# Automatic code generation based on generic description of intelligent instrument

Stéphane Perrin, Eric Benoit, Laurent Foulloy

IEEE Int. Conf. on Systems, Man and Cybernetics (SMC'02), CD-ROM WA2Q5, Hammamet, Tunisia, October 2002

# Automatic Code Generation based on Generic Description of Intelligent Instrument

Stéphane Perrin, Eric Benoit, Laurent Foulloy

LAMII-ESIA Université de Savoie

B.P. 806, F-74016 Annecy cedex, France

stephane.perin@univ-savoie.fr, eric.benoit@univ-savoie.fr, laurent.foulloy@univ-savoie.fr

**Abstract** - ***This paper presents an automatic outputs generation based on XML description of intelligent instruments. This description uses the service based approach called INOMs. The XML representation of intelligent instruments enables three different outputs: user informations output in html format, supervisor information output, and source code output for selected field bus network target. The intelligent instrument model is briefly described and the area of interest consists in the automatic code generation. A basic example is presented in order to illustrate the different steps and to show the simplicity of the operational implementation.***

**Keywords*: Automatic Generation Code, XML, Intelligent Instrument, Generic Description.***

## I INTRODUCTION

Intelligent instruments, i.e. intelligent sensors and intelligent actuators, are now commonly used in industry. Their design is performed both by software engineer and by physicist who not necessary understand each other. Recent studies propose instrument models based on a set of functionalities organized with a general behavioural description, i.e. automation graph or object model [1][2][3]. The idea of this paper is to propose a solution that cuts the design process into two pieces based on the same instrument modelling. The aim of this approach is to simplify the activity of the software engineer and the physicist with respect to their respective competencies.

This paper is a extension to recent works about based service approach [4] and a dedicated tool: CAPtool [5][6], that is able, to verify the model and now, to produce generic XML description of intelligent instruments. Wollschlaeger discusses about the representation of CANopen device profile in XML language[7]. He shows that specific informations for service and support can be extracted from a general XML based description. Based on XML format, the intelligent instrument representation can also be used to generate code source as well as user information can be created in html format or supervisor information i.e. Electronic Data Sheet (EDS) can be created in dedicated format. In this paper we propose to extend this approach to source code generation of the instrument into a source file that will be used for several translation in order to produce a source code.

## II SERVICE BASED INTELLIGENT INSTRUMENT MODELLING

### *A. Principle*

This section presents the chosen modelling for intelligent instrument. Recent studies propose models based on a set of functionalities organized with a general behavioural description, i.e. automation graph or object model [2][8][9]. The internal modelling of intelligent instruments is not sufficient for the design of large applications. Obviously, intelligent instruments need to inter-operate. Therefore an external model of intelligent instrument is required. Staroswiecki proposed to model a sensor by a set of services [4]. Services are organized into subsets called "USer Operating Modes" (USOM). In this model, a sensor service can be requested, and so serviced, only if the current active USOM includes this service. This prevents the requirement of services when they cannot be available.

The approach discussed in [10] was proposed to model existing instruments from the external point of view. In particular, the external model of the instrument can be used to build a global model for an application involving several instruments. This kind of approach can also be used to define the internal functional model of a sensor [11].

Instruments are considered as entities that offer some more or less complex services. These services represent the instrument functionalities from the user point of view. At a lower level, each instrument service is defined as a set of internal services Fig. 1.

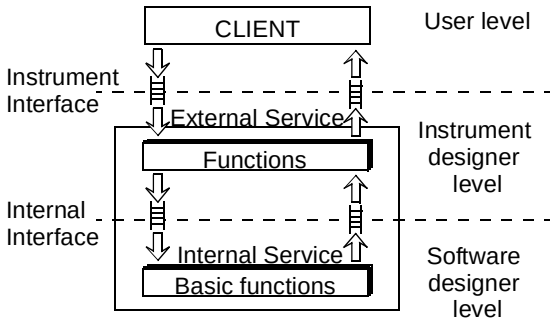


Fig. 1. Service based structure of an intelligent instrument.

