ANIMATION OF BEHAVIORAL SPECIFICATIONS THROUGH CODE GENERATION FOR A PAYMENT SYSTEM

Ozan Deniz
Payment Systems Department
Central Bank of the Republic of Turkey
Ankara, Turkey
ozan.deniz@tcmb.gov.tr

Mehmet Adak
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
bmadak@gmail.com

Halit Oğuztüzün
Department of Computer Engineering
Middle East Technical University
Ankara, Turkey
oguztuzn@ceng.metu.edu.tr

***Abstract*—We present a case study concerned with the animation of behavioral specifications through code generation for a payment system; namely, electronic funds transfer system (EFT). The exchange of messages between a central bank and two client banks during daily operations is modeled as a communications model of Live Sequence Charts (LSCs). Using an LSC to Java/AspectJ code generator, the communications model is converted to a base code and then the animation code is woven into this base code. Execution of the resulting code animates the messages exchanged among the central bank's EFT server, central bank's branch and two client banks' EFT servers for sample money transfer operations as a sequence of events respecting the partial order specified by the LSC. The woven aspect code also addresses two additional issues: One is domain specific processing such as queue operations and settlement operations at the central banks' EFT server, and the other is scenario processing for money transfers.**

Behavioral Specifications, Live Sequence Charts, Validation, Code Generation, Payment Systems.

I. INTRODUCTION

Behavioral specifications (or descriptions) represent the order of the communications among system components. For domain experts and users, systems' animations are expected to be more comprehensible than a formal notation. Animation of the behavioral specifications plays an important role in early validation of the systems and can be a useful method for their realization. In this study, the approach of generating the code of the animation by the help of a code generator is presented with a sample application in the payment systems domain; namely, electronic funds transfer system (EFT).

The exchange of messages between a central bank and two client banks during daily operations is modeled as a communications model of Live Sequence Charts (LSCs). Using an LSC to Java/AspectJ code generator, [2] the communications model is converted to a base code. Live Sequence Charts is a graphical language introduced by David Harel and his colleagues [1] for specifying interactions between components in a concurrent system. The input to the code generator is a LSC in the abstract syntax. The abstract syntax is given by a metamodel for LSCs [3].

The LSC metamodel is an instance of metaGME, the metalanguage offered by Generic Modeling Environment (GME), [4] a metamodeling tool developed at Vanderbilt University, USA. The code generator is implemented in Java as a "model interpreter" in GME. The approach to application generation is aspect-oriented in that the generated LSC code serves as the base code for the complete application code. The developer has to weave the application's computation aspect into the LSC base code, in which the cut points include the function calls corresponding with the events. The code generator also provides a default computation (in AspectJ) that exercises the input LSC by picking a random trace.

The EFT application illustrates a real time settlement system, which transfers funds from one bank to another. For example, you may want to use your deposit at *BankA* to send a payment to a person having an account at *BankB*. Upon your order, *BankA* sends a payment message, which includes necessary details like the sender, the beneficiary, value etc. to EFT server. EFT server which resides at central bank promptly processes the incoming payment message. If the balance of the account of *BankA* at EFT server is sufficient, the payment is transferred from *BankA* to *BankB* within a few seconds. Otherwise, the payment is queued.

The paper is organized as follows: In Section II we give brief information about LSCs and present behavioral models of the EFT system as LSC diagrams. The methodology used during the metamodeling process and LSC metamodel structure are defined in Section III. In Section IV the process of code generation from LSC models are defined and generated code samples are presented. Finally, Section V contains concluding remarks.

II. LSC MODELS

A. LSCs and Behavioral Model of EFT System

LSCs constitute an attractive visual formalism that is widely used to capture requirements during the early design stages of the system development. It is an extension of Message Sequence Charts (MSC), which are widely used in the specification of telecommunication systems [6] and are closely

