Implementing Incremental Code Migration with XML

Wolfgang Emmerich, Cecilia Mascolo[*] and Anthony Finkelstein
Dept. of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{W.Emmerich|C.Mascolo|A.Finkelstein}@cs.ucl.ac.uk

ABSTRACT
We demonstrate how XML and related technologies can be used for code mobility at any granularity, thus overcoming the restrictions of existing approaches. By not fixing a particular granularity for mobile code, we enable complete programs as well as individual lines of code to be sent across the network. We define the concept of incremental code mobility as the ability to migrate and add, remove, or replace code fragments (i.e., increments) in a remote program. The combination of fine-grained and incremental migration achieves a previously unavailable degree of flexibility. We examine the application of incremental and fine-grained code migration to a variety of domains, including user interface management, application management on mobile thin clients, for example PDAs, and management of distributed documents.

Keywords
Incremental Code Migration, XML Technologies

1 INTRODUCTION

The increasing popularity of Java and the spread of Web-based technologies are contributing to a growing interest in dynamic and reconfigurable distributed system architectures. Such reconfiguration can be achieved with code migration, transferring fragments of code across the network, from one host to another.

The potential mobility range is however wider, starting from simple data mobility, where information is transferred. Examples are the actual parameters that are passed to a remote procedure call or the web page that is returned to a get request in the HTTP protocol. At a level above this, code mobility allows the migration of executable code: browsers loading applet classes from remote servers are very common examples of code mobility. Java-based technologies, for instance, Java RMI [16] and Java Virtual Machines, such as those built into Web browsers, offer a mobility granularity at a class level. M0 [30], Telescript [19], Obliq [6] are non Java based mobile code languages and systems. Mobile agents [31, 14], in which code and data move together, can be considered the highest level of mobility that can be achieved in a logical context.

Several application domains need a more flexible approach to code mobility than can be achieved with Java and with mobile agents in general. This flexibility can either be required as a result of low network bandwidth or scalability. The 9,600 baud bandwidth between a server and a GSM mobile phone cannot cope with downloading large amounts of Java byte code from a server. Scalability requirements can mean for example, that applications on several thousand clients have to be kept in sync and be updated with new code fragments.

In this paper we show how to achieve more fine-grained mobility than in the approaches that are based on Java. We demonstrate that the unit of mobility can be decomposed from an agent or class level, if necessary, down to the level of individual statements. We can then support incremental insertion or substitution of, possibly small, code fragments and open new application areas for code mobility such as management of applications on mobile thin clients, for example wireless connected personal digital assistants (PDAs), user interface construction and document management.

This work builds on the formal foundation for fine-grained code mobility that was established in [21]. That paper develops a theoretical model for fine-grained mobility at the level of single statements or variables and argues that the potential of code mobility is submerged by the capability of the most commonly used language for code mobility, i.e., Java. In this paper, we share that vision and focus on an implementation of fine-grained mobility using standardized and widely available technology.

It has been identified that mobile code is a design concept that is independent of technology and can be embodied in various ways [13] in different technologies. The eXtensible Markup Language (XML) [5] can be exploited to achieve code mobility at a very fine-grained level. XML has not been designed for code mobility, however it happens to have some

[*]This author was partially supported by Dipartimento di Scienze dell'Informazione, Università di Bologna, Bologna, Italy.

interesting characteristics, mainly related to flexibility, that allow its use for code migration. In particular, we will exploit the tree structure of XML documents and then use XML related technologies, such as XPointer [20] and the Document Object Model (DOM) [3] to modify programs dynamically. The availability of this technology considerably simplifies the construction of application-specific languages and their interpreters.

This paper is further structured as follows. In Section 2, we discuss related work, most notably XML, and alternative approaches to logical code mobility. In Section 3, we show how XML supports the definition of high-level languages and how incremental code mobility can be defined with XML. In Section 4, we argue that incremental code mobility has the potential for a set of application areas such as user interface development, management of applications on mobile thin clients, and consistency management of distributed documents. We give examples of the application of our approach in these domains. In Section 5 we demonstrate the implementation of mobile code systems supported by off-the-shelf XML products. Section 6 evaluates the approach and identifies strengths and weaknesses. Section 7 contains a summary of the work done and some future work.

2 OVERVIEW OF XML AND LOGICAL MOBILITY

Physical mobility is concerned with the physical movement of hosts, such as notebooks, PDAs, mobile phones and wearable computers. Logical mobility is the ability to transfer data and/or code from one host to another by using a network. This paper focuses on logical mobility, though the approach is also applicable to information that transits between physically mobile hosts; in fact, Section 4 discusses how our work can be applied to manage applications deployed on PDAs. Logical mobility encompasses data and code mobility.