These levels are representative of the gap between the instrument user point of view, the instrument designer point of view and the software designer activity. In order to use instrument designer capabilities for the design of intelligent

instruments, instrument functionalities are described with basic internal services. Then the designer will just have to define each external service with a set of internal services.

### *B. Modelling*

The software designer creates basic pieces of code, i.e. internal services, using the usual C language. Then, he defines all possible sequences between internal services. Finally he creates internal modes as sets of internal services.

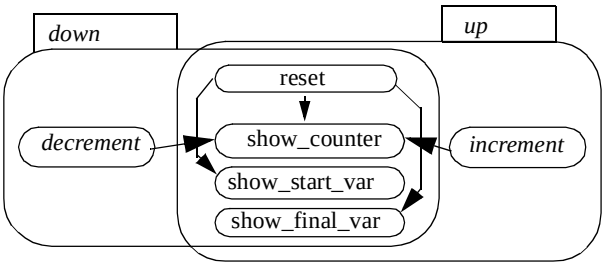


Fig. 2. Internal services and internal modes of the counter example.

The Fig. 2. shows an example of the graphical model produced by the software designer. In this example, the instrument handle a counter. When the counter is initialised, the counter value is set to 0, the initial value is set to 0 and the final value is set to 10. The default mode is *up*. Then counter value can be incremented while it is less than final value. When the counter value is greater or equal than final value, the counter switch to the *down* mode. Then counter value can be decremented while it is greater than final value.

Internal service *reset* sets the *counter* value to 0 and the *final_var* to 0. The *show_x* internal services, e.g. *show_counter*, send the value of the associated variable on the fieldbus connected to the instrument. *Decrement* and *increment* internal services modify the counter each time they are called. The internal data flow is included into the definition of internal services and does not appear at the model level. The designer have to indicate input and output variables: three variables are used to perform this instrument: *count_var* contains the current value of the counter (output), *start_var* contains the start value of the counter (input/output) and *final_var* contains the final value of the counter (input/output).

The instrument designer creates each external service by defining which internal service is started, and which other internal service is used by this external service. Then he creates external modes as sets of external services (Fig. 3.) and defines the authorized transitions between modes (Fig. 4.). In this example, the instrument designer defines the external service *count* that uses the *show_counter* internal service in order to make the service *count* sending the counter value over the network.

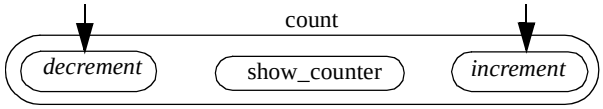


Fig. 3. Model of the external service *count*

The functional description describes the implementation of those functional requirements. Fig. 5. represents the generating of specific output descriptions for a device.

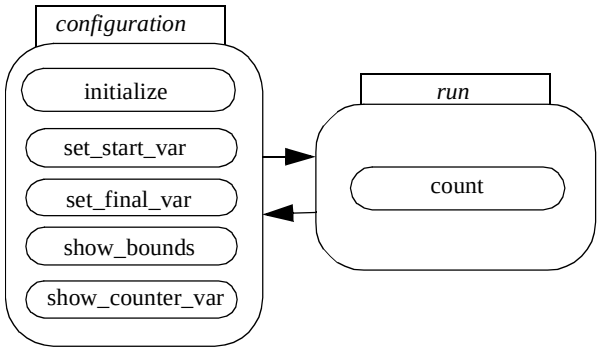


Fig. 4. External services and external modes of the counter example.

From the graphical modelling of the intelligent instrument a global generic device description file is created. A graphical user interface may be used for creating this XML description.