978-0-7695-3783-2/09 $26.00 © 2009 IEEE
DOI 10.1109/BCI.2009.39

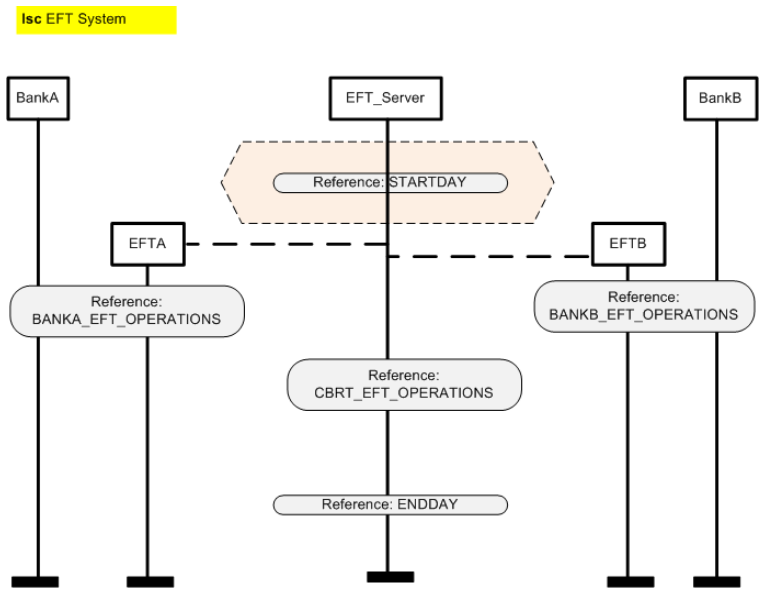

Figure 1. Behavior Model for the Electronic Funds Transfer System

related to the Sequence Diagrams of UML. But MSCs do not provide means to distinguish mandatory and possible behavior. The Live Sequence Chart (LSC) language introduces the distinction between mandatory and possible on the level of the whole chart and for the elements messages, locations, and conditions. Furthermore they provide means to specify the desired activation time by an activation condition or by a whole communication sequence, called pre-chart.

The behavior model of the electronic funds transfer system is presented in LSC graphical form (Figure 1). This figure denotes a high level LSC that consists of reference LSCs. The instances are (represented as rectangles): Central bank's EFT server (EFT_Server), EFT server's bank instances (EFTA and EFTB) and two client banks (*BankA* and *BankB*). The vertical lines represent lifelines for the instances. A typical LSC includes mainly two charts: a pre-chart (the diamond-shaped area on top) and a main chart (the rest of the chart). The pre-chart behaves like a conditional. If it is satisfied, the main chart is executed. In the pre-chart of the diagram, initial crediting is done by transferring banks' balance from central bank's branch to the central bank's EFT server. The behavior of this operation is defined in another LSC diagram, namely, "STARTDAY". If the EFT system successfully completes pre-chart, then the diagram proceeds with the reference structures "BANKA_EFT_OPERATIONS", "CBRT_EFT_OPERATIONS", and "BANKB_EFT_OPERATIONS" which all represent daily EFT operations. The system execution completes with the "ENDDAY" LSC reference which transfers banks' end day balances from EFT server to central bank's branch.

B. *Client Banks' EFT Operations*

The behavior model for a client bank's (*BankA*) EFT operations is presented in Figure 2. There are two instances: *BankA* and EFTA. This structure includes two operands run in parallel: the "Pay&Inquiry" thread and the "GetPaid" thread. A condition (ExitCondition) synchronizes the exit for these threads. In the "Pay& Inquiry" thread part, *BankA* instance can send three types of messages to the EFTA instance. The message "PayMSG" is sent whenever an electronic funds transfer request is ready at *BankA* instance. On receiving the message, EFTA instance calls the action "Settlement" which transfers payment if the balance of *BankA* is sufficient. Otherwise the payment is queued with the priority obtained from message context. The settlement action status (transferred or queued) is sent to *BankA* instance with the message "SettlementStatus". *BankA* can query the queued payments by sending "InquiryQueue" message to the EFTA instance. On receiving this message EFTA forms a queue list and sends this list to *BankA* instance. The last type of the messages in this thread is "ChangePriority" which is sent to change the priority of the queued payments at EFTA instance. EFTA instance changes the priority of the payment's priority in the action "ChangePriority" and informs *BankA* instance by the message "PriorityInfo".

In the "GetPaid" thread part *BankA* instance is informed with the settled payment messages which other client banks sent to the EFT_Server to transfer funds to *BankA*. *BankA* instance sends the message "InqSettledMsgs" to the EFTA. On receiving the message EFTA checks whether a settled payment messages exists for *BankA*. If there exist one EFTA sends a "PayMSG" to the *BankA* instance, so that *BankA* can update its local balance.

