A role-based component architecture for computer assisted interventions: illustration for electromagnetic tracking and robotized motion rejection in flexible endoscopy.

Release 2.00

Jean-Baptiste Fasquel[1], Guillaume Chabre[1], Philippe Zanne[2], Stéphane Nicolau[1], Vincent Agnus[1], Luc Soler[1], Michel de Mathelin[2] and Jacques Marescaux[1]

July 3, 2009

[1]Computer Science Research and Development Department, IRCAD, France
[2]University of Strasbourg, France

Abstract

This paper presents an original role-based software architecture facilitating the flexible composition, configuration and collaboration of separated components in the field of computer assisted interventions. Roles, which can be seen as methods dynamically attached to objects, are embedded in components, to limit build level dependencies and improve flexibility. An appropriate component definition and composition language is proposed to declare softwares, without any specific initialization or glue code, this remaining a challenging issue in component oriented programming. The potential of this architecture is illustrated for a software coupling electromagnetic tracking with a robotized system dedicated to the physiological motion rejection in flexible endoscopy. This software consists in several independent components with are combined at runtime thanks to a concise XML-based declaration.

Contents

1 Introduction

This paper faces the design of a framework for the rapid prototyping of softwares in computer assisted interventions, this being currently an active area of interest [3], with existing open source IGT architectures

such as CISST[7], Slicer IGT[6] or IGSTK[4].

The main originality of the proposed C++ cross-platform (Windows, Linux, Mac Os X) architecture relies in the coupling of a component approach with the notion of role-based programming [8] in this application. Roles define the composition by dynamically associating different tasks to the same conceptual entity although they are defined in separated code elements, similarly to dynamic inheritance. Such a mechanism is not supported by traditional object oriented languages and in particular by the C++ language traditionally used in our application domain for runtime performances reasons. Role orientation facilitates component collaboration as component instances share a common data support (the base object). Component orientation aims at removing, thanks to role abstraction, build level dependancies [9].

On top of this approach, we propose a concise *component definition and composition language* [9] facilitating the definition of applications independently from the implementation language, without any specific glue code and therefore with a high degree of flexibility, this being challenging [9].

Compared to our previous work [5], this paper presents to two main improvements. In terms of architecture, we simplified both class distributions (e.g. merging of both `ExtPtService` and `ExtService`) and the XML-based description formalism. Moreover, specific initialization or glue code is not required anymore, allowing to entirely define applications with a pure XML-based declaration. In terms of functionality, compared to [5], the application field is extended to robotized system management combined with an Aurora-based electromagnetic tracking [1].

2 Architecture

As traditionally considered in component oriented programming, the proposed framework is based on a strong separation between data (*base object*, directly inheriting from the `::layer::Object` in fig. 1, being data container restricted to getters and setters) and functionalities (*services*, being implementations of the `::layer::IService` interface in fig. 1).

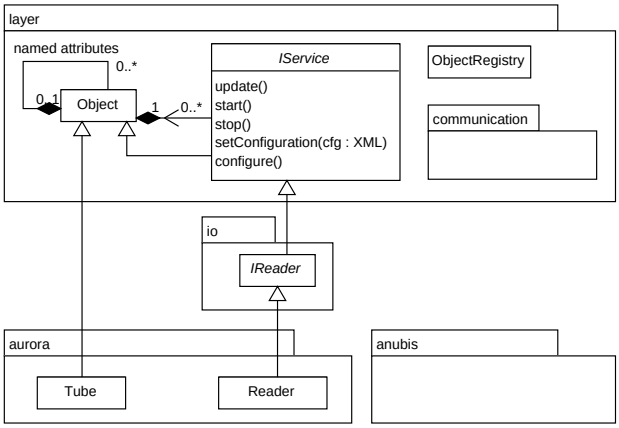

Figure 1: Simplified UML class diagram of the architecture (the `layer` package being the core of the architecture). The `io` package defines a functionality type (`IReader`), while `aurora` package (only partially described for clarity) represents an implementation package (a specific base object (`Tube`) and a specific service (`Reader`)). The `anubis` content is not detailed for clarity.

