Automatic Generation of Control Software Components for CAN Devices by Using Java and XML

Gerd Nusser, University of Applied Sciences Reutlingen, University of Tübingen, Germany
Gerhard Gruhler, University of Applied Sciences Reutlingen, Germany
Wolfgang Küchlin, University of Tübingen, Germany

The integration of automation systems in a general management and service infrastructure, like for example a company's logistic and management network, as well as the use of high level languages to support state of the art software development is a major challenge in automation engineering. Embedding devices in information technology infrastructures and applying modern software development methods require the representation of devices on a higher level of abstraction. Once low level device facilities and communication details are encapsulated and represented by a software entity with respective interfaces, the application software development gets much more efficient.

This paper introduces a system which offers the opportunity to automatically discover and classify CANopen devices running in a CAN system. An XML description of the discovered devices and their interfaces is generated, which is then used to automatically generate Java control software for these devices including high level application interfaces. The XML description offers a lot of additional possibilities like automatic generation of web interfaces, integration in document management systems, access to database systems and Internet-based services in general. The current system setup consists of an Industrial PC-based controller running either Windows NT or the real time operating system RMOS offering a real time Java Virtual Machine named RTVM. The goal of the presented system is, not to be concerned with low level device facilities and communication protocols but to easily develop applications using high level components wherever possible to support modern software development processes and to seamlessly integrate automation systems in a general information technology infrastructure.

Introduction

Moving towards a general service infrastructure of cooperating services and devices, it is important to factor out details about different hardware and software platforms by enhancing the level of abstraction [1]. Therefore each device can be regarded as a special kind of service, which offers its services over some kind of network [3].

This paper presents a system which automatically generates software control components which encapsulate the hardware access and hardware dependent communication to CANopen devices. The generated software components are located at the controller site and have access to devices connected to a CAN fieldbus. A single component receives requests for a specific device and handles all communication details, like send and receive CAN messages, internally.

One major concern is to abstract away from device dependent technical issues like for example communication protocols and to offer a high level interface.

The generation of a high level interface simplifies resp. facilitates the integration into some kind of middleware technology, and the possibility to apply modern software concepts like for example (mobile) agents to the field of automation engineering.

The main purpose is the integration of devices in a general management and service infrastructure.

Approach

For the representation of devices, XML [17], a subset of SGML is used. XML is an "open, platform independent and vendor independent standard" [7], which is easy to parse and supported by many tools.
The basic approach for automatic generation of software components for devices consists mainly of the parts discovery (device identification), representation and code generation, as illustrated in figure 1. First the system searches for CANopen devices connected to CAN and represents their interface using XML. During the generation process different components can be generated automatically. As shown in figure 1 the system offers the possibility to generate Java components as well as web interfaces (HTML comp).

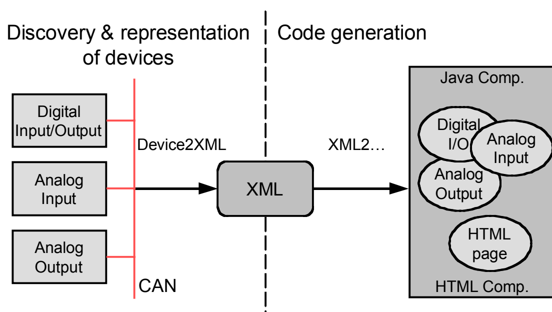

Figure 1: System Overview

The representation of devices in XML and the concept of automatic code generation based on these XML documents offer the following benefits:

1. The XML representation of device interfaces raises the level of abstraction and are (almost) independent of any implementation language. Furthermore, a wide range of different tools support the use of XML applications.
2. The process of automatic generation is less error prone and leads to more reliable and stable software applications. Additionally, it serves as a basis for rapid system prototyping and highly dynamic application environments, like for example agent based architectures [8].
3. High level interfaces support interoperability between different technologies, applications, and systems, with respect to information exchange.
4. Devices can easily be integrated into a distributed environment, such as the Internet, in common middleware architectures, or in a general information technology infrastructure, as the generated components directly support this kind of integration.
5. By making device representations persistent, structural query and retrieval facilities can be applied.
6. Due to the XML abstraction and dynamic generation, the overall development and integration process is speed-up tremendously.

