JEstelle
Novel approach to the distributed Java systems specification and development

Marcin Czenko
Warsaw University of Technology
Institute Of Computer Science
Nowowiejska 15/19
00-665 Warsaw, POLAND
M.Czenko@elka.pw.edu.pl

Jean-Luc Raffy
GET/INT
CNRS Samovar
Software-Networks Department
9 rue Charles Fourier
91011 Evry CEDEX, FRANCE
jean-luc.raffy@int-evry.fr

ABSTRACT
The design of distributed Java applications is a very complicated task. Regardless of Java's ease of use and portability, a method that provides a means of testing and validating complex systems in a satisfactory manner is still lacking. In addition system portability is often decreased due to strong dependencies between subsystem implementation and the way they communicate. In this article we would like to introduce JEstelle – the union of Java and the Estelle Formal Description Technique. JEstelle provides a solution that can significantly simplify the distributed Java applications development. JEstelle introduces some level of formalism to the communication part of a distributed Java application, thus allowing for its validation and improving readability of the system design. JEstelle does not introduce serious Java API restrictions and naturally supports automatic implementation code generation. Development of JEstelle support tools is simplified due to a combination of specific features of Java technology and the use of existing Estelle development tools. JEstelle is easy to use and practically does not require the designer to be familiar with Formal Description Techniques at all, and with Estelle in particular.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – d*istributed programming, parallel programming.*

D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages, methodologies, tools.*

C.2.2 [**Computer/Communication Networks**]: Network Protocols – *protocol architecture, protocol verification.*

C.2.4 [**Computer/Communication Networks**]: Distributed Systems – *distributed applications.*

General Terms
Documentation, Reliability, Standardization, Languages, Verification.

Keywords
Distributed Systems, Java, Estelle, Software Engineering, Formal Description Techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2003, 16-18 June, 2003, Kilkenny City, Ireland.
Copyright 2003 ISBN: 0-9544145-1-9 …$5.00.

1. INTRODUCTION
Distributed and concurrent systems constitute a significant fraction of the today's software. It is not only distributed software, used for solving complex computational problems but also a great number of commercial products that incorporate some kind of concurrency to improve their performance. Since Java has been introduced it has been shown to be a great solution to programming problems on the World Wide Web and for distributed software design. This is because of its portability and programmability that both make platform-independent development simpler. Even here, however, problems arise. Problems concerning the development process and reliability of Java distributed systems still need special attention to make the system more efficient. Even though there are well-tested methods for the synchronous systems design, it is not always true that we can use them in distributed applications with satisfactory results. To make distributed and parallel software design simpler and more efficient we thus need a unified approach to concurrent system modelling and implementation. It is especially important for proper communication between a number of subsystems that make up the whole distributed architecture. Using a standard approach to the Java distributed application development process we are unable to easily keep the system independent from the particular communication channel implementation, thus making system consistency harder to maintain. Going further, it is clear that to design distributed systems that are safe and secure in use, we need an efficient method for their validation. The validation is

especially important for the communication part of a system, as this is the point where uncertainty in the design results in unexpected behaviour of the whole system. To make it possible we introduce a level of formalism to the Java distributed systems development process. The formalism should concern only the communication part of the distributed system not touching the rest of the code. Using Unified Modelling Language (UML) terminology we want to distinguish between description of the system activities in a particular state, which should lean towards the implementation language, and the description of the way system states changes, which might be formalised and expressed in the intermediate form before final implementation. Formal Description Techniques (FDTs), which are well known and widely used in the protocols development, can be used to make Java design more formal and provable. Out of all FDT techniques commonly used today, we found Estelle [1] to be the best foundation to the formal extensions to Java language. What makes Estelle so interesting for our approach is that it permits a clear separation of the description of the communication interface between components and the description of the internal behaviour of each such component. Whilst components description shall be done purely with Java, the communication mechanism between them might be expressed with the Estelle framework. As a result we obtain a language, JEstelle, which merges the power of Java programming with the formalism of Estelle.

In section 2 we present in more detail the structure of JEstelle and a simple example. Section 3 presents the JEstelle development cycle and introduces the JEstelle Native Interface (JENI). In section 4 we give an example. We then present our current development and make some conclusions.

