but this is not mandatory. Components can also provide abstract and high level services, by relying on data produced by other components.

An important point is that the functional layer must be *modular*. By *modular*, we mean that the global set of functionalities must not be known *a priori*. Thus, the components cannot know which other components it will interact with, and should be written with this in mind. Such an approach provides a great level of re-usability, and also let the design of individual components be done in a distributed way.

A consequence is that both the *control flow* and the *data flow* must be defined outside the components. For instance, in the LAAS architecture, this is done at the *Executive* level. The code contained in the components thus becomes naturally independent of the particular robot on which it will execute.

3 Codels

Codels are pieces of code written by the developer of the component. The purpose of codels is to split the execution of each service into *atomic* entities. The main advantage is that this provides *reactivity* with respect to external events to the components. It also allows the implementation of a finite state machine as a model of execution of the services: the components are always in a known (and sane) state when they receive requests or execute several services in the same thread.

The main idea to keep in mind when writing codels is that they cannot be interrupted. When their execution has begun, it will continue until the

codel ends: the component can only perform actions before and after the execution of this code. Of course, this does not mean that codels are mutually exclusive: the underlying operating system can still perform task scheduling between threads of a component or between components themselves.

3.1 What code should codels contain

One idea that we want to promote is that codels should be *totally independent* of the communication libraries. This is a very important property that will allow the definition of *several* strategies for the data flow (between components) definition and achieve a very *decoupling* between components definition and the way they are used by the system. The components will thus be highly reusable and portable across several robots, machines and missions. This also let several communication schemes to be used, with no impact on the components themselves. Note that this is *much* stronger than simply relying on *standard libraries* (and defining standard functions for communication).

Actually, codels should contain *only* the code that performs some action on the data, or controls the hardware (through the sensors and effectors libraries as shown in figure 1). Even more: codels should only interface *libraries* with the component's structure. These libraries could then be used in other context.

The interface specification presented in the next section tries to exhibit those ideas: all the data the codel wants to manipulate must be declared in the description file. Thus, a *prototype* is naturally defined, and all data can be *directly* passed to the function. The way these data are retrieved does not influence neither the description of the module, nor the code of the codel itself. One can then imagine several strategies to get the data: code generation (as for $G^{en}_{o}M$), standard templates, manual writing, ...

3.2 Interruption strategies

As already stated, codels cannot be interrupted; the components only act upon transition between codels. But, in some situations, it is necessary to stop the execution of a service. There are at least three such situations: *i)* when another service that is incompatible must be started, *ii)* when a service (or a codel of it) has already lasted more than the allowed upper time bound, or *iii)* when the decision level wants to stop a service. Thus, every service must be made interruptible.

If a codel is stopped because of an incompatibility or because the decision level decided it, a special codel will be invoked (the `inter` codel, see next section): it should perform all the necessary actions to cleanly stop the service, and prepare the component to handle new requests. Then, new services will be able to start.

If a codel is stopped because it has executed more than the allowed time, it is much more difficult to react: the faulting codel must return by itself so that the component can invoke the `inter` codel. A strategy that could be used in this situation would be to send the final reply to the caller of the service (with a report such as `timeout`), and to invoke the 'inter' codel as soon as the codel ends (by itself).

4 Components' interface

The purpose of such an interface is to propose a *formal* description of the services offered by components, and the way they can be invoked. This description *must not* presuppose any particular implementation.

As for GenoM, we propose to describe this interface by using a particular language. Indeed, existing languages do not provide a satisfactory way to express all the information needed. We payed much attention to the fact that this new language must *not* be a new *programming* language. Its purpose is to provide a *description* only. We proposed the notion of *codels* to handle the programming aspects (see section 3).

The description language contains either *structures*, or *attribute / value* pairs. Structures are used to encapsulate attributes for a particular description. They are all contained in the same file: figure 2 (page 6) summarizes the interface description which is detailed below. We do not provide a formal grammar yet, mostly for clarity reasons.

4.1 General information

This structure gathers high level attributes that describe the overall component. It is not very well defined yet and could be extended very much. However, the following fields could be included:

```
component <name> {
   category: none|real-time|...;
   lang:     C | C++ | Java | TCL | ...;
   ...
}
```

Note that we choose the term `component` here, according to the "glossary" of Orocos. GenoM (and the papers referenced at the end of the document) uses the term "module" instead.