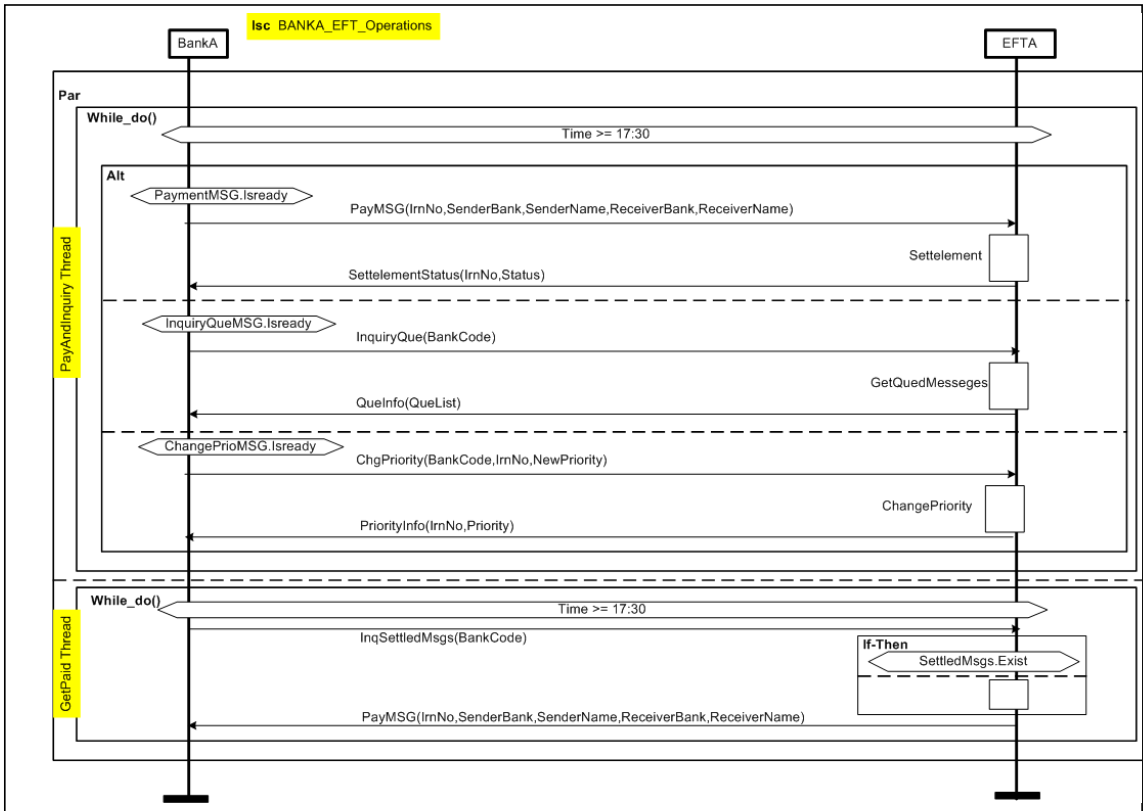

Figure 2. Behavior Model for the BankA's EFT Opeartions

```
public static void Bank_AMainMethod() {
        class InlineOperand_0035 extends Thread {
            InlineOperand_0035() {}
            public void run() {
                while (((Boolean)MSC.coldChoices.get("while_0060")).booleanValue()) {
                    int Alt_003c = -1;
                    if (MSC.altChoices==null ||MSC.altChoices.get("Alt_003c")==null ||
((Integer)MSC.altChoices.get("Alt_003c")).intValue() == -1)
                                Alt_003c=chooseAlt(3,"Alt_003c");
                    else
                        Alt_003c= ((Integer)MSC.altChoices.get("Alt_003c")).intValue();
                    switch (Alt_003c) {
                    case 0:
                                                                    …
                    case 1:
                        SendMessageMsg_InqQueMSG_004cEFT_A(new Object());
                        condRecvMessageMsg_QueInfo_0049EFT_A();
                        break;
                    case 2:
                         …
                    }
                    try {
                      Thread.sleep(2000);
                    } catch (InterruptedException e1) {
                      // TODO Auto-generated catch block
                                            e1.printStackTrace();
                }
                    try { MSC.WhileDo_0038.await();

                    } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                stop();
            }
        }
```

Figure 3. Part of the generated Java code for BankA's EFT Operations

```
public static boolean SendMessageMsg_InqQueMSG_004cEFT_A(Object BankCode_0084) {
        LSCLib.LSCObject proc= new LSCLib.LSCObject();
        proc.name="Msg_InqQueMSG_004c";
        proc.pars=new ArrayList();
        LSCLib.LSCAttribute parNew0 =new LSCLib.LSCAttribute();
        parNew0.name="BankCode_0084";
        parNew0.type="Object";
        parNew0.objClass="Object";
        parNew0.objVal=BankCode_0084;
        proc.pars.add(parNew0);
        EFT_A.RecvMessageMsg_InqQueMSG_004cBank_A(proc);
        return true;
    }
```

