Using Automatic Code Generation Methods for Reusable Software Component Development: Experience Report

Elif Kamer Karataş, Barış İyidir
Defense System Technologies Division
Aselsan
Ankara, Turkey
{ekkaratas, biyidir}@aselsan.com.tr

***Abstract*— Quality of reused components becomes one of the dominating factors on the overall quality of the software when the component-based approach is adopted for development. In cases where reusable components are developed to be compatible with reference architecture, the contracts of the components are predefined. Nevertheless, the detailed design and implementation of the component depends mostly on the experience of the developers. The quality and the productivity of component development process can be improved by systematic sharing of domain knowledge and experiences. In this paper, automatic code generation is adopted in order to achieve systematic distribution of this knowledge throughout developers. Also, the experiences gained during the application of automatic code generation approach for the development of components that communicate via serial channel protocols are shared.**

Keywords-code generation; domain specific languages; domain knowledge

I. INTRODUCTION

The proposed automation method is aimed to be used in Embedded Real-time Control Software (ERCS), which is mission critical software that collects data from its sensor environment and processes them with control algorithms to give the proper commands to its actuation environment. The quality of software in such systems is of great importance since the cost of any failure is very high.

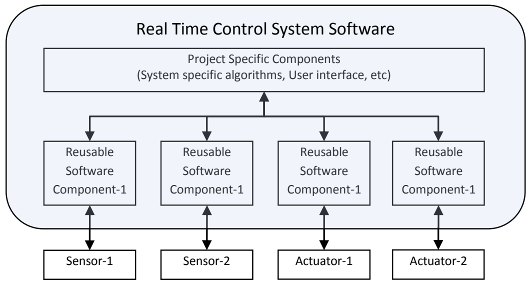

Real Time Control System Software
Project Specific Components
(System specific algorithms, User interface, etc)
Reusable Software Component-1
Reusable Software Component-1
Reusable Software Component-1
Reusable Software Component-1
Sensor-1
Sensor-2
Actuator-1
Actuator-2

Figure 1. Real-Time Control System & Software Architecture.

The real-time control system architecture and the corresponding layered software architecture are given in Figure 1. The reusable software components, which are the main candidates for automation in this study, are responsible for the communication with the surrounding sensors and actuators to receive data and to give commands. These components are developed according to Interface Control Documents (ICDs) delivered together with the sensor or actuator hardware.

In ERCS software, the analysis of defects showed that the average ratio of errors originating from reusable components is 23.48%, which is the primary motivation of quality improvement studies on these components. Difficulties in sharing domain rules and experiences with developers and also the difficulties in proving the conformance to such rules are regarded as significant obstacles on the way to improve the quality and productivity.

Generative programming is defined as a class of tool technology that captures knowledge about how to generate code by enabling automation [1]. Generators are usually based on domain specific notations and they close the gap between high-level system description and executable [2]. Since auto generated codes enforce domain rules and best practices, they provide an effective way for uniformly sharing of domain knowledge among development teams. This study aims the systematic distribution of domain knowledge and expertise using automatic code generation methods.

The paper is organized as follows: Section 2 gives a brief literature review. The proposed method for automatic code generation is described in Section 3. Case studies and the results obtained are given in section 4. Section 5 discusses the results and proposes future works. Starting from section 3 of this text the term “component” will refer to mention reusable components in ERCS.

II. LITERATURE

Software development began with the employment of low level binary machine language and went along with the introduction of assembly language, high-level languages, modeling languages and Domain Specific Languages (DSLs) for providing increased abstraction for developers while performing the generation of source-code automatically [3]. Although code generation without modifications by developer seems impossible, developers are exempted from writing large amounts of repetitive or trivial code sections and have more time to focus on their core engineering problems [3][4].

The advantages of automatic code generation mentioned by Cullum [5] are:

- Code generation enables enhanced consistency since it serves as a repository of reuse. Also, each application developed by automatic code generation will have the same structure independent of the developer.
- Quality will be improved since the amount of manually written code –which is a source of quality variations-, is decreased.
- Productivity is increased since code generators can produce thousand lines of code very fast and these codes are correct by construction.