The `<name>` is a string that identifies this component. Note that this is *not necessarily* the name of a particular instance: this should rather be understood as a C++ class name, since several instances of the same module can co-exist in the same system. Actual naming of particular instances

```
component <name> {
   category: none|real-time|...;
   lang:     C | C++ | Java | TCL | ...;
}

#include "header.h"

database <name> {
   <type> <variable> [:=<default-value>];
   ...
}

thread <name> {
   priority: <number>;
   period:   <seconds>;
   stack:    <size>;

   start:    <function>;
   stop:     <function>;
}

service <name> {
   doc: "short description of the service";
   type: request|init|auto;

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

   stops|delays|denies: <service> [, <service>];

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

   /* imported/exported data */
   import|export:
      <type> <variable>, ...;

   [ start|inter|control ] codel <name> {
      exec: <function>( [const] <variable>, ...);
      update: <dbref>|<import>|<export>;
      on-inter: <codel>;
      max-time: <seconds>;
      next: <codel> [, <codel>, ...];
   }
}
```

Figure 2: Summary of the component's interface description

should be done at run-time, and defined (for instance) in the Executive layer.

The `category` field could be used to distinguish between several execution models, though we do not identify them in this document. For instance, the category `real-time` could be defined so that the components of the category run in constant-time, or perform no dynamic memory allocation. Of course, such categories are highly dependent on the content of the codels (see section 3).

The `lang` attribute can be defined to tell in which language the codels are written.

4.2 Data structures

This part can be used to define which data structures are to be used within that component. We still do not have defined a strategy to *share* these definitions (this should appear later in the Orocos lifetime).

We think a good way to define data structures is to allow header inclusion within the description file:

```
#include "header.h"
```

Such headers should be written in the language of the component (`lang` attribute in the paragraph 4.1). This allows to share these headers with the code of the codels, or with other libraries (in particular with the *clients* of the component).

Headers should contain only the necessary structures (*i.e.* those that are used in the codel definition or in the database below), to avoid extra dependencies.

There is also the need of a *database* (internal to the component), which represent 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 databases, when concurrent accesses are needed (databases will be typically *locked* when accessed).

```
database <name> {
   <type> <variable> [:=<default-value>];
   ...
}
```

This is mostly like a `C`-structure declaration, where `<name>` is the name of this database, `<type>` a type which must have been defined before (thanks to the `#include` statement above), and `<variable>` the name of the member.

We also added the possibility of defining a *default value* for each member of the database.

The names can be used to refer to particular members later in the description file. Note that this definition must have no implication on the actual implementation of the database: in particular, codels should not rely on the existence of a structure that will contain all the data mentioned. See subsection 4.4 for information on how to use this database.

4.3 Code execution

This part will allow the definition of several *threads* that will actually execute the services. Note that the word `thread` does not necessarily suppose a *multi-threaded* implementation (although this would be a good choice). If a multi-threaded implementation is chosen, threads could be created either statically or on the fly. The only property that should be required is that several `thread`s should be able to parallelize the execution of codels.

```
thread <name> {
   priority: <number>;
   period:   <seconds>;
   stack:    <size>;

   start:    <function>;
   stop:     <function>;
}
```

`<name>` will be used to refer to this thread later. A thread has two important values: `priority` and `period`. Since numerical values of priorities are quite system dependent, we should agree on a definition of these priorities (*e.g.* from -128 to 127). 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 codel execution at the specified rate.

We also let the possibility to define a stack size (`stack`), for systems where it is available and two hooks (function written in the language of the component) that are to be invoked upon thread creation (`start`) and deletion (`stop`).

Note there is always a default thread (which needs not to be explicitly defined) which is the *control* thread: it manages the overall behavior of the module (requests, replies, ...) and also particular codels (namely, *control codels*, see below).

4.4 Services

Services are the most complex part: the goal is to describe their input, their output, the shared data they work on or produce, and the codels they are made of.

Services description begins with the keyword `service`, followed by a name:

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

`init` or `auto` attribute can be used to distinguish between three types of services (init, auto and the default):

`default` : the standard service, which is invoked by the *Executive* layer (or an external entity) through a *request*.