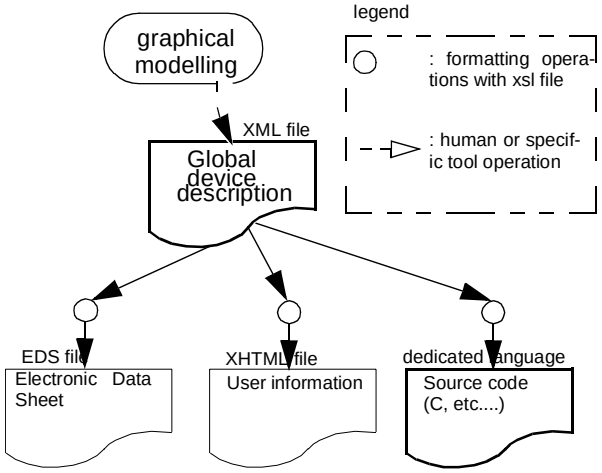


Fig. 5. Principle of using XML for intelligent

## III XML REPRESENTATION OF THE MODEL

The produced XML file contains all information included in the graphical modelling. The XML file of the example can be seen in the annexe. The C format definition of internal services is not included into the XML file and stays in this format for convenient use. The XML representation of internal services then includes services names and the sequence between services. A small piece of the XML file of the example is presented below:

<!--Internal services definitions -->

```
<iservice id="reset">
  <starts>show_counter</starts>
  <starts>show_start_var</starts>
  <starts>show_final_var</starts>
</iservice>
<iservice id="decrement">
    <starts>show_counter</starts>
</iservice>
<iservice id ="increment">
    <starts>show_counter</starts>
</iservice>
<iservice id="show_start_var" />
<iservice id="show_final_var" />
<iservice id="show_counter" />
```

An XML based representation file can easily be transformed into an other file (i.e. XML, or HTML, etc....). This can be performed by a set of rules. These rules are expressed using extensible style language for transformation (XSLT) and can be contained into XSL files. Each XSL file appears as conventional files that includes some transformation rules. Indeed an XSL file dedicated to the generation of an HTML file includes essentially HTML markers, and some transformation rules that translate XML information into HTML markers.

For example, if the XSL file includes:

```
<H1>List of services</H1>
<xsl: for-each select="instrument/iservice">
internal service: <xsl:value-of select="@id" /><BR/>
</xsl:for-each>
```

then the translation of the XML file of the counter example by this file will includes:

```
<H1>List of services</H1>
internal service: reset<BR/>
internal service: decrement<BR/>
internal service: increment<BR/>
internal service: show_start_var<BR/>
internal service: show_final_var<BR/>
internal service: show_counter<BR/>
```

This is a pure HTML file. The same approach can be used for the generation of EDS or source code. For example, an XSL file dedicated to the generation of C source code is essentially a C source code with some rules that translate XML information into C items. For example, the following XSL file translate the XML example file into a C file.

```
void initializeIServices(void){
<xsl:for-each select="instrument/iservice">
start_<xsl:value-of select="@id"> = 0;</xsl:for-each>
}
```

This translates the XML file into:

```
void initializeIServices(void){
start_reset = 0;
start_decrement = 0;
start_increment = 0;
start_show_start_var = 0;
start_show_final_var = 0;
start_show_counter = 0;
}
```

## IV FROM GRAPHICAL MODEL TO FINAL SOFTWARE

This kind of approach allows the designer to be independent of a field bus technology. Therefore its developments can be directly used for a dedicated field bus. Thus, from a only one XML representation of instrument intelligent, source code files can be generated. The specific rules, in regards to field bus network specifications, is expressed into XSL files. So, to obtain source code for a dedicated field bus, it is just necessary to add corresponding transformation rules using XSLT language. No other dedicated tools are necessary. Fig. 6. illustrates the code generation procedure.

