EXTENSION OF SCALA LANGUAGE BY DISTRIBUTED AND PARALLEL COMPUTING TOOLS WITH LINDA COORDINATION SYSTEM

M. M. Glybovets, S. S. Gorohovskiy, and M. S. Stukalo

Abstract. *The paper describes a new library developed for Linda language implementation for Scala programming language (language support of the component programming software). The library uses client-server architecture and a database for the tuple space representation. It implements the search for similarities and "primitive" and additional operations. The library can be used for distributed and parallel application development.*

Keywords: *distributed computing, coordination systems.*

INTRODUCTION

A component-oriented approach to designing and implementing complex software systems develops in a sense an object-oriented paradigm. Its use becomes more efficient to develop large-scale projects and distributed systems, for example, for corporate applications [1, 2].

Using the potential of large-scale distributed systems needs program models explicitly operating with concepts of parallel cooperation among a large number of active entities that compose the unique application. Such a need has resulted in designing and introducing several coordination models together with individual languages that support these models. They were created to grant a framework to developers that would improve the module, promote the reuse of the components (sequential or already parallel), and to increase the cross-platformness and language compatibility. However, the concept of coordination is that these models differ in: just what is coordinated, by what means it is achieved, and what metaphors are applied to represent these concepts [3].

Coordination models are focused on problems (such as Linda) and use common data space. Such an approach provides the developers with a simple and efficient way to achieve many objectives; the main of them is optimal interaction of processes. Process communications is implemented here by introducing a small number of operations that can easily be used by programmers.

In the middle 1980s, D. Gelernter from Yale University proposed Linda [4], more likely an approach than an individual parallel programming language. In Linda, a parallel program consists of many processes, each operating as an ordinary sequential program. These processes have access to the common memory, a tuple being a storage unit. There are six basic operations to connect processes to the common memory.

Along with stylistic differences between the coordination models that influence the degree of separation of the computational and coordination parts, there is also difference in the application.

The *data-driven* coordination model is mainly used to parallelize computational problems. The *control-driven* coordination approach is usually applied to model systems. This can be explained by the fact that within the framework of the configuration component, the programmer's control of data is greater if languages that support *data-driven* coordination rather than *control-driven* coordination are used. As a rule, representatives of the first category try to coordinate data and representatives of the second one try to coordinate entities (can be not only ordinary processes but also devices, system components, etc.).

Many researchers consider that partial absence of progress in component software is because of shortcomings of the programming languages used to define and integrate components. The majority of languages propose only a limited support of the abstraction and composition of components. This concerns, for example, statically typed languages such as Java and C#.

The present paper is aimed at solving this problem by using basic ideas of Linda coordination model as regarded to the improvement of integration mechanisms of component programming languages with the use of Scala extension as an example.

1. SCALA PROGRAMMING LANGUAGE

Scala language was created in 2001–2004 in EPFL (programming methods laboratory) [5] within the framework of the improvement of the language support of component software. These studies were oriented mainly to two ideas. First, the component application language should be scaled, i.e., it should be possible to use the same concepts to describe both small and large parts of applications. Therefore, language developers were focused on the efficient implementation of abstraction, composition, and decomposition mechanisms rather than on introducing a large number of primitives, useful only at any one level of scaling. Second, it is quite natural to create a language that would unify and generalize object-oriented and functional programming. Some of the main technical innovations of Scala are a concept that unites these programming paradigms.

Scala language provides no primitives for parallel programming. The language core is constructed so as to simplify the creation of libraries supporting various parallelism models built over the current model of the parent language [6].

Let us also note the key aspects of Scala. The programs are similar in many respects to Java programs and can freely interact with Java code; the language includes a unified objective model in the sense that any value is an object and any operation is a call of the method; the language functionality implies that functions are full values. Powerful unified concepts of abstractions for types and values are introduced; we have flexible symmetric mixin constructs (*traits*) for the composition of classes and collections of methods; in view of the above, objects are decomposed by comparing them with a sample (is already implemented based on Java and .NET platforms).

For the key information on the language see http://www.scala-lang.org/, noteworthy is the book *Programming in Scala* [6] by M. Odersky, a co-author of the language.