2. JESTELLE

Estelle can be viewed as a set of extensions to ISO Pascal [7] level 0 and in the similar manner JEstelle can be regarded as a set of extensions to Java language. JEstelle extensions model a given system as a hierarchical structure of communicating automata. These automata, that can be also regarded as separated components of a distributed system, can run in parallel, and communicate by exchanging messages and by sharing, in a restricted way, some variables. Each active component is an instance of a module (comparable to the class definition in Java) defined within the JEstelle code by a module definition. Thus, it is appropriate to call components module instances, but unless this leads to confusion we will use the term module rather than module instance [1]. The description of a module is composed of a module header declaration and a module body description. A module header can be imagined as an external interface to the module which is defined as a set of input/output access points and exported variables. The module body provides the internal description of a component and can include definitions of child modules and internal interaction points. The behaviour of a module is specified by the set of transitions (of an extended state transition model [1]) that the module may perform. Informally, a module can be represented graphically as a box possibly with points on its boundary (external interaction points) and inside of it (internal interaction points). Several modules instances can then communicate by exchanging messages using communication channels between external interaction points and by exported variables from the parent modules. Estelle used Pascal level 0 language [7] to describe modules behaviour while JEstelle uses Java. Despite the fact that some more complicated Java structures are not allowed in JEstelle source code [6] there is no objection to the use of the Java standard API. We can bypass most of the syntactic restrictions in a JEstelle program by providing external Java libraries in the form of **.class* files (see next paragraph). In the JEstelle module body definition we can point out the exact correspondence between fragments of the JEstelle automaton and the elements of the visual modelling languages like UML (Unified Modelling Language). JEstelle modules correspond well to the UML *State Machine View* as all of them might be treated as classes while active modules as instances of such classes. Figure 1 presents an example *state machine diagram* and figure 2 the

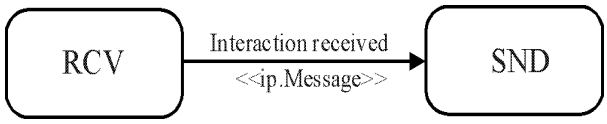

Figure 1. UML state machine diagram.

JEstelle module body transition part fragment that could be equivalent to this UML diagram. A typical behaviour of the JEstelle module is awaiting events that take the form of interactions sent by other modules. When a new interaction arrives we speak of an external transition having occurred changing the object state. With the presented UML diagram all activities performed by the module during its change from state *SND* to *RCV* can be expressed within the JEstelle program with a single call to the appropriate class object method (in the example we call method *Process_request* for the *obj* object) that encapsulate all lower level details of the processing for this transition. The important feature of JEstelle programming that should be noted is that the implementation of the class for the *obj* instance can be external to the JEstelle program. This means that the class for the *obj* instance can be implemented with full-

```
// JEstelle module body transition part
// fragment
  trans
    from RCV
    to SND
      when ip.Message
      name Interaction_received
      {
        obj.Process_request(data) ;
      }
```

Figure 2. JEstelle specification fragment reflecting UML state machine digram from figure 1.

featured Java language.

As JEstelle is a merger of Java and Estelle it would be desirable to take advantage of development support that exists for the both. Even if possible, it is not a good idea to build the execution environment for JEstelle from the ground up as it would take years to make design with JEstelle possible. The best we can do is to adopt the existing Estelle toolset by extending it with the Java

Virtual Machine to execute Java code without semantic lost. To do that the JEstelle program must be convertible to Estelle code. In practice this reduces to the problem of conversion of Java statements to the Pascal level 0 used in Estelle. To support this we developed what we call the JEstelle Native Interface (JENI). Furthermore as the process should transparent to the designer, the conversion will be automatized.

3. JESTELLE DEVELOPMENT

The JEstelle development idea is presented in figure 3. A JEstelle project includes the JEstelle program file and possibly several user's Java class files (*.class) obtained form the corresponded java sources (*.java). User provided classes and the standard Java API determine resources that are accessible from the JEstelle program and must be visible to the JEstelle executive environment. As we want to take advantage of the existing Estelle support tools the JEstelle program must be converted to the Estelle equivalent. This is the moment where JEstelle Native Interface (JENI) comes in, replacing all Java statements with appropriate collection of Estelle primitives.