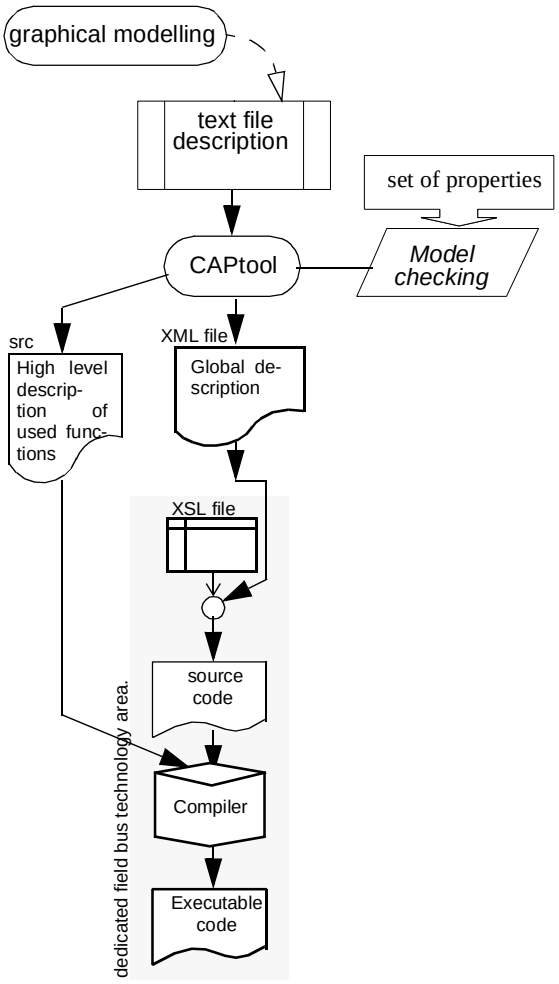


Fig. 6. Implemented approach for designing intelligent instrument based on XML representation.

In order to improve the safety of the instrument design, a set of properties must be satisfied [12]. These properties can be imposed during the design by limitations of the modelling. They can also be checked after the design. Most of the modelling steps are based on the *set and graph theories*. In a first step the design is defined graphically. Then, the model properties are translated into a text file. This translation is presently performed manually.

The generated text file is verified by the dedicated tool CAPtool. This one verifies all the properties that cannot be imposed by the graphic aspects of the model. An extension of this tool has been developed in order to perform the XML representation of intelligent instrument. Furthermore, the same tool is able to check properties and creates the global description device file. The high level description of internal services is simply extracted then added to a software library.

In the last analysis, the global description device file (XML format) fully describes the intelligent instrument functionalities. To obtain the corresponding source code for a dedicated field bus technology, the designer have just to invoke the set of rules (XSL/XSLT format) which translates the global description into the source code according to the dedicated field bus technology. From this global expression, the source code for dedicated field bus technology can be created.

For a dedicated technology i.e. CANopen, Echelon, etc...., the executable code is created from the global description (XML file), a set of transformation rules (XSL file) and the high level description file, i.e. C language. Note that the XML based description could be used any way for the last one. Actually, high level language is preserved to simplify the internal service description by the user.

Finally, the compiler can be solicited to obtain the executable code. It use the generated source code, the library that includes the code of internal services and a library dedicated to the chosen field bus. This last library includes all software pieces that depend from the field bus but that are independent from the instrument behaviour.

To add a new supported field bus technology, it just necessary to provide a set of transformation rules (XSL file) See Fig. 7. In addition, the XLS file is easy to be developed and is serviceable for others intelligent instruments.

Writing XSL files for the generation of source code is a solution chosen in order to reduce the complexity of the code generation. This approach simplifies the adaptation of the automatic generation mechanism to a specific fieldbus. Indeed, as the differences between field buses are very important and concern generally the highest layer of the OSI model, the differences in the source code are also very important.