System architecture

The current setup consists of an industrial personal computer named SICOMP [9], running either Windows NT or RMOS, a real-time operating system from Siemens AG. The system is equipped with various components (SMP16), like digital and analog components, and a CAN/CANopen component including CANopen master software.
CANopen uses an object-oriented approach to the definition of standard devices, where each device is represented as a set of objects that can be accessed through the network. All the objects are accessible through the object dictionary of a device. The entries in an object dictionary, i.e. the objects, are referenced by a simple addressing scheme consisting of an unsigned 16-bit index and an unsigned 8-bit subindex. To read and write to object dictionary entries low priority messages, so called service data objects (SDOs) are used. High priority messages, so called process data objects (PDOs) are used for real-time communication.
For CAN/CANopen development purposes the SICOMP system offers a C/C++

library. Three different I/O modules, namely DIOC711 (digital input/output), AIC711 (analog input) and AOC711 (analog output) are currently connected to CAN. All of these devices comply to the draft standard DS-401 “Device Profile for Generic I/O Modules” [6].
Supposed that appropriate peripherals and system support for CAN are available, the presented system may also be used on an embedded Java Virtual Machine. An implementation for a Java low cost embedded system equipped with CAN, named TINI, is currently under development (cf. section 0).

CANopenProvider

To access the previously mentioned CAN/CANopen library from inside Java, the library is encapsulated in a dynamic link library on the system level and encapsulated in a Java package called *CANopenProvider* interfacing the system library through the Java Native Interface (JNI). On the system level the CANopenProvider processes all of the CAN resp. CANopen specific communication details. On the Java language level the CANopenProvider offers a high level object oriented interface to CANopen. Currently the system does not support real-time capabilities, even though the SICOMP system offers a real-time Java Virtual Machine, called RTVM.
The higher level protocol CANopen is very well suited for introspection, as it is possible to gather all information of a device because of device profiles. Additionally each CANopen device holds an object dictionary with all internal parameters, which are described in detail in the electronic data sheet (EDS). An electronic data sheet contains all device parameters along with a description, data type, object type, access type, etc. As described in section 5 this information is used for documentation and component generation.
The system heavily depends on CANopen but only makes use of the mandatory requirements of CANopen and CANopen device specifications.

XML representation of devices

During startup the system first searches for available CANopen devices connected to CAN. This discovery process uses CANopen messages according to DS-401, which contains some pre-defined communication objects. One of these objects is the so called device type at index 0x1000, subindex 0x0. The return value of this entry consists of four bytes (cf. figure 2).

The first two bytes hold the device profile number (DPN). For generic I/O modules this DPN is 401, the specification identifier.

Additional Information		General Information	
Specific Functionality	I/O Functionality	Device Profile Number	
byte 4	byte 3	byte 2	byte 1

Figure 2: Device type entry ([6])

According to the DPN the third byte represents the I/O functionality of the device. For I/O modules this can be either digital input (DI), digital output (DO), analog input (AI) or analog output (AO) or any combination of these. The fourth byte defines some device specific functionality. Requesting the device type with different message identifiers each device is requested to send its type. If an answer arrives, a device with the demanded identifier exists. With this technique, all devices currently connected to CAN are discovered.
During the process of introspection, additional information about a specific device is gathered. The knowledge about the I/O functionality allows to read the so called PDO mapping parameters, which are directly mapped to the according inputs resp. outputs of the device. According to the device type the mappings are defined at well known object entries (0x1600 for DOs, 0x1A00 for DIs, 0x1601, 0x1602, 0x1603 for AIs and 0x1A01, 0x1A02, 0x1A03 for AOs). The object entries hold the number of mapped objects, the mapped objects themselves

and the corresponding data length. For example, for DOs with 8 output lines the default entry at index 0x1600, subindex 0x0, the number of mapped objects is 0x01. Index 0x1600, subindex 0x1 holds the first mapped object, which is mapped to index 0x6200, subindex 0x1 with a data length of 0x8. As a digital output device is considered, the implication is that this device has eight output lines.
Like shown in figure 3, the current setup consists of a digital I/O device with 8 input and 8 output lines, an analog input device