Data mobility is a very common mechanism and often used to exchange or spread information among different hosts distributed on a network. Data mobility can be achieved by passing parameters to remote procedure calls, object requests or the put and get operations of the file transfer protocol. With the introduction of the Internet and the World-Wide-Web the Hyper Text Markup Language (HTML) has been used as the predominant format for data that moves between hosts on the Internet.

XML [5] is the basis for next generation markup languages for the Internet. XML is a subset of the Standard Generalized Mark-up Language (SGML) [15]. Unlike HTML both XML and SGML allow users to define their own set of mark-up tags for structuring documents. These user-defined markup tags are defined in document type definitions (DTDs). A DTD is a grammar that defines the syntax of documents. XML documents always declare a reference to their DTD in order to enable generic parsers to obtain the specification of the grammar. Thus with the advent of XML, different formats for transferable data can be defined. Many different DTDs have been standardized to encode specific notations in XML. Examples of software engineering applications of XML include the UXF [29] and the XMI [23] DTDs. They both define transfer formats for UML [4] models.

XML is not only useful for publication of documents on the World-Wide-Web, but can also be used as an application-specific transport protocol in distributed system construction. We report in [11] about the use of XML for the transport of data between different distributed and heterogeneous components of a financial trading system. That system uses XML documents as parameters to CORBA object requests. Moreover, the OMG have requested proposals for the interoperability between their Interface Definition Language and XML [24] that will address the seamless interchange of XML documents and equivalent complex values of IDL data types.

Data and code mobility in Java are supported through object serialization and class loading. The status of objects can be serialized and transferred from one host to another while the class loading strategies can vary, depending on the application. For instance, the Netscape class loader downloads applet classes from the web server of the containing HTML page; the Java RMI class loader allows the application to download the classes of the objects remotely passed as parameters at run time. The class of the moved object can migrate onto the new host or it can be fetched from a remote server. Many different technologies have been built on top of these simple mechanisms.

Two more sophisticated mobile code paradigms are classified in [13] as remote evaluation and mobile agents. Remote evaluation allows the proactive shipping of code to a remote host [28] to be executed. Mobile agents are autonomous objects carrying their state and code that proactively move across the network. Many new systems have been developed to support mobile agents [17, 31, 14]. Agent mobility requires the migration of both code and state of the agent at the same time and they can move proactively performing tasks on behalf of users.

3 SPECIFYING INCREMENTAL CODE MOBILITY WITH XML

XML provides a flexible approach to describe data structures. We now show that XML can also be used to describe code. XML DTDs are, in fact, very similar to attribute grammars [18]. Each element of an XML DTD corresponds to a production of a grammar. The contents of the element define the right-hand side of the production. Contents can be declared as enumerations of further elements, element sequences or element alternatives. These give the same expressive power to DTDs as BNFs have for context free grammars. The markup tags, as well as the PCDATA that is included in unrefined DTD elements, define terminal symbols. Elements of XML DTDs can be attributed. These attributes can

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!ELEMENT KarelProgram (turnon|go|
       turnleft| pickbeeper|putbeeper|
       turnoff|times)*>
  <!ELEMENT turnon EMPTY>
  <!ELEMENT go EMPTY>
  <!ELEMENT turnleft EMPTY>
  <!ELEMENT pickbeeper EMPTY>
  <!ELEMENT putbeeper EMPTY>
  <!ELEMENT turnoff EMPTY>
  <!ELEMENT times (turnon|go|turnleft|
       pickbeeper|putbeeper|turnoff|times)*>
  <!ATTLIST times howoften CDATA #REQUIRED>
```

Figure 1: The DTD for Karel's Instruction Set.

be used to store the value of identifiers, constants or static semantic information, such as symbol tables and static types. Thus, XML DTDs can be used to define the abstract syntax of programming languages. We refer to documents that are instances of such DTDs as XML programs. XML programs can be interpreted and in Section 5 we discuss how such interpreters can be constructed using XML technologies. When XML programs are sent from one host to another we effectively achieve code mobility.

In order to demonstrate these ideas, we consider a very simple programming language to instruct Karel, the robot. The language has first been defined in [25]. In this paper we only consider a subset of it for reasons of brevity. Karel's language has a set of primitives. These include `turnon`, to switch the robot on, `go` to make it proceed one step into its current direction, `turnleft` to change the robots current direction by turning left, `pickbeeper` and `putbeeper` to get and dispose of beeper objects and `turnoff` to turn Karel off. Moreover, Karel's programming language includes a number of control structures for repetition and conditional execution. Here, we only consider the `times` statement. It repeats a