III. PROPOSED METHOD

For the systematic distribution of domain knowledge and best practices throughout the component development process, we propose automatic code generation as a plug-in to the model-based IDE (i.e., Rhapsody [6]) used for software component development in our projects. The reusable components given in Figure 1 are targeted for automation since the quality of these components affects all the projects they have been used.

The proposed automation process has two main phases, as given in Figure 2, namely, (i) ICD transcription, which is the process of transforming natural language message definitions to machine readable XML (Extensible Markup Language) format, and (ii) model-based code generation.

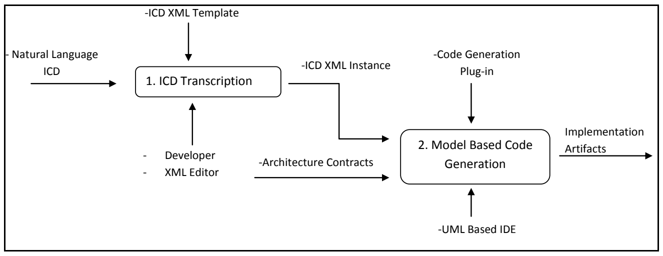


Figure 2. Proposed Automatic Code Generation Method.

One major obstacle on the automatic implementation of component-device communication protocol defined in ICDs is that the ICDs are prepared in natural language, and requires human interpretation. In order to accomplish automatic code generation, representation of the device ICDs in a machine readable format is mandatory. Another important problem is the variability of message structure defined in device ICDs. In the scope of this study, commonality & variability analysis is performed and device ICDs are modeled as an XML template, as given in Figure 3. The XML template given in Figure 3 will be described in detail in the following subsections.

A natural language ICD is the input to the ICD transcription phase where the proposed XML template given in Figure 3 is used as a guideline. The transcription activities are performed manually by the developer with the help of an XML editor. The output of this phase is an ICD XML instance, which is conformant with our domain model and includes the information content of a natural language document (such as communication parameters, message definitions, etc.) in a machine readable format.

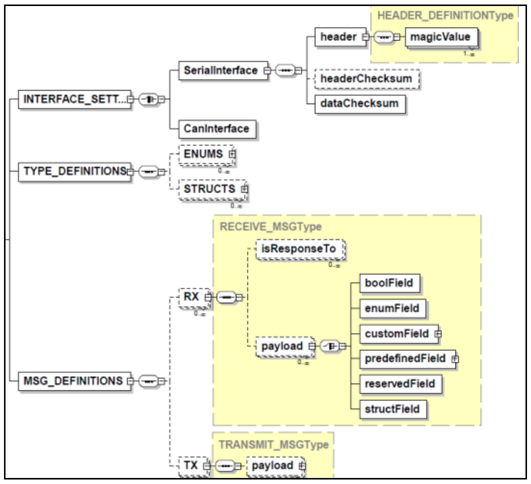
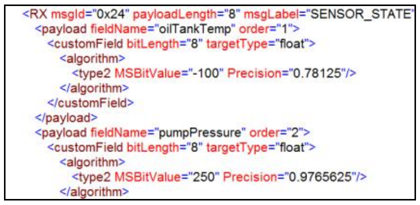


Figure 3. Proposed ICD XML Template.