JEstelle Native Interface (JENI) is the programming interface that allows Java calls to be made from within the Estelle specification file. JENI is provided to help with the JEstelle support tools development. Using the JENI library (it means the implementation of JENI functionality) it is possible to use existing Estelle development tools to debug and validate JEstelle programs. Existing Estelle tools, in the simplest case, when desktop version of the software is to be used, do not need to be altered. The only practical requirement is to extend the native library to include the JENI implementation. JENI in the principle, is the interface containing the set of Estelle primitive functions and pure procedure declarations [1]. Using this set of primitives, operations on Java class objects can be expressed in the Estelle specification. The JEstelle Native Interface has been modeled on Java Native Interface [2]. Similarities of the both native programming interfaces should be helpful for Java programmers to more easily understand the Estelle code which makes use of JENI. Note, however, that the JENI native interface has been created with the objective to standardize the JEstelle to Estelle translation by automatic parsers rather than by programmers themselves. The objective of JENI is to support JEstelle development and not to facilitate to use Java functionality in Estelle programs by Estelle users. The designer should rather use JEstelle and automatic tools to generate the correct Estelle code.

Designing with JEstelle might be eased if special support tools are accessible to the designer. For example, by providing a JEstelle Graphical Editor, the JEstelle program framework can be expressed with easy-to-learn graphical representation. We will adapt the Estelle Graphical Editor [4]. It uses the well-known graphical representation of SDL (Specification and Description Language) [8]. This way the required knowledge concerning textual Estelle representation is minimal. Figure 5 presents recommended JEstelle development cycle using *JEstelle Development Package* and *XEDT* (Xwindow Estelle Development Toolset) execution environment. JEstelle Development Package consists of JEstelle/GR graphical editor, JEstelle/Estelle parser and Java implementation code generator. To work properly the Java implementation code generator needs the EstellLib package and all user Java source files. The EstelleLib [3] package provides Java classes for the Estelle extended state transition model implementation in pure Java code.