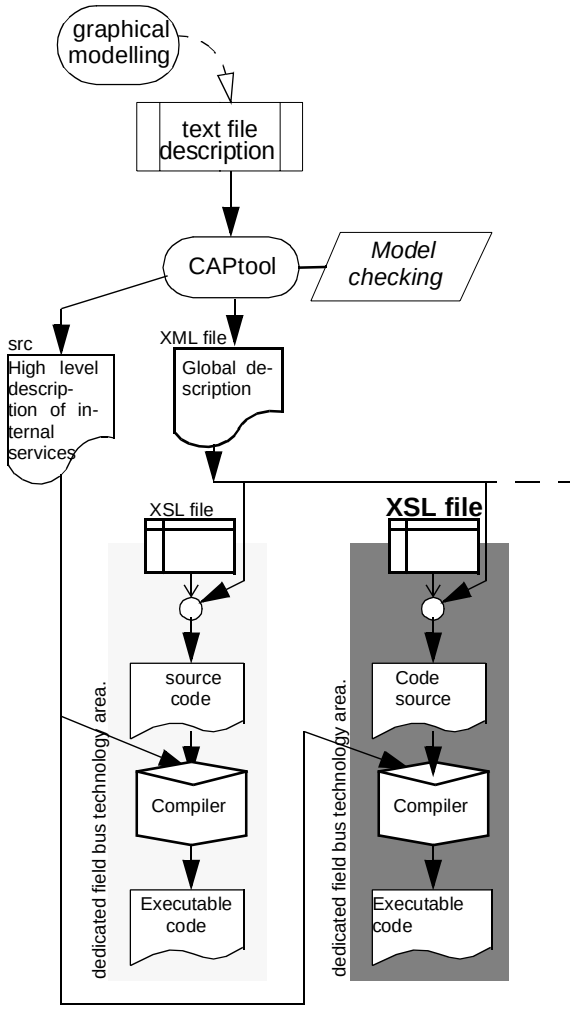


Fig. 7. Adding field bus technology in the designing of intelligent instrument based on XML representation.

The solution is first to define the model of a simple example, then to write manually the source code associated to this example, and dedicated to the chosen fieldbus. Then each source file that depends on one or more item of the INOM model is used as a basis for the definition of the associated XSL file. After all XSL files are written, they are inserted into the automatic code generation system Fig. 7.

This approach allows the automatic generation of source code that respects "safe programming" constraints. For example, a source code that contains no arrays can be easily generated this way. Naming rules for variables can also be applied. This translation method is not reserved to the generation of C source code. Indeed, C++, java or LonWorks files can also be generated using the same approach. The only limitation is the capability of the generated code to use the librarian created with the C definition of internal services. Presently this limit do not make possible to generate a VHDL source of the instrument. It would be possible to suppress this limitation with an XML definition of internal services.

## V CONCLUSIONS

The design of intelligent instruments requires more and more multiple competencies. The design complexity of instruments requires several specialists such as, sensor or actuator designers, and software designer. The model approach and its global description, proposed in this paper, allows the designer to be independent of filed bus technology. Source code can be automatically created for any field bus technology if a dedicated set of rules is provided. In addition, it is easy to implement these rules with XSL/XSLT format. The approach chosen is compatible with the application model based on USOMs.