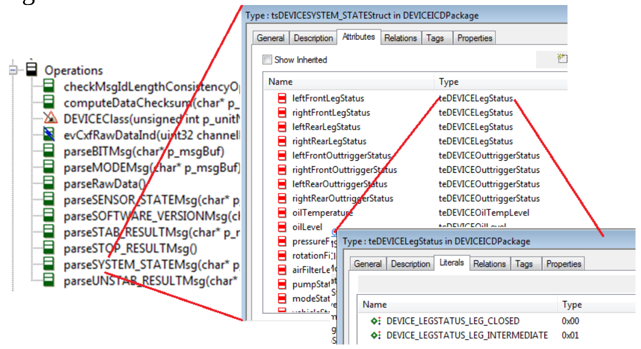
A cross section of a sample XML instance, that models a “Sensor State” message with the message identifier “0x24” and contains payload fields for oil tank temperature and pump pressure is given in Figure 4. The oil tank temperature is expressed with 8 bit data which is mapped to a float variable in code generation phase. The mapping algorithm is described with the value of most significant bit (i.e. -100) and the precision value (i.e. 0.78125).



```xml
<RX msgId="0x24" payloadLength="8" msgLabel="SENSOR_STATE"
  <payload fieldName="oilTankTemp" order="1">
    <customField bitLength="8" targetType="float">
      <algorithm>
        <type2 MSBitValue="-100" Precision="0.78125"/>
      </algorithm>
    </customField>
  </payload>
  <payload fieldName="pumpPressure" order="2">
    <customField bitLength="8" targetType="float">
      <algorithm>
        <type2 MSBitValue="250" Precision="0.9765625"/>
      </algorithm>
```

Figure 4. A section from XML instance (Sensor State Message).

XML instance is the input to the code generation phase together with the architecture-based component contracts. Automatic code generation is performed by invoking the code generation rules embedded into a tool that is developed as a plug-in to our current development environment. At the end of code generation, the outputs are various implementation artifacts such as attributes, operations, events, type definitions, state chart diagram elements, etc. Some of the artifacts after code generation phase are given in Figure 5.



Operations
- checkMsgIdLengthConsistencyO
- computeDataChecksum(char* p_
- DEVICEClass(unsigned int p_unitN
- evCxfRawDataInd(uint32 channel
- parseBITMsg(char* p_msgBuf)
- parseMODEMsg(char* p_msgBuf)
- parseRawData()
- parseSENSOR_STATEMsg(char* p
- parseSOFTWARE_VERSIONMsg(c
- parseSTAB_RESULTMsg(char* p_r
- parseSTOP_RESULTMsg()
- parseSYSTEM_STATEMsg(char* p
- parseUNSTAB_RESULTMsg(char*

Type : tsDEVICESYSTEM_STATEStruct in DEVICEICDPackage

General | Description | Attributes | Relations | Tags | Properties

Show Inherited

Name	Type
leftFrontLegStatus	teDEVICELegStatus
rightFrontLegStatus	teDEVICELegStatus
leftRearLegStatus	teDEVICELegStatus
rightRearLegStatus	teDEVICELegStatus
leftFrontOuttriggerStatus	teDEVICEOuttriggerStatus
rightFrontOuttriggerStatus	teDEVICEOuttriggerStatus
leftRearOuttriggerStatus	teDEVICEOuttriggerStatus
rightRearOuttriggerStatus	teDEVICEOuttriggerStatus
oilTemperature	teDEVICEOilTempLevel
oilLevel	
pressureF	
rotationFi	
airFilterLe	
pumpStat	
modeStat	

Type : teDEVICELegStatus in DEVICEICDPackage

General | Description | Literals | Relations | Tags | Properties

Name	Type
DEVICE_LEGSTATUS_LEG_CLOSED	0x00
DEVICE_LEGSTATUS_LEG_INTERMEDIATE	0x01

Figure 5. Snapshot of auto-generated model elements.

The detailed information on the construction of the XML template and the code generation activities are given in the following subsections.

A. ICD XML Template Structure

According to the model given in Figure 3, not every interface definition has a header checksum part, and in case of its existence the size and algorithm for its computation is variable. Also, data bytes in the payload part of messages can be converted to float or double values with some processing.

Although the structure of the messages defined in ICDs does not display variation depending on the direction of messages (transmit and receive directions), due to their semantics and behavioral differences our model distinguishes transmit and receive messages. From the components point of view, in addition to the content of a received message the information of to which transmit message it is a response to, is also important for the behavior.

Another variation point in message contents is how to decode the values encoded within byte sequences. Decoding methods extracted from the ICDs can be grouped as;

- Bit field definitions
- Float/double value transformation from discrete byte sequences with a given resolution
- Direct casting of byte sequences to short, integer, float, double, etc. values.