ID	Name	Type	Mapping	Length
4	Dx11	DI DO	6000,1 6200,1	8 8
8	AI11	AI	6401,1 6401,2 6401,3 6401,4	16 16 16 16
9	AO11	AO	6411,1 6411,2 6411,3 6411,4	16 16 16 16

Figure 3: Device setup

and an analog output device with four input resp. output lines. In contrast to digital I/O where each line is represented by a binary value 0 or 1, each analog input resp. output has a length of 2 bytes (16 bits).

XML representation of CANopen devices

To construct a general representation of a device, each device is characterized by its interface. The interface reflects the device functionality and is represented by a XML document. The XML document is compatible with the Object resp. Class Markup Language (OML, CML) [2]. CML defines objects, methods and parameters in a general manner, where each object consists of a number of methods and each method takes parameters and returns a value. As mentioned previously, a digital I/O device offers an interface to set the output lines, and to read the input lines. Additionaly, each CANopen device offers extended functionality, like e.g reading the device type or device name. Taken into consideration, that each of these attributes can be set resp. get, it is possible to identify a set of methods, along with parameters and return values. To introspect a device, a specification file is needed. This specification file is written in XML and contains objects and parameters defined in DS-401, along with data types and address information (index, subindex). Currently, this specification file contains information for generic I/O modules.
The discovery process and the introspection progress lead to the following XML document (excerpt):

```
<CML>
  <Class>
    <Name>Dx11</Name>
    <Package>Devices</Package>
    <Attributes>
      <NodeID>4</NodeID>
      <Types>
        <Type>DigitalInputDevice</Type>
        <Type>DigitalOutputDevice</Type>
      </Types>
    </Attributes>
    <Methods>
      <Method>
        <Name>DeviceType</Name>
        <Datatype>Unsigned32</Datatype>
        <Desc>Type of device</Desc>
        <Attributes>
          <Index>0x1000</Index>
          <Subindex>0</Subindex>
          <AccessType>r</AccessType>
        </Attributes>
      </Method>
      ...
      <Method>
        <Name>NrOfOutput</Name>
        <Datatype>Unsigned8</Datatype>
        <Desc>Number of output modules</Desc>
        <Attributes>
          <Index>0x6200</Index>
          <Subindex>0</Subindex>
          <AccessType>r</AccessType>
        </Attributes>
      </Method>
      ...
    </Methods>
  </Class>
</CML>
```

This document defines a class named *Dx11* which belongs to a package *Devices*. The node identifier is *4*, and the device is a combination of a *Digital-InputDevice* and a *DigitalOutputDevice*. Further on two methods are shown. The method *DeviceType* returns a value of type *Unsigned32* and is mapped to index *0x1000*, subindex *0x0*. The method *NumberOfOutput* returns the number of output modules. In this way, every

parameter defined in DS-401 is represented as a node in the XML document.

Automatic code generation out of XML representations

Once the interface of the connected devices are represented as standard XML documents, they can be easily processed and used to generate different components. Two generation processes will be presented. The generation of Java classes, and the generation of web interfaces.