Services (roles) can be dynamically associated to a base object, using, at code level, a generic invocation

such as: `::layer::add(myObject,"::io::IReader","::aurora::Reader")`, where `myObject` is an instance of `::aurora::Tube`, and will play the `::io::IReader` role type. Note that the notion of service type facilitates both code factorization and object classification (useful to perform some tasks according to the type, independently from effective low-level implementations). Service state (attributes' value) is defined using an XML like structure passed as parameter (`IService::setConfiguration(cfg:XML)` and `IService::configure()` methods in fig. 1).

```
<plugin id="aurora">
 <library name="libAurora"/>
 <point id="::aurora::Reader" schema="AuroraReader.xsd">
  <implements>::io::IReader</implements>
  <implements>::aurora::Tube</implements>
 </point>
</plugin>
```

Figure 2: Example of component description file content: `point` declares, with a uniform formalism, that `::aurora::Reader` inherites from `::io::IReader` and can play a reading role for objects of type `::aurora::Tube`.

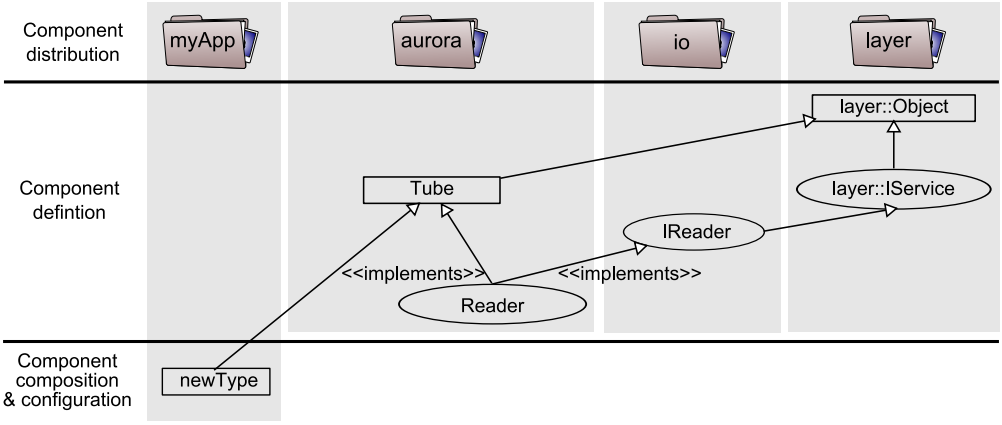

Figure 3: The complexity of the component distribution (top) is hidden by the XML-based graph provided by their descriptions (middle). Applications (bottom) can be seen as new types (`newType`, see figure 4 for full declaration), possibly declared in separated components (e.g. `myApp`), consisting in particular compositions and configurations of base objects and services. Note that `myApp` can be restricted to a description file only (i.e. no dynamic library). All together forms a uniform object-oriented like graph integrating both definitions and new declarations. Thanks to the `implements` XML statement, `Reader` seems to inherite from `Tube`, although it concerns role declaration only.

At initialization (`IService::start()` method in fig. 1), a service can dynamically affect shared attributes (optional attributes [5]) to its base object (natively restricted to hard-coded intrinsic attributes [8]).

At runtime, depending on the functionality, a service can perform computations using its own attributes, intrinsic attributes of its base object, or even a combination of all. If the service execution depends on attributes of another service, it is preferable (inter-service independency) that this other service exports the required attributes to its base object, as shared attributes. Therefore the union of the base object's intrinsic and shared attributes represent the data support for service execution (i.e. `IService::update()` method in fig. 1). Service execution ordering (chaining) can be managed in a generic manner using a list of identifiers

(*explicit chaining*), as each service can be retrieved from a unique identifier (e.g. invocation looking like `::layer::get(myObject,serviceUID)->update()`). Chaining can also be implicit (*implicit chaining*) using the event-based collaboration mechanism previously described [5] (`communication` entity in fig. 1).