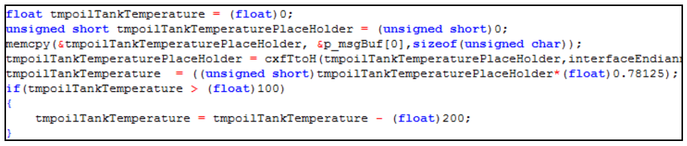
B. Implementation of Component Device Communication Protocol

For each component, there is a device (sensor, actuator, etc.) that it communicates over physical channels (serial, CAN, Ethernet, etc.) with conformance to an ICD. Since message parsing functionality is common for components in this study, this functionality is seen as the most suitable candidate for automation. Also, automatic implementation of enlisted messages, enumerated values, and numerical constants in the ICD are in the scope of this study.

TABLE I. XML TO IMPLEMENTATION MAPPING RULES

XML Element	Implementation Artifacts
Interface Settings	Serial port parameters (baud rate, parity, etc) and structure of messages (header, id, size, crc, etc.)
Type definitions and message payload	Enumerations and structures
Receive message names	Message specific parser function declarations
Receive message payload	Message specific parser function implementation
Decoding algorithm type	Converting byte sequences to target language types(float, int, etc.)
Checksum algorithm and data length	Checksum function implemention

Basic transformation rules can be applied after translation of ICD into XML file. The mapping rules from XML elements to implementation artifacts are described in Table I.



```
float tmpoilTankTemperature = (float)0;
unsigned short tmpoilTankTemperaturePlaceHolder = (unsigned short)0;
memcpy(&tmpoilTankTemperaturePlaceHolder, &p_msgBuf[0],sizeof(unsigned char));
tmpoilTankTemperaturePlaceHolder = cxfTtoH(tmpoilTankTemperaturePlaceHolder,interfaceEndian
tmpoilTankTemperature  = ((unsigned short)tmpoilTankTemperaturePlaceHolder*(float)0.78125);
if(tmpoilTankTemperature > (float)100)
{
    tmpoilTankTemperature = tmpoilTankTemperature - (float)200;
}
```

Figure 6. Code generated from XML instance.

Using the transformation rules and sample XML instance given in figure 4 automatic code generation is done. The automatically generated code for parsing oil temperature is given in Figure 6.

C. Implementation of Component Interfaces

Apart from the physical interface with devices, reusable software components also have contracts with the internal project specific components. While realizing these contracts different design alternatives can be adopted and different assumptions can be made by the developers. Also, it is difficult to prevent and diagnose the cases where different components have conflicting design decisions. In the scope of this study, critical interfaces and their expected design decisions are identified to provide a common behavior through the contracts. Our intention is to embed this common behavior into the component automatically by state chart design and reaction implementation.

TABLE II. CONTRACT TO IMPLEMENTATION MAPPING RULES

Contract Element	Implementation Artifacts
Component mode information	States
Component mode change indication	state transitions
Component activation request	state reception and response to activation request
Component setting request	State reception and response to component setting request
Component deactivation request	State reception and response to component deactivation request

Basic transformation rules from contract elements to implementation artifacts are given in Table II. Also a sample auto generated statechart implementation with the given rules is shown in Figure 7. The main states of the component, transitions between common states and common reactions are auto generated.

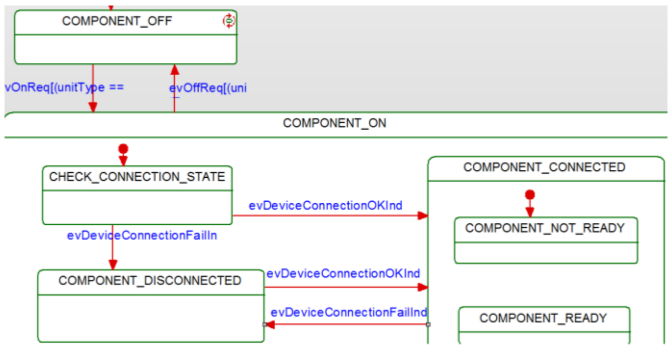


Figure 7. Example statechart implementation.