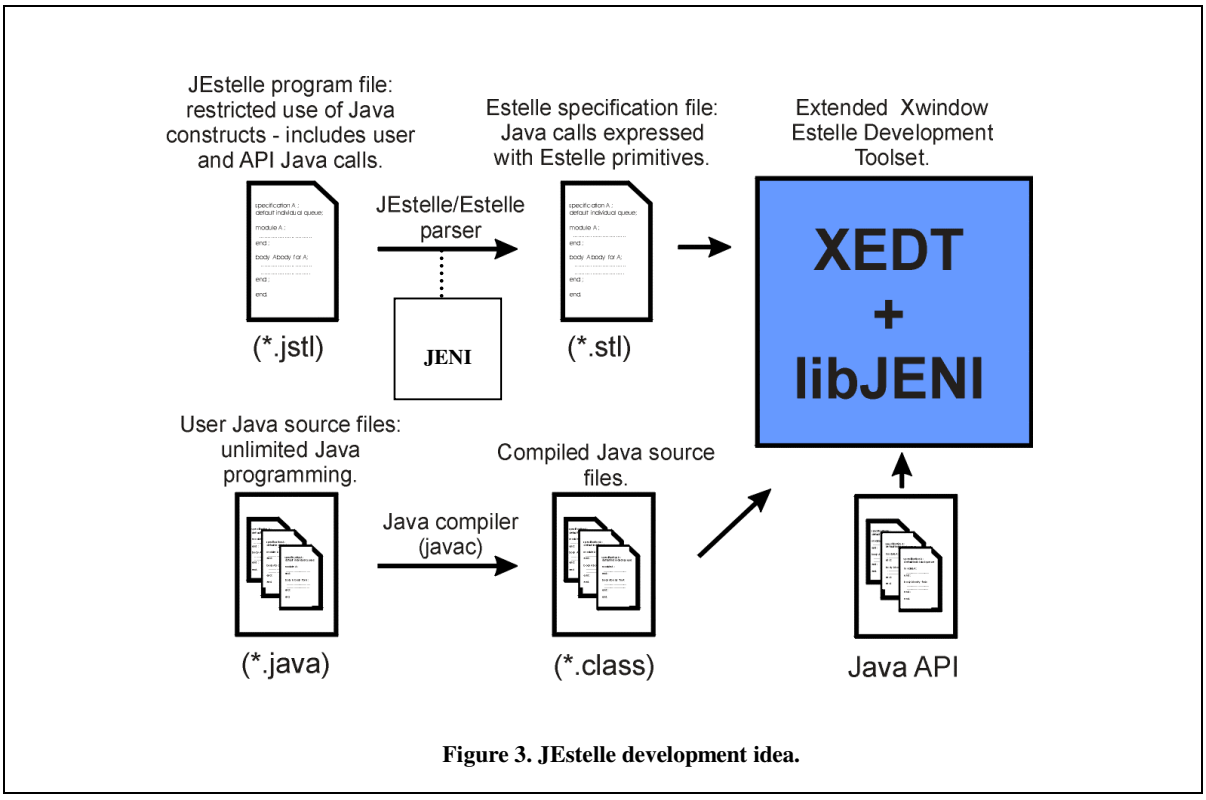

Figure 3. JEstelle development idea.

4. EXAMPLE

In this section we present a short example. Figure 4 presents definition of a native Java class that we want to use inside a JEstelle specification (figure 5). The JEstelle specification describes an example system consisting of two communicating components. The components simply exchange data and during the change of their internal state from RCV to SND the *PrintHello* method of the *Prog* user Java class is called and the *Hello from JEstelle* string is printed to the Java console. Figure 6 shows automatically generated Estelle specification where we can find out how the JENI interface is used to produce the correct Estelle code. The reader is referred to the JENI reference manual [5] for detailed information about JENI.

```
public class Prog {
  public void PrintHello(String[] args)
  {
    System.out.println(args[0]);
  }
};
```

Figure 4. Prog.java.

```
specification example systemactivity;
  default individual queue;
  timescale second;
   // channel definition

channel CommChannel(Send,Recv);
  by Send, Recv:
    SendMessage(int i);

module A activity(boolean m);
  ip p1 : CommChannel(Send);
     p2 : CommChannel(Recv);
  };

   // body definition for module header A

body Abody for A;
state SND, RCV;

  // local module data

  int d ;
  Prog obj ;
  String str ;
  String[] strArray ;

  // module initialization

initialize
  provided m
    to RCV {};
  provided not m
    to SND { d = 0 ; } ;

  // transition part

trans
  from SND
  to RCV
  name sending {
    output p1.SendMessage(d) ;
  };

trans
  from RCV
  to SND
  when p2.SendMessage
  name receiving {
    d++ ;
    str = new String(
                "Hello from JEstelle");
    strArray = new String[1] ;
    obj = new Prog() ;
    strArray[0] = str ;

    obj.PrintHello(strArray) ;

  };

}; // end of Abody definition


  // modules declaration

modvar
  A X, Y ;

  // specification module initialization

initialize
{

  // X is a receiver instance

  init X with Abody(true);
  init Y with Abody(false);
  connect X.p1 to Y.p2;
  connect X.p2 to Y.p1;
};

}. // specification module
```