Automatic code generation

To automatically generate Java classes to wrap the basic device functionality, the XML document is processed. As the document already contains an object oriented view on a device, it is fairly easy to generate a basic Java interface, i.e. the class along with the appropriate methods.
The basic device functionality is inherited from some prepared Java classes (base classes) with a general interface for I/O modules. According to the device type, the generated class implements the appropriate interfaces, like for example *DigitalInput-, DigitalOutput-, AnalogInput-,* or *AnalogOutputDevice*.
The CAN resp. CANopen communication details are hidden in the methods themselves. Inside each method a SDO or PDO with the corresponding values is constructed and sent to CAN with the help from the previously mentioned CANopen provider. If a message is expected, the corresponding message is read, processed and returned with the specified return type. The data type mapping between Java and CANopen is done automatically, whereby some complex datatypes, like for example the PDO mapping, are mapped to appropriate Java classes (*PDOMapping*).
Figure 4 shows three generated Java classes (AI11, AO11, DX11). The CANopenProvider encapsulates the CAN interface hardware and hides the protocol

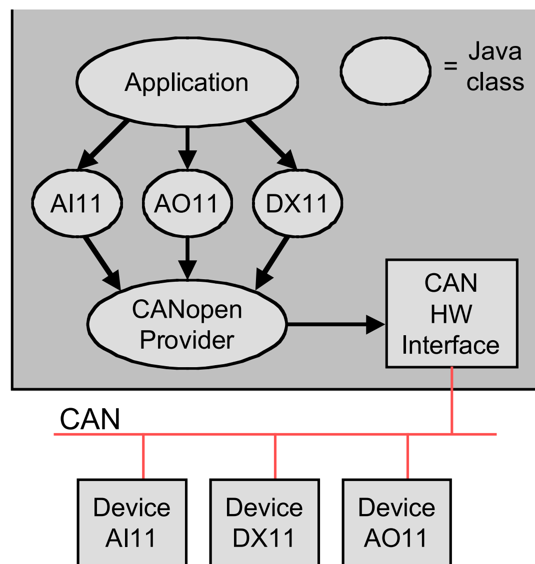

Figure 4: Java classes encapsulating CANopen communication details

specific communication details to the CANopen devices.
The following source code excerpt shows a typical automatically generated method to read the device type. The name DeviceType from the previously shown XML document is extended by *set* resp. *get* according to the given access type.

```
public int getDeviceType()
{
  int retval;
  byte[] data;

  // recv value at index 0x1000, sub 0x0
  data = recv(0x1000, 0);
  // convert resulting byte array to int
  retval = DataConverter.
            ByteArray2int(data));

  return retval;
}
```

As each device extends the base class *Device*, the basic CAN/CANopen functionality is implemented in the base class, which offers some methods like for example *send* and *recv*. The mapping between datatypes is done by a helper class, called *DataConverter*.
Another attractive feature is the generation of automatic callback functionality. As CANopen input devices can be parameterized to send a message when the value of the input lines have changed, the wrapping Java class should also support this feature. Therefore, the code for asynchronous notification is also

generated dynamically. An object which is interested in a device only has to subscribe to this specific device, and gets notified if the state changes.
To use the generated Java class, an application simply creates an instance of the class and uses it like any other Java class.

```
public class App
  implements DigitalInputListener
{
  DX11 dx11;

  public App()
  {
    // create instance
    dx11 = new Dx11();
    // get devicetype
    dx11.getDeviceType();
    // subscribe for events
    dx11.addDigitalInputListener(this);
  }

  // callback method
  public void inputChanged() {
    ...
  }
}
```

Besides the source code files, the system generates some files for ant [11] to automatically compile the sources and to automatically generate the Java documentation.
Once, a Java class is generated, it can be reused in a variety of ways. It can easily be used by other local Java classes or used remotely for the purpose of **teleservice** resp. **telecontrol** [4], by integrating the classes into a **common middleware architecture** like e.g. RMI [14], JINI [15], or CORBA [16]. Like presented in [2], this can also be done automatically by introspection of the generated Java classes, XML representation of these classes and **automatic generation of middleware components**. This procedure can also be applied to **Windows COM** [12] **applications**, as these components support some kind of introspection. Therefore it is possible to combine the generated Java classes for a device with any COM object. Therefore the devices can easily be interfaced with **standard Windows applications**, like for example Microsoft Excel.

As the XML representations are **programming language independent**, the generation of source code in other programming languages is possible too.