2. LINDA FOR SCALA

The implementation of Linda for Scala described in this study is based on the object-oriented approach and consists in creating a specialized library. It is the conceptual elegance of Linda, including the associative structure of the tuple space, that has caused certain difficulties in the implementation [7]. To represent data types, classes rather than built-in data types are applied in Scala (a statically typed language). An abstract basic class is *LindaType*. Each class employed in tuples should be inherited from the basic class, which allows the developer using data types defined in this library and, what is important, creating new data types.

In *LindaType*, only a Boolean variable *formal.* is defined, which specifies whether the class is an actual or formal parameter. To transmit objects in a network, serialization built-in in Java is applied.

2.1 Data Types. Each data type of the library is inherited from the basic class *LindaType.* This allows implementing a simple transformation of classes *TupleN* defined in Scala (n-dimensional tuple, $1 \le n \le 22$) into *List*[*LindaType*], more convenient from the point of view of processing a tuple as a list. Limiting the number by 22 is due to how tuples are implemented in Scala. Such a representation has become necessary because the class *TupleN* does not support iteration, which considerably complicates the comparison of a tuple with anti-tuple. One more case for using a list as a data structure for a tuple is that many functions in Scala are defined to operate with lists.

To compare data types derivative from *LindaType*, a redefined equality statement (==) and an auxiliary method *canEqual* are used. Figure 1 determines the method of equality for *LindaInteger*.

The implementation of Linda for Scala has a set of predefined data types shown in Table 1.

For example, the fragment of a code in Fig. 2 shows how the data type *LindaInteger* is used.

The last row of the code shows an important feature of the library. Any actual parameter can be transformed into formal one and vice versa by using methods *toFormal* and *toActual* defined in the basic class *LindaType*.

```
override def equals (other : Any) = other match {
 case that: LindaInteger =>
  (that canEqual this) &&
  (this.Value == that.Value)
 case _ =>
  false
 }
 def canEqual (other: Any) = other.isInstanceOf[LindaInteger]
```

Fig. 1

TABLE 1

Class	Data Type
LindaDouble	Double
LindaFloat	Float
LindaInteger	Int
LindaLong	Long
LindaString	String

```
val lint:LindaInteger = 20        //actual type

val lint2 = new LindaInteger (15)     //actual type
lint.toFormal   //now lint is a formal parameter that can be applied
                  for in or rd operation
```

Fig. 2

```
case class LindaPoint (coord1 : Int, coord2 : Int) extends LindaType {
 val x = coord1
 val y = coord2

 override def getType = "point" //method to obtain the type signature
 override def toString = x+" "+y //method to transform values into a string
 }
```

Fig. 3

```
def getSignature (tuple : List[LindaType]) : String ={
 var output = ""
 tuple.foreach(arg => output = output +" " + arg.getType)
 output.drop(1)
 }
```

Fig. 4

Let us describe how new data types can be created and used in Linda for Scala. For the beginning, it is necessary to define a new data type (Fig. 3) as a derivative from *LindaType*.

The keyword *case* before the class definition specifies that the pattern matching mechanism built-in to Scala is applicable for this class. To save tuples in a database, the signature of each element of the tuple should be obtained by the function *getType*. Then these signatures form a string necessary to find tuples in the database. The function of obtaining such a string is determined in the singleton object *LindaTuple* (Fig. 4).

For new classes, it is also necessary to determine a method to compare objects of this class with each other similarly to Fig. 1.