Figure 5. example.jstl.

```
specification example systemactivity;
  default individual queue;
  timescale second;

  const
    MAXJVSIZE = 25 ;
  type
    String   = packed array[1..MAXJVSIZE]
               of char ;
    jeobject = integer ;
    jestring = integer ;
    jeobjectArray   = integer ;

    jevalue  = record
       z : jeboolean ;
       b : jebyte ;
```

```
      c : jechar ;
      s : jeshort ;
      i : jeint ;
      j : jelong ;
      f : jefloat ;
      d : jedouble ;
      l : jeobject ;
    dsc : char ;
  end ;

  value_arg = array[1..10] of jevalue ;
  jevalueArray = record
    num_of_args : integer ;
    arr         : value_arg ;
  end ;

{ JENI interface declarations }

function NewString(s : String) : jestring ;
         primitive ;
function NewObject(classSignature,
         constructorSignature : String ;
         args : jevalueArray) : jeobject ;
         primitive ;
function NewObjectArray(length : integer ;
         elementClass : String ;
         initialElement : jeobject) :
         jeobjectArray ; primitive ;

pure procedure CallVoidMethod(
         objectInstance : jeobject ;
         methodName,
         methodSignature : String ;
         args : jevalueArray) ; primitive ;
function InitReference(ref : jeobject) :
         jeobject ; primitive ;
function AssignReference(ref1, ref2 :
         jeobject) : jeobject ; primitive ;
pure procedure DecreaseReference(
         ref : jeobject) ; primitive ;


  { channel definition }

channel CommChannel(Send,Recv);
  by Send, Recv:
    SendMessage(i : Integer);

module A activity(m : boolean);
  ip p1 : CommChannel(Send);
     p2 : CommChannel(Recv);
  end;

  { body definition for module header A }

body Abody for A;
state SND, RCV;

  { local module data }

var
  d : integer ;
  obj : jeobject ;
  str : jestring ;
  strArray : jeobjectArray ;
  args : jevalueArray ;


  { module initialization }

initialize
  provided m
    to RCV
      begin
      end;
  provided not m
    to SND
      begin
        d := 0 ;
      end;

  { transition part }

trans
  from SND
  to RCV
  name sending : begin
    output p1.SendMessage(d) ;
  end;

trans
  from RCV
  to SND
  when p2.SendMessage
  name receiving : begin
    str := InitReference(str) ;
    strArray := InitReference(strArray) ;
    obj := InitReference(obj) ;
    str := AssignReference(str,
           NewString('Hello
            from JEstelle\0    ')) ;
    strArray := AssignReference(strArray,
                NewObjectArray(1,
                 'java/lang/String\0    ',
                  str)) ;

    args.num_of_args := 0 ;
    obj := AssignReference(obj,
     NewObject('Prog\0                    ',
               '()V\0                     ',
               args)) ;

    args.num_of_args := 1 ;
    args.arr[1].dsc := 'L' ;
    args.arr[1].l := strArray ;

    CallVoidMethod(obj,
              'PrintHello\0              ',
              '([Ljava/lang/String;)V\0 ',
              args) ;

    DecreaseReference(obj) ;
    DecreaseReference(strArray) ;
    DecreaseReference(str) ;

  end;

end; { end of Abody definition }


  { modules declaration }
```

```
modvar
  X, Y : A ;

   { specification module initialization }

initialize
  begin

    { X is a receiver instance }

    init X with Abody(true);
    init Y with Abody(false);
    connect X.p1 to Y.p2;
    connect X.p2 to Y.p1;
  end;

end. { specification module }
```

Figure 6. example.stl.

5. CURRENT DEVELOPMENT

We have successfully tested the idea using the Xwindow Estelle Development Toolset working on Unix/Linux based systems. The current version of the JENI interface implementation works on Linux systems and uses Java Native Interface (JNI – do not mislead with JENI) technology to achieve proper Java code execution. JENI implementation consists of the JENI library (libJENI) linked in with the Estelle simulator from the XEDT package and the JENI server that services requests for Java native code execution. JENI server and Estelle execution environment communicate using CORBA technology. The work on the JEstelle Development Package is on-going.

6. CONCLUSION

In this document the novel approach to Java distributed application development has been presented. In its novelty it merges the Estelle formalism with the ease-of-use of Java development. The possibility of using user designed libraries during the design phase gives designers a new effective method for distributed system development in which the implementation code can be used even in the design phase. The application area is potentially unlimited and ranges from formal protocol specification to the design of distributed Java applications and multi-agent systems. With JEstelle the system design reliability and readability can be improved and the implementation code generation process can be simplified reducing the time-to-market rate. Using JEstelle Development Package designers working with SDL (Specification and Description Language) and with Estelle in particular can quickly move to JEstelle and take advantages of the Java power. At the same time Java programmers can take advantage of formal distributed system modelling with minimal learning required.

7. ACKNOWLEDGEMENTS

We would like to thank Justin Templemore-Finlayson who was at the beginning of this work and Nico de Wet who made some corrections.

8. REFERENCES

[1] Estelle standard ISO/IEC 9074:1997.

[2] JNI - Java Native Interface specification: http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html.

[3] Justin Templemore-Finlayson “JEstelle - A super Java-Estelle idea” INT Evry, France 2000.

[4] Justin Templemore-Finlayson “The Estelle/GR Editor - User Guide” INT Evry, France, 2000.

[5] Marcin Czenko “JENI - JEstelle Native Interface - Reference Manual”, Tech. Report, LOR department, INT Evry, France, 2002.

[6] Marcin Czenko “Using Java in JEstelle specification”, Tech. Report, LOR department, INT Evry, France, 2002.

[7] Pascal standard ISO 7185.

[8] Specification and Description Language standard ITU (International Telecommunication Union) Recommendation Z.100.