Automatic web interface generation

A more static view on the devices can be produced by an eXtensible Style Sheet Language Transformation (XSLT). As shown in figure 5 an XSLT processor [10] is integrated into a web server and delivers dynamically generated HTML pages.
An XSLT processor takes an eXtensible style sheet language document (XSL) to extract information out of a given XML document and produces some output like for example an HTML page, shown in figure 6.
A table oriented view on the devices shows the methods, corresponding return values, parameter types and names, and a description. The underlying XML document is the same as before. In general, different views can be generated on the basis of a single XML document describing the interface of CANopen devices.

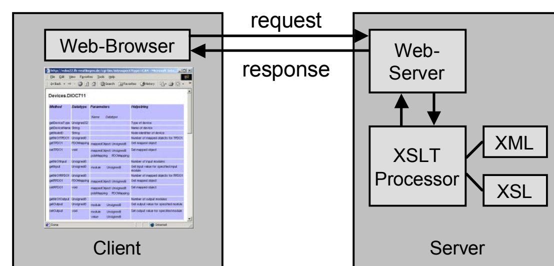

Figure 5: Web server and XSLT processor

Future work

In the future the current system will be extended to support graphical plug and play functionality. The goal is to visualize the interface and connect different XML documents, resp. objects and to generate all the necessary code for interaction automatically. This would result in a

http://robo22.fh-reutlingen.de/cgi-bin/introspect?type=CAN - Microsoft Inte...

Devices.DIOC711

Method	Datatype	Parameters		Helpstring
		Name	Datatype	
getDeviceType	Unsigned32			Type of device
getDeviceName	String			Name of device
getNodeID	String			Node identifier of device
getNrOfTPDO1	Unsigned8			Number of mapped objects for TPDO1
getTPDO1	PDOMapping	mappedObject	Unsigned8	Get mapped object
setTPDO1	void	mappedObject pdoMapping	Unsigned8 PDOMapping	Set mapped object
getNrOfInput	Unsigned8			Number of input modules
getInput	Unsigned8	module	Unsigned8	Get input value for specified input module
getNrOfRPDO1	Unsigned8			Number of mapped objects for RPDO1
getRPDO1	PDOMapping	mappedObject	Unsigned8	Get mapped object
setRPDO1	void	mappedObject pdoMapping	Unsigned8 PDOMapping	Set mapped object
getNrOfOutput	Unsigned8			Number of output modules
getOutput	Unsigned8	module	Unsigned8	Get output value for specified module
setOutput	void	module value	Unsigned8 Unsigned8	Set output value for specified module

Done Internet

Figure 6: Dynamically generated HTML-page for a digital I/O CANopen device visualized in a standard browser.

system, which can be used to build complex applications with a visual builder tool very quickly.
Regarding reality-driven visualization based on Java, Java3D and XML [5], the XML documents for visualization could be combined with XML document describing the functionality of devices.
As Java source code is generated automatically it is also possible to automatically generate components for JINI, RMI and CORBA. The concept of the automatic generation of middleware components regarding standard Java and COM objects is presented in [2].
The integration into a mobile agent system [8], will be part of the future work. Mobile agents have the possibility to discover devices and can talk to devices with XML as an intermediate layer.
The embedded system TINI [13] is a low cost system supporting a Java Virtual Machine and CAN. A CANopen implementation for this system is currently under development. Therefore, it will be possible to use the generated Java control software components in an embedded environment.

If XML device representations are made persistant and stored in a database, it is possible to apply structural query and retrieval capabilities, for example to search for all available devices of a certain type (e.g. Drives, I/O modules, etc.).

Summary

Dynamic discovery of devices, XML based representation of interfaces and automatic component generation allows a nearly transparent integration of devices into a general management and service infrastructure. As high level interfaces are generated, the integration into a higher level language supports the use of modern software concepts and architectures. The XML based representations are easy to process and suitable to generate different target components.