IV. CASE STUDY

In order to evaluate the effectiveness of the proposed method, a group of software components which are previously implemented with traditional methods are re-generated by the proposed method. Since these traditionally developed software components are already being used in the system, these components are seen as golden units. In order to provide the same functionality with golden units, automatically generated software components need to have approximately the same number of source lines. This assumption is based on the fact that automatic code generation process aims to generate the same code as traditional development. Under the given assumption, the effectiveness of proposed can be measured by comparing the source code line counts of automatically generated component and traditionally developed components.

TABLE III. CASE STUDY METRICS

Metric Name	SC1	SC2	SC3
Number of messages received	5	8	13
Number of data typed needed	4	12	21
Number of data fields transmitted	34	27	96
Total code line count for traditionally developed component	3524	3663	6378
Total code line count with proposed approach	1339	1357	2071
Auto-generated code rate(%)	37.9	37.04	32.42

The proposed automation method is applied to 3 different software components (SC1, SC2, and SC3), which are already available in our component repository. The source line count measurements related to traditionally and automatically developed components are given in Table III. In order to give information about the size and complexity of the interfaces, the number of messages in the receive direction, the number of distinct data types required to implement the content of the receive messages, and the number of data fields carried within receive messages are also shown in Table III.

The results indicate that proposed method is applicable to devices that have different interface complexities, since the automatically generated code ratio remains approximately the same for different components with different interfaces.

V. CONCLUSION AND FUTURE WORK

Based on our case studies, we can state that with the current scope of our proposed method, it is possible to achieve over 32% automatic code generation. It is estimated that this rate can be increased up to 50% with the addition of potential functionalities and behaviors that are scoped out in the first phase of our study. Considering that the automated code section handles most of the low level parsing operations and establishes a basis for the infrastructure of the components, we assess 30% as an effective automation rate for our domain. Another advantage of the automation is that it removed some mechanical actions during manual development, such as several type and function definitions and implementing predetermined reactions to requests in known states.

By enabling automatic code generation in one of the most error prone sections of component development, namely the “parser codes”, we estimate that quality costs will be decreased in the long run and unit integration process can be completed more efficiently.

In addition to its direct effects to the component development, XML template based approach establishes a guideline for developers while inspecting the ICDs provided to them since it makes explicit the information content required for the accurate implementation of a component-unit interface. Although in scope of this study, the transformation of ICDs written in natural language to XML format is performed manually in order to increase the efficiency and usability of the proposed approach we plan to develop a wizard to guide the user during the ICD XML instance creation process. In the long run, we hope that software developers will not need to transcript ICD XML from the natural language document, but instead unit vendors will design their communication protocol on this wizard, thus its output will be ready to use by the code generation tools.

The current scope of the proposed method includes the message parsers, common states, transitions between common states, default reactions in the common states and the required attributes, types, and events to implement them. In the later phases of the study, the code generation capability will be extended to include the message sending functions and the realization of unit type specific interfaces.

REFERENCES

[1] R. Slaghi, and A. Strohmeier, “Better Generative Programming with Generic Aspects,” Technical Report, Software Engineering Laboratory, Swiss Federal Institue of Technology, Switzerland, 2003.

[2] K. Czarnecki, “Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models,” Ph.D. dissertation, Department of Computer Science and Automation, Technical University of Ilmenau, Germany, Oct. 1998.

[3] D. P. Gluch, A. J. Kornecki, and I. N. Sneddon, “Automated Code Generation for Safety-Related Applications: A Case Study,” Proc. International Multiconference on Computer Science and Information Technology, pp. 383–391, 2006.

[4] K. Fertalj and M. Brcic, “A Source Code Generator Based on UML Specification”, International Journal of Computers and Communications, Issue 1, vol. 2, 2008.

[5] S. Cullum, “The Effect of Automatic Code Generation on Developer Job Satisfaction,” Technical Report No:2007/19, Open University, U.K., Sept. 2007

[6] E. Gery, D. Harel, and E. Palachi, “A Complete Life-Cycle Model-Based Development System,” In Proceedings of the Third International Conference on Integrated Formal Methods, 2002, pp. 1-10.