Thanks to the abstraction provided by the `layer` (e.g. `add`, `get` methods and `IService` API), we can avoid build level inter-service dependencies by embedding services in components [5]. A component is defined by a XML description file (e.g. see figure 2), (optionally) coupled with a dynamic library and ressources (e.g. icons,...), being bundled into a single entity (e.g. a directory of a filesystem). The dynamic library embeds the implementation of a (set of) service(s) defined in the XML description file. When requesting an attachement (e.g. `::layer::add(myObject,"::io::IReader","::aurora::Reader")`), the component management system (wrapped in the `layer` package) analyzes the description files of the component distribution, loads the appropriate dynamic library (e.g. `libAurora`) to instanciate the appropriate service and finally to store the resulting association in the `::layer::ObjectRegistry` (singleton storing all instanciated base objects and attached services). The formalism used to describe components (*component definition language*) leads to a graph (see figure 3) similar to an inheritance hierarchy (where each class is declared with a XML element named `point`). Such a graph-based representation appears more concise and structured than previously [5], and hides the complexity and the specificity of the component distribution (as illustrated by figure 3).

```
<extension id="newType" implements="::aurora::Tube" >
<object uid="root" type="::aurora::Tube">
  <service uid="reader" type="::io::IReader" implementation="::aurora::Reader" >
   <location>...</location>
  </service>
  <service uid="render" type="::render::IRender" implementation="::aurora::Render" >
   <window id="900"/>
  </service>
  <service uid="tracker" type="::tracker::ITracker" implementation="::aurora::Tracker" />
  <service type="::gui::IAspect" implementation="defaultAspect" >
    <windows>
      <window id="900"/>
    </windows>
  </service>
  <start type="::gui::IAspect" />
  <start uid="tracker" />
  <start uid="reader" />
  <update uid="reader" />
  <stop uid="tracker" />
  <stop type="::gui::IAspect" />
</object>
</extension>
```

Figure 4: XML-based software declaration: Aurora-based electromagnetic tracking (`tracker`), including 3D rendering (`render`) in a specific view (corresponds to the left side of the snap shot given in fig. 7). The `::aurora::Tube` base object is a set of points, their position being modified by the tracker, involving 3D rendering refresh (implicit chaining).

On top of this, new types (i.e. base objects with specific service attachements to be performed at instanciation time) can be defined using a dedicated XML declaration (*composition language*), as illustrated in figure 4. The root XML element `extension` enables to integrate this declaration into the graph resulting from component definition, leading to a uniformization of both definitions and compositions. Note that optional

XML elements can specify services to start (`start` statement) and execute (`update` statement) when ending instanciation, and those to stop (`stop` statement) when destroying the object (taking the declaration order into account). Compared to our previous work [5], such a declaration is used to define a complete software (evaluated for the VR-Render freeware [2]), using a generic launcher (the root object type, e.g. `newType`, being a parameter), independent from any component (dependency limited to the `layer` entity) and without glue code. Note that service existence and attachment compliance are checked through a graph analysis. Services' state (children of `service`-named XML elements) can be checked using XSD schemas provided as component ressource files and specified XML descriptions (e.g. `schema` attribute value of `point` XML element in figure 2).

```
<extension id="illustration" implements="::layer::Object" >
<object uid="root" type="::layer::Object">
 <object uid="aurora" type="::aurora::Tube">
  <service uid="reader1" type="::io::IReader" implementation="::aurora::Reader" >
   <location>xxx</location>
  </service>
  <service uid="render" type="::render::IRender" implementation="::aurora::Render" >
   <window id="900"/>
  </service>
  <service uid="tracker" type="::tracker::ITracker" implementation="::aurora::Tracker" />
 </object>
 <object uid="anubis" type="::anubis::Robot">
  <service uid="reader2" type="::io::IReader" implementation="::anubis::Reader" />
  <service uid="motionFilter" type="::layer::IControler" implementation="::anubis::Motion" >
   <uid>s1</uid>
   <uid>s2</uid>
   <uid>s3</uid>
  </service>
  <service uid="s1" type="::tracker::ITracker" implementation="::anubis::Tracker"/>
  <service uid="s2" type="::anubis::ICommand" implementation="::anubis::Command"/>
  <service uid="s3" type="::fwRender::IRender" implementation="::anubis::Image">
   <win guiContainerId="902" />
  </service>
  <service uid="Ctrl" type="::gui::IEditor" implementation="::anubis::HeadDriver">
   <win guiContainerId="901" />
  </service>
 </object>
 <service type="::gui::IAspect" implementation="defaultAspect" >
  <windows>
   <window id="900"/>
   <window id="901"/>
   <window id="902"/>
  </windows>
 </service>
 <start type="::gui::IAspect" />
 ...
 <stop type="::gui::IAspect" />
</object>
</extension>
```