## ANNEXE: XML FILE

```
<?xml version="1.0" ?>

<instrument>
<!-- Software designer section -->
<!--Variables definition -->
<variable id="start_var" type="unsigned8" acces="io"/>
<variable id="final_var" type="unsigned8" acces="io"/>
<variable id="counter_var" type="unsigned8" acces="io"/>

<!--Internal services definitions -->
<iservice id="reset">
  <starts>show_counter</starts>
  <starts>show_start_var</starts>
  <starts>show_final_var</starts>
</iservice>
<iservice id="decrement">
    <starts>show_counter</starts>
</iservice>
<iservice id ="increment">
    <starts>show_counter</starts>
</iservice>
<iservice id="show_start_var" />
<iservice id="show_final_var" />
<iservice id="show_counter" />

<!--Internal modes definition -->
<imode id="down">
    <includes>reset</includes>
    <includes>show_counter</includes>
    <includes>show_start_var</includes>
    <includes>show_final_var</includes>
    <includes>decrement</includes>
</imode>
<imode id="up">
    <includes>reset</includes>
    <includes>show_counter</includes>
    <includes>show_start_var</includes>
    <includes>show_final_var</includes>
    <includes>increment</includes>
</imode>

<!-- Instrument designer section -->
<!-- External services definition -->
<service id="count">
    <starts>increment</starts>
    <starts>decrement</starts>
    <uses>show_counter</uses>
</service>
<service id="initialize">
    <starts>reset</starts>
    <uses>show_counter</uses>
    <uses>show_start_var</uses>
    <uses>show_final_var</uses>
</service>
<service id="set_start_var">
    <uses>show_start_var</uses>
</service>
<service id="set_final_var">
    <uses>show_final_var</uses>
</service>
<service id="show_bounds">
    <starts>show_start_var</starts>
    <starts>show_final_var</starts>
</service>
<service id="show_counter_var">
    <starts>show_counter</starts>
</service>

<!-- External modes definition -->
<mode id="run" >
    <includes>count</includes>
</mode>
<mode id="configuration" >
    <includes>initialize</includes>
    <includes>set_start_var</includes>
    <includes>set_final_var</includes>
    <includes>swow_bounds</includes>
    <includes>show_counter_var</includes>
</mode>

<transition start="configuration" end="run" />
<transition start="run" end="configuration" />

</instrument>
```

## REFERENCES


[1] Bloch G., Eugene C., Robert M., Humbert C., “Measurement Evolution: from Sensors to Information Producer”, Proc. of IMEKO TC1 TC7, (September 8-10, 1993) London, UK, pp 335-341.

[2] J.M. Riviere, M. Bayart, J.M. Thiriet, A. Bouras, M. Robert, “Intelligent instruments: some modelling approaches”, Measurement and Control, (July-August 1996) Vol.29, pp.179-186.

[3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, “Object Oriented Modeling and Design”, Prentice Hall International, 1991.

[4] M. Staroswiecki, M. Bayart, “Models and Languages for

the Interoperability of Smart Instruments", Automatica (1996), Vol. 32, n° 6, pp. 859-873, .

[5] Benoit E. , Foulloy L., Tailland J. "Automatic Smart Sensors Generation Based on InOMs", Proc. of the 16th IMEKO World Congress. Vienna, Austria, Sept. 25-28 2000. Vol IX, pp. 335-340.

[6] Benoit E., Tailland J., Foulloy L., Mauris G., "A software tool for designing intelligent sensors", *in Proc. of the IEEE Instrumentation & Measurement technology IMTC/2000 Conference*, Baltimore, Marynland, USA, may 1-4 2000, pp. 322-326.

[7] Wollschlaeger M, Diedrich C., Thron M., "Generation of Role-specific information for an Entreprise integration Framework using XML description", *in ETFA 2001*, Antibes, France, Octobre 2001, pp 237-244.

[8] Q. Yang, C. Butler, "An Object-Oriented Model of Measurement Systems", *in Proc. of the IEEE/IMTC Instrumentation and Measurement Technology Conference*, Ottawa, Canada, May 19-21, 1997.

[9] H.J.W. Spoelder, A.H. Ullings, F.C.A. Groen, "Virtual Instrumentation: A survey of Standards and their Interrelation", *in Proc. of the IEEE/IMTC Instrumentation and Measurement Technology Conference*, Ottawa, Canada, May 19-21, 1997.

[10] A. Bouras, M. Staroswiecki, "How can Intelligent Instruments Interoperate in an Application Framework ? A Mechanism for Taking into Account Operating Constraints", Proc. of Int. Conf. SICICA'97 (9-11 juin 1997) Annecy, France.

[11] Benoit E., Foulloy L., "InOMs model: a service based approach to intelligent instrument design", *in ETFA 2001*, Antibes, France, Octobre 2001.

[12] Benoit E., Foulloy L., Tailland J., "InOMs model: a Service Based Approach to Intelligent Instruments Design", Proc. of the World Conf. on Systemics, Cybernetics and Informatics (SCI 2001), Vol. XVI, Orlando, USA, 2001, pp 160-164.