`init` : a particular type of service, mostly the same as `request`, but which must be invoked before any other service. It will be typically used to define some characteristics of the robot on which the component will be executed (geometrical model, application, ...). One could allow the definition of several `init` services, though only one should probably be executed.

`auto` : this is a service which executes as soon as the module is started (or as soon as an `init` service has been executed — if the module defines one). `auto` services should typically never end, and perform a background task (servoing, monitoring, ...).

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

Four attributes that are self-explanatory. `thread` is the thread which will execute the service. `fail-msg`s are character strings, which define several execution reports (interpreted as failures): those reports are transmitted to the caller of the service, when the execution ends. `max-time` can be used to ensure that this service will not last more than the specified value (a report such as `timeout` could be sent if the service lasts more than the allowed duration). `credential` can be used to restrict the usage of this service to a particular class of user, or to a particular way of use.

```
stops|delays|denies: <service> [, <service>];
```

This part is not well specified yet. It is intended to define compatibilities (or incompatibilities) between services. `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, and one of the listed services is invoked, the latter should be registered but not started until this service has terminated. `denies` would prevent any listed service to execute when this service is running. By default, all services would be compatible with each other.

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

This part defines the input and output *parameters* of the service. Parameters are typically small data, which are required to control the behavior of 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 database (such as `<dbname>.<member>`). In the later case, the input received upon invocation would be automatically stored in the database, whereas the output would be taken from the database. The `short description` could be used when invoking the service interactively and prompting the user for the required input.

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

`import` and `export` can be use to define *shared data*, imported or exported by the service. This data goes into a *centralized* database (see figure 1), which is always up-to-date and available. The difference with *parameters* is that shared data are *permanent*, *public*, and do not belong to the component once they have been produced. They might also be continuously updated. They will typically be the result of most services (maps, positions, trajectories, ...).

The `import` and `export` tokens define the `type` of the produced data (and name them). Section 5 shows how these data could be handled by and passed to the components.

```
    [ start|inter|control ] codel <name> {
       exec: <function>( [const] <variable>, ...);
       update: <dbref>|<import>|<export>;
       on-inter: <codel>;
       max-time: <seconds>;
       next: <codel> [, <codel>, ...];
    }
}
```

This part defines the heart of the service: the *codels*. Codels are the smallest code entity that the module can handle (see section 3 and [3]). A service can be made of as many codels as necessary. There are four types of codels:

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, see next section). One can thus define any *finite state machine* (FSM).

inter : this defines a codel that can be executed when the service is interrupted (by an external entity) or stopped. There can be as many `inter` codel as needed, and each codel tells with the `on-inter` token which interrupt routine it needs.

control : this is a particular codel, executed by the *control* thread. It is used to check the input and output parameters *before* the service starts (and also before the data is put in the databases, if so requested).

default : any other codels are *execution* codels, and implement the FSM of the service. The `next` attribute defines the *authorized* transitions between codels.

The figure 3 shows a sample FSM and illustrate the use of the four types of codels.

The `exec` attributes performs the linking between the *code* of the codel and its name. It looks much like a `C` function call, where the input and output parameters that the codels uses are listed, and must match the prototype of the underlying function.

The `update` keyword is used to tell whether the data used by the codels must be refreshed (or not) when the codel executes. For `dbref` (database reference) and `import` variables, this indicates that the data is to be *read* before the codels executes (if not specified, they keep the values they had during the previous codel execution). For `dbref` and `export`, this indicates that the data must be written (in the database, or in the shared data area) *after* the codel has returned. The section 3 explains this mechanism further.

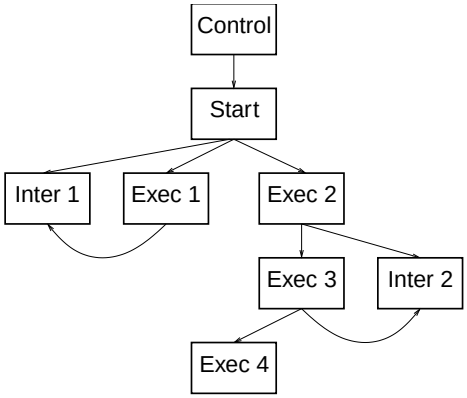

Figure 3: Sample FSM with the four types of codels

5 Communication

We have proposed a component specification that allows the communication to be completely decoupled of the components' code. Potentially any strategy can thus be defined. However, we present here some basic ideas, mostly the one that are implemented in the current version of G^{en}oM. These ideas illustrate the mechanism of the service requesting (paragraph 5.1), or that of the `update` token (paragraph 5.2) or, finally, the mechanism of the `import` and `export` tokens (paragraph 5.3) defined in the specification language.