Figure 4. Send Message (Generated Java Code)

```
boolean around(Object BankCode_0084):pcSendMessageMsg_InqQueMSG_004cEFT_A(BankCode_0084) {
        BankCode_0084=new Object();
        proceed(BankCode_0084);
        return true;
    }
```

Figure 5. Send Message (Generated AspectJ Code)

III. METAMODELLING PARADIGM AND LSC METAMODEL

GME is a generic, configurable modeling environment. In other words, the configuration of GME is not an option; it is the first step that must be taken before anything meaningful can be done with it. The configuration process itself is also a form of modeling - the modeling of a modeling process. This is called metamodeling. The output of the metamodeling process is a compiled set of rules, the paradigm that configures GME for a specific application domain.

The users do not need to know much about metamodeling; what they see is a graphical tool with editing capabilities that is already customized for their domain. The main advantage of model-integrated computing is that the work of the users is guided by the modeling environment. The first thing that a metamodeling expert needs is a specification (or at least some vague idea) of the modeling application to be implemented. This typically comes in a natural-language description.

The most basic step in the metamodeling process consists of determining two things: the entities used by the model, and the relations between them. Information used to identify and qualify certain entities and relations will be assigned to them as attributes. Metamodeling, in a nutshell, is the mapping of specification concepts onto entities, relations and attributes.

During the metamodeling process, a suitable GME concept (First Class Object) is chosen to represent each of the real-world concepts that appear in the specification. A set of GME meta-datatypes, or *kinds*, are established. Each defines a set of attributes, composition rules, association options, constraints, etc. The generic GME First Class Object (model, reference, etc.) that is selected for an entity or association kind (such as a router, or network connection) is called its *stereotype*. LSC metamodel defines the basic LSC concepts such as events, instances, messages, parallel, alternative and the connection between these concepts in meta level. Each of these concepts is represented by GME's First Class Objects. For example actions, instances and messages are represented by model object.

IV. USING THE TEMPLATE

In principal, the code generation strategy is based on the Aspect Oriented Programming approach, which allows us to generate code to exercise LSCs in a computation-free manner. Then the user can weave, using AspectJ, application-specific computational (and other non-communication) aspects into the generated base code.

Eclipse 3.3 is used [8] for the development environment into which AspectJ Development Tools (AJDT 1.5.3) extension [9] is installed. Firstly, a new AspectJ project was created and generated base code classes were extracted into it. And then the animation aspect code was weaved into the base code. Finally the code was executed.

For each entities described in the behavior model, code generator generates one Java class and one AspectJ class. And for each animation, one base class is generated. In the base class, one thread is defined for each entity and these threads run in parallel. The messages exchanged between the entities guarantee synchronization of the threads.

An example of the generated Java code for *BankA*'s EFT operations is presented in Figure 3. For the sake of brevity only the first thread's content is included; but by looking at Figure 2 and Figure 3, one can see how some of the LSC model elements (such as parallel, do while, alter, if then else etc.) are converted into Java code by the code generator.

```
public static boolean condRecvMessageMsg_InqQueMSG_004cBank_A() {
        while (!boolMessageMsg_InqQueMSG_004cBank_A())
            SleepThread(100);
        LSCLib.LSCObject proc=null;
        try {
            proc = (LSCLib.LSCObject) queMessageMsg_InqQueMSG_004cBank_A.dequeue();
        } catch (InterruptedException e) {
            e.printStackTrace();
            return  false;
        }
        ProcessRecvMessageMsg_InqQueMSG_004cBank_A(proc);
        return  true;
    }
```

Figure 6. Receive Message (Generated Java Code)

```
void around(Object BankCode_0084):pcRecvMessageMsg_InqQueMSG_004cBank_A(BankCode_0084) {
        System.out.println("Received message:"+BankCode_0084);
        g_BankCode_0084=BankCode_0084;
        proceed(BankCode_0084);
    }
```