2.2. Features of the Implementation of the Operations. The Linda coordination model contains a small number of operations of process communication and data space distribution. Since communication operations should accept any data type at the input and have simple syntax, it is logical to declare that the input parameter of these operations should be *Any*, which is the basic type in the hierarchy of Scala classes. Actually, a Scala tuple (*TupleN*) to be transformed into a Linda tuple arrives at the input of operations. Therefore, the number of input objects is restricted to 22, which is due to how tuples are implemented in Scala. A tuple can be transformed into the necessary form by the function *createLindaTupl* from the singletone object *LindaTuple*. This function is used once in the system, when a client passes a request from a process to the server. After that, the tuple is a list of values of type *LindaType*.

```
package org.linda.Test
import org.linda.LindaTypes._
import org.linda.Client._

object process extends Application {
  val i1 = new LindaInteger (28) //creating new data objects
  val i2 = new LindaInteger (34)
  val i3 = new LindaInteger (14)
  val s = new LindaString ("hello")

  val client = new Client ///creating a new client
  client.out (i1, i2, s) //executing the out operation
  client.out (i1, i3, s)
  i2.toFormal            //transforming an actual parameter into a formal one
  client.in(i1, i2, s)   //executing the in operation
}
```

Fig. 5

```
def in (antiTuple : List[LindaType]) : List[LindaType]= {
 println(" db in")
 val signature = LindaTuple.getSignature(antiTuple) //obtaining tuple signature
 val tuple = chooseTuple(find(antiTuple)) //searching for and choosing the tuple
  if (db(signature).length == 1) //if the chosen tuple is unique with such a signature
                db - signature //then delete the signature from the mapping
   else
  db(signature) = deleteTuple(tuple,db(signature)) //deleting the tuple from the database
   tuple //return the tuple
 }
```

Fig. 6

To relate the process with the space of tuples, it should create a new copy of the client and use it to operate with the server database (Fig. 5).

Figure 5 shows the key points in operating with a client: creating new objects of values, creating a client, and executing operations. Note that the last operation *in* returns one of two tuples: (28, 34, "hello") or (28, 14, "hello") since the actual parameter *i*2 has become formal one, has really become a pattern of type *<Int>*.

2.3. Database for Tuples. The client-server link is used to transfer requests to the database, which implements the common distributed space of tuples and is responsible for searching for tuples in this space. The structure for data storage is mappings *Map* of the form *[signature : String –> List [List [Lindatype]]]*, where each signature string is associated with a list of tuples with the same signature. Such arrangement of tuples simplifies the search for a necessary tuple. To find the correspondence with the antituple, first, the signature of the antituple is obtained, then the list of tuples with the corresponding signature is selected from the mapping, and the function *find* is used to find all the tuples that satisfy the pattern obtained from the antituple. If there are more than one tuple, the function *choosetuple* randomly chooses one of them. Then it is removed from the database and the updated list is written in the mapping. This process is detailed in Fig. 6.

2.4. Linda Client. The client is responsible for passing requests from the process to a server and for returning the result. For the server-client communication, Java-sockets are used. When executing some method of the client, for example, operations *in* or *out*, a new socket to communicate with the server and streams) to record and read objects from the server, a request is passed to the server are created, an answer is expected, and the values obtained are returned to the process. Figure 7 describes the implementation of the operation *in* for a client.

```
def in (input : Any) : List[LindaType] = {
val socket = new Socket(ia, port) //creating a new socket
val outStream = new ObjectOutputStream(new DataOutputStream(socket.getOutputStream()))
 //creating a new output stream
val inStream = new ObjectInputStream (new DataInputStream(socket.getInputStream()))
 //creating a new input stream
val antiTuple = LindaTuple.createLindaTuple(input)                    //creating a Linda tuple
                    outStream.writeObject(("in","",antiTuple))        //passing the request to the server
                    val tuple = inStream.readObject().asInstanceOf[List[LindaType]]          //reading an
                    object from the server
                    outStream.close()          //closing streams
                    inStream.close()
                     tuple          //returning the found tuple of the process}
```

Fig. 7

```
case "in" => {while (db.find(tuple).isEmpty)
          db.wait()
   outStream.writeObject(db.in(tuple))
}
```