Figure 5: XML-based declaration of the application managing the robotized flexible endoscope (`anubis`) coupled with the tracking system (`aurora`).

3 Illustration

The XML declaration of the application considered in this paper is given by figure 5. Figure 6 (resp. 7) concerns the experimental setup (resp. the software). Both `anubis` and `aurora` objects are declared as independent elements composing the application `root` object. This root object has only one `IAspect` service in charge of configuring the overall layout of the GUI. The core of the previous `newType` declaration (figure 4) has been entirely reused without any code modification or specific glue code.

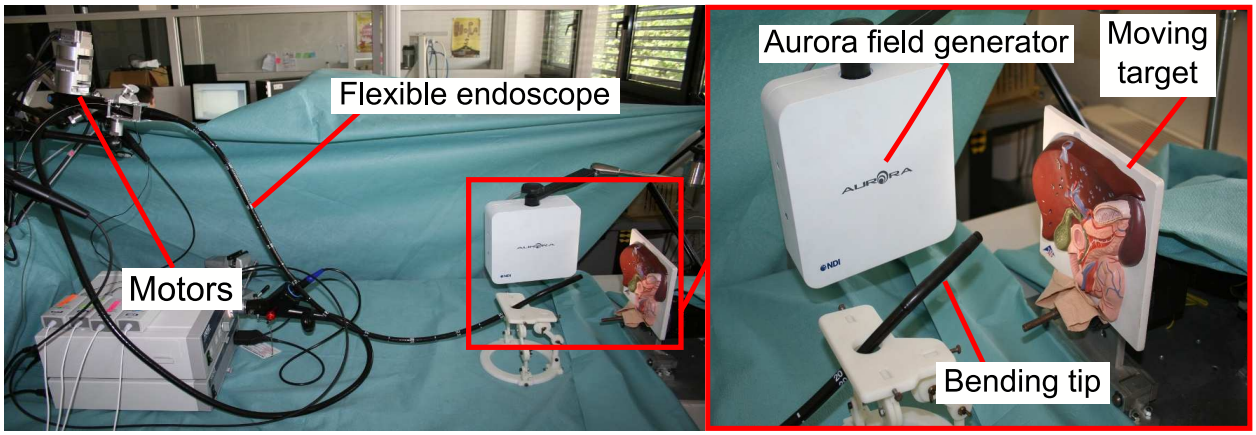

Figure 6: Experimental setup related to the system managed by the application (snapshot in fig. 7).

The `aurora` object focuses on the monitoring of flexible endoscope motion using an electromagnetic tracking system: a set of sensor coils are regularly placed on a catheter (`::aurora::Tube` in figure 5) which is introduced in a channel of the flexible endoscope. Associated services perform tracking as well as 3D rendering (both `tracker` and `render` services in figure 5) in a predefined view area of the application (right panel in fig. 7, corresponding to the window identifier `900` in the declaration). For this functionality, the native event-based mechanism is used to synchronize both tracking and rendering (similarly to the IGSTK approach [4] with notions of `SpatialTransform` and `SpatialObject`).