5.1 Requests

Services will be typically launched by sending a *request* to the component. The request reception is handled by the internal engine of the component, and should be transparent to the component developer. As of today, we use the `csLib` library, which has the following properties:

- Both the sending of requests and the reception of replies can be done in non-blocking mode.
- The reception of a request is associated to the execution of code.
- A request generates at most two replies.

A request can generate two answers. The first one, called *acknowledgment*, is sent to the client *as soon as* the request has been successfully received. It also indicates that the request has been accepted, and that it will be honored. The second answer, called *final reply*, is sent to the client when the execution of the request is terminated.

This mechanism allows asynchronous processing of services.

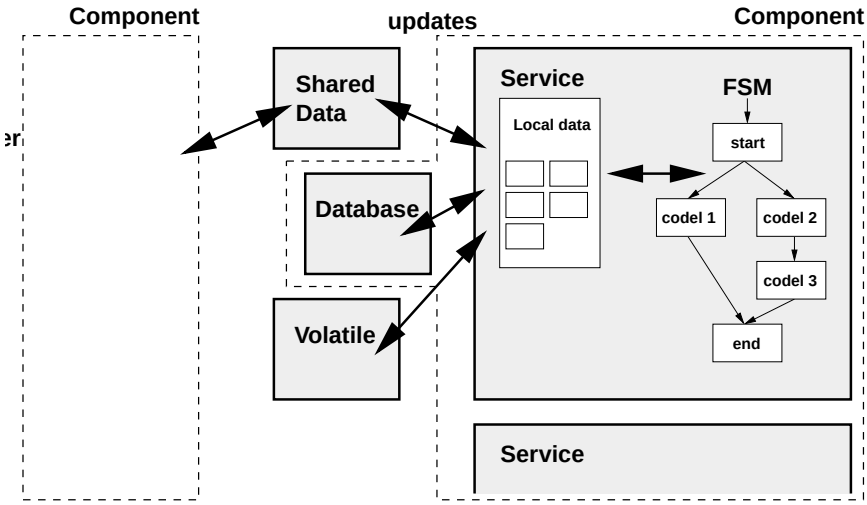

Figure 4: Codel data management

5.2 Updates

Figure 4 depicts the usage of the `update` token in the codel definition, as well as the global data management strategy for services.

Basically, there are three sources of data for the codel functions:

- **shared data** (`import` and `export` keywords),
- **parameters** that go into the **internal** component **databases** (`input` and `output` with a `dbref` variable),
- and **volatile parameters** (`input` and `output` without a `dbref` variable).

When the service starts, provisions should be made (by the component, not by the codel) to locally copy data (within the service address space).

It is possible that, during the execution of the service, shared data or database members are modified by the execution of other services (either from other components, or from the same). Depending on the action of the codels, these data should be read again *or not*. The same also stands for data modified by the service itself: they should be exported systematically, or only after a given action. The purpose of the `update` token is to indicate, for each codel, what strategy should be used.

With the `update` mechanism, the codels has *nothing* to do to publish the results of its action. Thus there is *no code* that will depend on a particular interface in the component itself. Instead, this code is contained in the engine that runs the component, and can thus can be very different from one system to another.