Figure 7. Receive Message (Generated AspectJ Code)

```
<BankA> Send Message Msg_PayMSG >>IRN=1111111 Amount=50 Priority=1
SenderBank=BankA SenderName=Ozan ReceiverBank=BankB ReceiverName=Mehmet

<EFTA> Receive Message Msg_PayMSG >>IRN=1111111 Amount=50 Priority=1
SenderBank=BankA SenderName=Ozan ReceiverBank=BankB ReceiverName=Mehmet

<BankB> Receive Message Msg_PayMSG >>IRN=1111111 Amount=50 Priority=1
SenderBank=BankA SenderName=Ozan ReceiverBank=BankB ReceiverName=Mehmet

<BankA> Receive Message Msg_SettlementStatus >>Irn=1111111 Status=SETTLED

<BankA> Send Message Msg_ChgPriority >>Irn=3333333 BankCode=BankA NewPrio=100

<EFTA> Receive Message Msg_ChgPriority >>Irn=3333333 BankCode=BankA NewPrio=100

<BankA> Receive Message Msg_PriorityInfo >> Irn=3333333 Priority=100

<BankB> Send Message Msg_InqQueMSGBankA

<EFTB> Receive Message Msg_InqQueMsg >> BankCodeBankA

<BankB> Receive Message Msg_QueInfo >>2222222&&3333333
```

Figure 8. Animation of LSC Models

Sample Java code, generated for message sending operation is presented in Figure 4. In the code sample the message "SendMessageMsg_InqQueMSG_004cEFT_A" is sent from the instance *BankA* to the instance EFTA. A message object is created and put into the message queue prepared for EFTA. Sending operation is completed successfully when EFTA receives the message.

The code presented in Figure 5 is part of the generated AspectJ code for *BankA* instance. In this sample code the content of the message to be sent is created.

In figure 6, part of the Java code generated for message receiving operation for EFTA instance is presented. In this part of the code, the message "condRecvMessageMsg_InqQueMSG_0151Bank_A" is received from *BankA* instance. The operation is completed

when the message is get from the message queue which was queued in message sending operation (Figure 4).

The AspectJ code presented in Figure 7 is taken from the generated AspectJ code for EFTA instance for message receiving operation. The content of the messaged is printed to the console.

V. CONCLUSION

For the animation of the sample EFT system's specifications through code generation we have completed three main phases: Modeling Phase, code generation phase and the finalization phase.

In the modeling phase we constructed the behavioral model using LSCs, and using the domain specific language provided by LSC metamodel. This effort took one person week.

In the code generation phase, the code generator automatically generated the base code which is basically the application code with preliminary computation aspect weaved. The generated code has 2822 lines of code (LOC). Execution of the generated code animated the messages exchanged among the central bank's EFT server, central bank's branch and two client banks' EFT servers for sample money transfer operations as a sequence of events respecting the partial order specified in the LSC.

The execution of the generated code facilitated the early validation of the behavioral specifications expressed in modeling phase. By observing the traces produced as the generated code runs, we could check if there were any unsafe behaviors such as a payment message request with no reply.

In the finalization phase we wove the application logic as computation aspect. This part includes implementation of getting input data from files, queuing unsettled messages at EFT server, sorting payments by their priorities and updating balances. What is pleasant for this phase is no modifications were necessary for the generated Java code, which proves a successful LSC modeling phase and a consistent code generation. So it was just enough to modify AspectJ codes for weaving the application logic. We also benefited the further use of AspectJ such as logging. The final code has 3572 LOC.

The animation of the application is seen in Figure 8. In the animation, part of the EFT messages for client banks' are displayed which were defined at section 2.2.

Future work will address reviewing the animation of various scenarios with the participation of domain experts and discuss the usability of the animation presented throughout this study.

REFERENCES

[1] Harel D., Marelly R., "Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine", Springer-Verlag, (2003)

[2] Adak M. and Oğuztüzün H., A Code Generator for LSC and MSC. Technical Report (METU-CENG-TR-2007-04), Middle East Technical University, (2007).

[3] Topçu, O., Adak, M. and Oğuztüzün, H., Metamodelling live sequence charts for code generation, Software and Systems Modelling, to appear.

[4] Generic Modeling Environment (GME).
Available:http://www.isis.vanderbilt.edu/projects/gme

[5] Turkish Interbank Clearing and Electronic Funds Transfer System Available: http://eft.tcmb.gov.tr/index.php?newlang=english

[6] Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (2004)

[7] Aspect-Oriented Software Development by Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit

[8] Eclipse, Eclipse.org. Available: http://www.eclipse.org/

[9] AspectJ Project at Eclipse. Available: http://www.eclipse.org/aspectj/