The `anubis` object is mainly dedicated to motion cancelation in flexible endoscopy, in the case of the robotized system described in [10]. This system is represented by a specific base object (of type `::anubis::robot`), which mainly consists in data related to both motors and the image (video) being visualized by the head of the endoscope. A tracking service performs the visual tracking (`s1` in fig. 5), and a command service (`s2` in fig. 5) controls endoscope motors. GUI aspects have been implemented in separated services, preserving the control loop (`s1` and `s2`) from the GUI specificity, regarding both the video rendering and the interactive definition of the target to be tracked (`s3` in fig. 5). In addition to motion filtering, the user can control the orientation of the head of the flexible endoscope (`ctrl` in fig. 5). Due to the video frame rate, a complete control loop cycle must be shorter than 40 ms. For this reason, a control service (`motionFilter` in figure 5), triggered by video acquisition (`reader2` in figure 5), manages both control loop and video rendering refresh (explicit chaining). As rendering is optional, the `s3` service is executed only if enough time remains. Despite the proposed abstraction (well known to reduce performances due to indirections), it has been observed that runtime performances were compliant with requirements. In our sense, this is facilitated by the role orientation, as services collaborate through a direct access to their common base object content, preserving performances. Indirections mostly concern system initialization (e.g. service attachement and configuration) and observation based collaborations (which can be, as for the considered control loop, replaced by explicit chaining, depending on required reactivity).