Acknowledgement

This work is partially based upon work within the research consortium VVL funded by the Ministerium für Wissenschaft, Forschung und Kunst of the state of Baden-Württemberg through the research initiative Virtuelle Hochschule.

[1] R.-D. Schimkat, G. Nusser and D. Bühler: “Scalability and Interoperability in Service-Centric Architecture for the Web”, Proc. of the Network based Information Systems (NBIS 2000), Greenwhich (GB), Nov. 2000.

[2] G. Nusser and R.-D. Schimkat: “Rapid Application Development of Middleware Components by Using XML”, Proc. of the 13th Intl. Workshop on Rapid System Prototyping (RSP 2001), Monterey, CA (USA), Jun. 2001.

[3] G. Nusser and G. Gruhler: “Dynamic Device Management and Access based on Jini and CAN”, Proc. of the 7th Intl. CAN Conf. (ICC 2000), Amsterdam (Netherlands), Sep. 2000.

[4] G. Nusser and G. Gruhler: “Teleservice of CAN systems via the Internet”, Proc. of the 6th Intl. CAN Conference (ICC ‘99), Torino (Italy), Nov. 1999.
[5] G. Nusser, D. Bühler, G. Gruhler, and W. Küchlin: “Reality-driven Visualization of Automation Systems via the Internet based on Java and XML”, Proc. of the 1st Intl. Conf. on Telematics Applications (TA 2001), Weingarten (Germany), Jul. 2001.
[6] CAN in Automation (CiA), DS-401 V2.0, “CANopen Device Profile for Generic I/O modules”, CiA draft standard 401, Dec. 1999, CiA.
[7] Object Management Group (OMG), http://www.omg.org/. OMG XML Metadata Interchange (XMI) Specification
[8] R.-D. Schimkat, M. Friedrich, W. Küchlin, “Deploying distributed state information in mobile agents systems”, Proc. of the 9th Intl. Conference on Cooperative Information Systems (CoopIS 2001), volume 2172, pages 80-94, Trento, Italy, September 2001. Springer LCNS.
[9] Siemens AG, SICOMP homepage, http://www1.ad.siemens.de/sicomp
[10] M. Kay, SAXON homepage, http://users.iclway.co.uk/mhkay/saxon/
[11] The Apache Jakarta Project Ant, http://jakarta.apache.org/ant/
[12] Microsoft Corp., http://www.microsoft-.com/com. Homepage for Microsoft COM: COM, DCOM, COM+ technologies.
[13] TINI homepage, http://www.ibutton-.com/TINI/
[14] Sun Microsystems, http://java.sun-.com/rmi/. Java Remote Method Invocation (RMI).
[15] Sun Microsystems, http://www.sun-.com/jini/. JINI Network Technology.
[16] Object Management Group (OMG), http://www.omg.org/. The Common Object Request Broker: Architecture and Specification, Aug. 1997, Rev. 2.1.
[17] World Wide Web Consortium (W3C), http://www.w3.org/TR/REC-xml. Extensible Markup Language (XML) 1.0.

Dipl.-Inform. Gerd Nusser
University of Applied Sciences Reutlingen
Institute for Applied Research (IFA)
Alteburgstr. 150
72762 Reutlingen
Phone: +49 7121 271-625
Fax: +49 7121 25713
gerd.nusser@fh-reutlingen.de

Prof. Dr-Ing. Gerhard Gruhler
Steinbeis Transfer-Center Automation (STA)
Alteburgstr. 150
72762 Reutlingen
Phone: +49 7121 271-331
Fax: +49 7121 25713
gerhard.gruhler@fh-reutlingen.de
http://www.fh-reutlingen.de/~www-sta

Dipl.-Inform. Gerd Nusser,
Prof. Dr. sc. techn. Wolfgang Küchlin
University of Tübingen
Wilhelm-Schickard-Institute (WSI)
Sand 13
72076 Tübingen
Phone: +49 7071 29-77047
Fax: +49 7071 29-5060
nusser@informatik.uni-tuebingen.de
kuechlin@informatik.uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de