Fig. 8

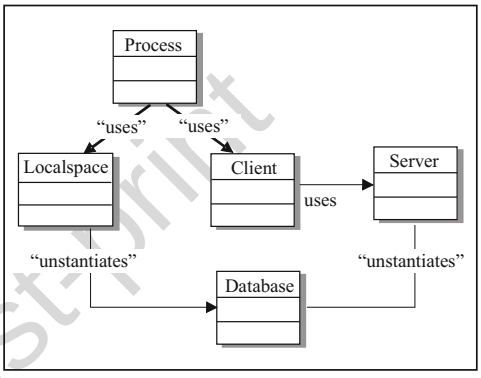

Fig. 9

2.5. Linda Server. The task of the server is to expect requests from clients and to execute them. To execute each request, an isolated stream is started, which makes requests to the database. The streams can be executed in parallel. Each stream opens *streams* to read and write objects from/to the client. It is in streams is implemented that the operations such as *in* or *rd* suspend the process execution until a necessary tuple appears the in the database. The streams are synchronized with respect to the database object with the use of the Java syncronized method: *db.synchronized* {…}, and if there is no corresponding tuple in the database, the command *db.wait*() suspends the execution of the stream. Figure 8 describes the implementation of the operation *in* on the server.

2.6. Local Tuple Space. The local space of tuples is used to support additional operations such as *collect* or *copy-collect*, where not a separate tuple but a list of tuples to be saved is returned from the distributed space of tuples. The framework of the client-server interaction with regard for the local space of tuples is presented in Fig. 9, whence it is seen that the local data space and the server use the same class of the database to create the space of tuples. For the process, the local space performs the same functions as a server for a client does, namely, passes requests of the process to the database.

2.7. Additional Operations. This implementation of Linda for Scala implements the following additional primitives or operations: *collect*, *copy-collect*, *query*, *copyquery.* The operations *copy-collect* and *collect* were described earlier. The last two operations have also a predicate form, which does not suspend the process but returns the value equal to *true* or *false.* The operation *query* is similar to the operation *collect*; the difference is that the number of elements to be returned from the space of tuples is specified in this case.

Respectively, the operation *copyquery* has the same functionality as previous ones, apart from deleting found tuples from the tuple space.

CONCLUSIONS

Obviously, to solve complex and large-scale computational problems, methods for paralleling and coordinating isolated computational problems should be used and improved. D. Gelerter proposed an efficient program solution to implement them. The Linda coordination model he developed can be added to virtually any programming language.

The Linda model is based on the concept of common data space; to access to this space, processes have a small but sufficient set of primitives or operations such that the process can place, read or delete a tuple from the common data space. Processes communicate only through the common data space, which allows their space and time decoupling.

The present paper describes the library of classes (created by the authors) (http://www.ukma.kiev.ua/~gor/LindaforScala/) oriented to implementing Linda for Scala programming languages (language support of the component software). The library uses the client-server architecture, a database to represent the tuple space. The pattern matching mechanism and "primitive" and additional operations are implemented in the library. The library can be used to develop distributed parallel applications.

The library requires Eclipse development environment [7] with a plug-in to operate with Scala language or an individual packet with a command line interpreter and a compiler, which can be found on the Scala web site [8].

The library is cross-platform since Scala language uses Java for operation and is compiled in a Java byte code. The library was developed in Mac OS X version 10.5.7 with Java 2 Runtime Environment version 1.5.0_16, version of Eclipse SDK 3.4.2 and Scala 2.7.4.

REFERENCES

1. P. I. Andon, A. Yu. Doroshenko, O. A. Letichevsky, O. L. Perevozchikova, et al., "Models, methods, and technologies in parallel programming," in: State of the Art and Prospects of the Development of Information Sciences in Ukraine [in Ukrainian], Naukova Dumka, Kyiv (2010), pp. 242–258.
2. http://folding.stanford.edu/.
3. N. N. Glibovets and V. M. Fedorchenko, "Simplified infrastructure for the transformation of XML models," Cybern. Syst. Analysis, **46**, No. 1, 93–97 (2010).
4. D. Gelernter, "Generative communication in Linda," ACM Trans. on Programming Languages and Systems (TOPLAS), **7**, No. 1, 80–112 (1985), http://www.ece.rutgers.edu/~parashar/Classes/03-04/ece572/papers/gencommlinda.pdf.
5. http://www.epfl.ch.
6. M. Odersky, L. Spoon, and B. Venners, Programming in Scala, http://www.artima.com/shop/programming_in_scala
7. http://www.eclipse.org/.
8. http://www.scala-lang.org/.