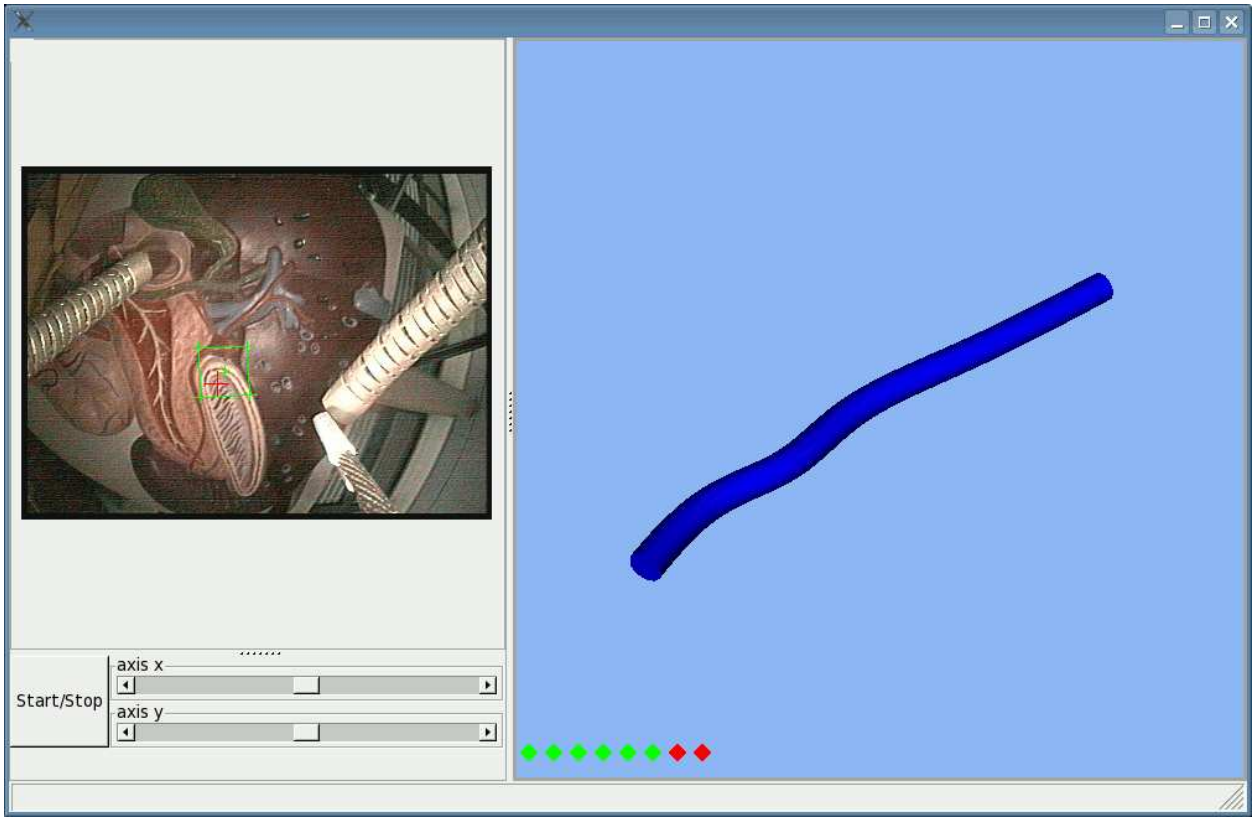

Figure 7: Snapshot of the illustrative application. Lef panel: part dedicated to the robotized system, including endoscopic view (with tracked target in green), endoscope head control (both sliders) and button to start/stop the `motionFilter` service (see XML declaration 5). Righ panel: part dedicated to 3D rendering of the catheter deformation tracked by the Aurora system.

4 Discussion

In our sense, the strength of the role orientation in the presented framework is that component composition is natively supported (common data support), as well as behavioral collaboration using the integrated observer design pattern (used in the illustration for electromagnetic tracking). Component abstraction and (build level) independencies facilitate application prototyping with a concise XML declaration without specific glue code or initialization code. Due to direct access to the base object, it seems that such an architecture, despite its abstraction, can be advantageously used for critical applications such as the one presented here. Besides, due to the well structured organization of base objects and services, including communications and, at service level, the state pattern, any application can be easily monitored at runtime. Such monitoring capability appears essential for debugging and could be avantageously used for managing safety, this being critical in softwares dedicated to computer assisted interventions.

References

[1] http://www.ndigital.com/medical/aurora.php, 2009. 1

[2] VR-Render, http://www.ircad.fr/softwares/vr-render/software.php, 2009. 4

[3] K. Cleary, S. Aylward, P. Kazanzides, K. Vosburgh, R. Ellis, J. Duncan, K. Farahani, H. Lemke, T. Peters, WB. Lorensen, D. Gobbi, J. Haller, LL. Clarke, S. Pizer, R. Taylor, R. Jr Galloway, G. Fichtinger, N. Hata, K. Lawson, C. Tempany, R. Kikinis, F. Jolesz, S. Dimaio, and Kapur T. Challenges in image-guided therapy system design. *Neuroimage*, pages 144–151, 2007. 1

[4] A. Enquobahrie, D. Gobbi, M. Turek, P. Cheng, Z. Yaniv, F. Lindseth, and K. Cleary. Designing tracking software for image-guided surgery applications: IGSTK experience. *International Journal of Computer Assisted Radiology and Surgery*, 3(5):395–403, 2008. 1, 3

[5] J.-B. Fasquel, J. Waechter, L. Goffin, S. Nicolau, V. Agnus, L. Soler, and J. Marescaux. A XML based component oriented architecture for image guided surgery: illustration for the video based tracking of a surgical tool. In *Insight Journal, Workshop on Systems and Architectures for Computer Assisted Interventions, 11th International Conference on Medical Image Computing and Computer Assisted Intervention*, 2008. 1, 2, 4

[6] N. Hata, S. Piper, F. Jolesz, C. Tempany, P. Black, S. Morikawa, H. Iseki, M. Hashizume, and R. Kikinis. Application of open source image guided therapy software in MR-guided therapies. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, 2007. 1

[7] A Kapoor, A. Deguet, and P. Kazanzides. Software components and frameworks for medical robot control. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 3813–3818, 2006. 1

[8] B.B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2:143–160, 1996. 1, 2

[9] K.-K. Lau and Z. Wang. Software component models. *IEEE Transactions on software engineering*, 33(10):709–724, 2007. 1

[10] l. Ott, F. Nageotte, P. Zanne, and M. de Mathelin. Simultaneous physiological motion cancellation and depth adaptation in flexible endoscopy. *IEEE Transactions on Biomedical Engineering*, pages 1–4, 2009. 3