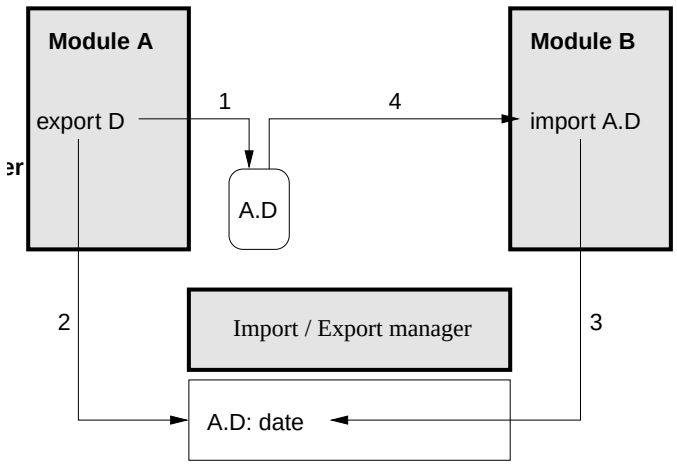

Figure 5: Import/Export strategy

5.3 Imports and exports

Figure 5 shows a basic data negotiation between two modules `A` and `B`. The following typical scenario could be realized:

- First, `A` wants to make a data structure named `D` public. This is expressed thanks to the appropriate `export` statement in the description file, along with an `update` within the appropriate codel. The component then knows that it has to make this data public.
- When the codel that exports `D` has been executed, the public data `A.D` is produced. It is kept locally, within the component address space, and a *shared data management* unit is then notified that the data `A.D` has been updated: the date of such a modification is registered.

  Keeping the data locally, until it is needed, is purely a performance optimization strategy: data is potentially large, and communication between the component and the data management unit potentially slow. With such an optimization, the component is able to produce data at its own pace (overwriting previous data if it has not been read by other parties), with no or very little performance issues.
- Then, `B` wants to import the data previously produced by `A`. This is expressed thanks to an `import` statement, along with an `update` in some codel.

  Note that the `import` statement will only define the *type* of the data. The *name* (in this example, it could be `A.D`) is typically given at runtime, when the service is called. How this name is given is typically application dependent: this could be given either by the user, or by a script running in the *executive* layer, or even by a task planner.

- At the time when the codel that wants `A.D` in `B` executes, the component checks that the *data management* unit has got such a data structure, and checks its date of last modification. It is has been modified since (from `B` point of view — `B` could hold a previously cached copy), it is actually transferred from `A` to `B`.

  Note that the transfer operation should not imply any action in `A`. This should be done by the *data management* unit in order to prevent to affect the execution of `A`.

6 An example

To illustrate how to define a component from the specifications, we present a small example. This is the same example as in the GenoM user's manual [3].

We will create a component which will control a mobile that can translate on a 2 meters long rail, and we name it `demo`. Some of the services the "demo" component offers are:

- select the speed (two symbolic values `DEMO_SLOW` and `DEMO_FAST`),
- move the mobile for a given distance,
- read the current speed or position at any moment,
- suspend the motion,
- monitor particular positions and inform when the mobile goes through these positions.

1.

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

/* Admissible speeds */
typedef enum DEMO_SPEED {
  DEMO_SLOW,                     /* low speed */
  DEMO_FAST                      /* high speed */
} DEMO_SPEED;
```

We first define some `C`-structures that the component will use, and store them in a file which we call `demoStruct.h`

2.

```
component demo {
   category: none;
   lang:     C;
}
```

We can then write the interface of the component, by declaring its name, language and category

3.

```
#include "demoStruct.h"

database db {
   DEMO_STATE_STR state {   /* Current state of the mobile */
      position:=  0.0;
      speed:=     0.0;
   };
   DEMO_SPEED     speedRef:= DEMO_SLOW; /* Speed reference */
   double         distRef;           /* Distance reference */
   double         monitor;           /* Positions monitors */
}
```

We define a database, that we call 'db': it will hold the instantaneous value of all parameterizable values of the component. You can see how default values are provided for structures (by specifying the name of each member we initialize).

4.

```
thread motionTask {
   priority: 0;         /* normal priority */
   period:   1s;        /* 1 second */
   stack:    2k;        /* 2 kilo bytes */

   start:    demoInit;
}
```

We create one thread, that will execute all services. Since the component is simple, it does not need more threads. Numerical values for the `priority`, `period` and `stack` size are only examples. This thread has an initialization routine: a `C` function named `demoInit`.

5.

```
export DEMO_STATE_STR state;
export double         distRef;
```

We declare here all exported data. Remember that exported data are public data, shared among all the functional layer. We declare these structures here, so that someone looking into this file can immediately see what the component exports. Instead, we could have declared those exports in the services that actually perform the export. The names `state` and `distRef` now refer to these exported structures.

6.

```
service setSpeed {
   doc: "Modifies the default speed";

   fail-msg: INVALID_SPEED;

   /* input/output parameters */
   input: db.speedRef:= DEMO_SLOW "Speed of the mobile";

   control codel {
      exec: demoControlSpeed(db.speedRef);
   }
}
```

Now we create a first service, named `setSpeed`. As mentioned in the `doc` field, this service sets the speed of the mobile. It can return the `INVALID_SPEED` execution report if the speed chosen is invalid. The service has one input: the value of the speed, which is to be stored in the component's database `db`. We assigned it a default value of `DEMO_SLOW`.

The service has only one control codel (executed by the control thread), which tests the validity of the speed value. This codel uses the input, as indicated in the `exec` field.

7.

```
service getSpeed {
   doc: "Returns the default speed";

   /* input/output parameters */
   output: db.speedRef "Current mobile speed";
}
```

Another service, which *returns* the current speed of the mobile. There

is no codel, since there is nothing special to do here. The output is taken from the database, which is always up-to-date.

8.

```
service moveDistance {
   doc: "Translates of a given distance";

   thread: motionTask;
   fail-msg: TOO_FAR_AWAY;

   stops: moveDistance;

   input: db.distRef:=0.0 "Distance to move";

   control codel {
      exec: demoControlDistance(db.distRef);
   }

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

   codel goto {
      exec: demoGotoPosition(db.distRef);
      update: db.distRef, state;
      on-inter: stop;
      next: goto, stop;
   }

   inter codel stop {
      exec: demoStopMobile();
      update: state;
      next: ;
   }
}
```

This service translates the mobile of the distance given as an input parameter (which default to `0.0`). There are four codels. The first one controls the validity of the input, and is executed by the control thread. The three other codels are executed by the `motionTask` thread. The `start` codel is the one that is executed first (but after the control codel has been successfully executed). The `goto` codel actually moves the mobile. It is periodic, and it will typically implement a servoing: at each period, it would computes a reference, then send its reference to the motors, and return. The `inter` codel is used both when the caller of the service wants to stop the motion,

or when the mobile has moved of the requested distance.

What is new here is the use of the `stops` token: it indicates that if the `moveDistance` service is invoked twice, the second activity would stop the execution of the first one (for short: this service is not reentrant). Thus, the component executes only the last motion request (of course, one could have implemented other strategies).

One can also notice the use of the `update` token, which indicates which codels modify the `state` structure (declared in the beginning of the component description).

9.

```
service stop {
   doc: "Stops the mobile";

   stops: moveDistance;
   delays: all;
}
```

This service has no codel, but still performs interesting action: the `stops` field indicates that if this service is requested, it will stop any running instance of the service `moveDistance` (defined above). Thus, invoking the service `stop` will stop the mobile motion. Moreover, the `delays` token indicates that while the `stop` service is running, any other request for a new service will be delayed until the stop has completed.

10.

```
service monitor {
   doc: "Monitor a particular mobile's position";
   type: request;

   thread: motionTask;
   fail-msg: TOO_FAR_AWAY;

   stops: none;

   /* input/output parameters */
   input: db.monitor:=0.0 "Position to monitor";
   output: db.state.position "Actual position";

   start codel monitor {
      exec: demoMonitor(const db.monitor,
                        const db.state.position);
      next: monitor;
   }
}
```

Lastly, this service describes a monitoring task. Given a particular position, it will run until the mobile pass through this position, and return the actual position (which might not be exactly the same as the monitored value, because of time discretization).

The sole codel uses two references to the database. Note the use of the `const` keyword, that indicates that the codel does not modify the values.

7 Conclusion

We have presented the draft specification of a *component's interface description language*. This specification has much in common with the one defined by G$^{\text{en}}_{\text{o}}$M, but several improvement have been added.

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.

This specification could be implemented in the near future within a new version of G$^{\text{en}}_{\text{o}}$M.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337, April 1998.

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

[3] S. Fleury, M. Herrb, and A. Mallet. Genom user's manual. Technical report, LAAS-CNRS, December 2001.

— December 14, 2001. Revision 1.0 —