```
<?xml version="1.0"?>
<!DOCTYPE KarelProgram SYSTEM "karel.dtd">
<KarelProgram>
  <turnon/>
  <times howoften="2">
    <turnleft/>
    <times howoften="2">
    <go/>
    </times>
  </times>
  <turnoff/>
</KarelProgram>
```

Figure 2: An XML Program for Karel.

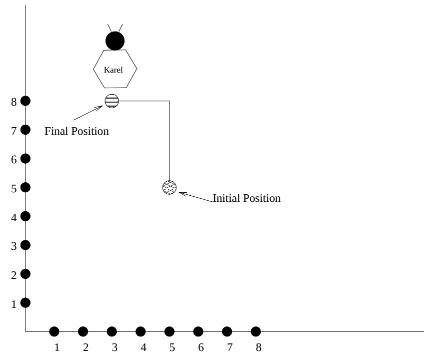

Figure 3: The Actions of the Robot.

cycle of commands for a given number of times. Figure 1 shows the syntax of the subset of Karel's programming language defined as an XML DTD.

Figure 2 shows an instance of the DTD in Figure 1. This instance is an XML program that instructs Karel first to turn left, then to proceed two steps, turn left again and proceed two more steps. Karel's route is shown in Figure 3. If we imagine that Karel is a real robot, that is instructed from some control host by sending these XML programs via, for example, a radio network, we have then achieved logical code mobility with XML.

Unlike Java programs, which are sent in a compiled form, XML programs are transferred as source code and then interpreted on a remote host. Unlike Java, XML does not confine us to sending coarse-grained units of code; XML documents do not need to begin with the root of the DTD, they can also start with other symbols of the grammar. This enables us to specify sub-programs and even individual statements. We refer to such code fragments as XML program increments. Hence, we can specify complete programs as well as arbitrarily fine-grained increments in XML.

As Karel is controlled by a slow radio network, we want to avoid re-sending the whole program but rather incrementally send new program increments. Figure 4 shows such a fine-grained program increment. We can imagine that we want to change the behaviour of Karel by replacing the `turnleft` statement with this increment and thus change the behaviour of Karel making it turn right instead of left [1]. Note, that the

[1] Because the Karel language does not have a primitive to turn right, we

```
<?xml version="1.0"?>
<!DOCTYPE times SYSTEM "karel.dtd">
  <times howoften="3">
    <turnleft/>
  </times>
```

Figure 4: XML Code Increment.

```
root().child(1,times).child(1,turnleft)
```

Figure 5: XPointer Address for Increment.

DOCTYPE that determines the root symbol in this increment is times rather than KarelProgram in the full XML program in Figure 2.

The question that arises is how we specify the insertion or replacement of program increments. Addressing particular locations in an HTML document is achieved by "anchors". These anchors, however, cannot be defined by users who do not have control over the document but have to be included by the author of the document. Likewise in our approach, the sender of an increment does not have control over the program once it has been sent. However, we cannot assume that programmers identify anchors or other labels a-priori that could then later be used for incremental code insertion or replacements.

To solve this problem, we use XPointer, an XML-related standard. XPointer is part of the XLink specification [20] and overcomes the limitation of HTML by supporting navigations within XML documents. These navigations are capable of addressing every document component without having to modify the document itself. We use XPointer to identify that component of an existing XML program that we want to replace with a new increment.

Going back to our example, Figure 5 shows an XPointer expression that determines the Karel program statement that we want to replace. The XPointer expression starts from the root of the program and then selects the first statement of type times, and in that statement it selects the turnleft statement. Thus, by specifying a fragment of a program in XML together with an XPointer expression, we can express incremental code mobility. Figure 6 shows how Karel's behaviour will differ after the new increment has replaced the turnleft statement.

have to implement turning right by turning left three times.

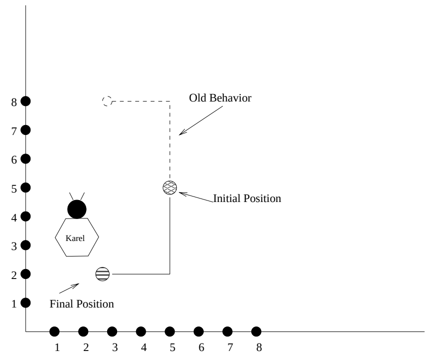

Figure 6: The Incremental Change to Karel's Behaviour.

4 APPLICATIONS

In the preceding section we have presented our ideas through a deliberately simple example. In this section, we describe application domains that will benefit from incremental code mobility with XML. These include user interfaces engines, the management of applications on portable digital assistants, and the flexible co-ordination of consistency checks in distributed documents.

User-Interface Engines

The installation and administration of large-scale systems with thousands of clients is a potential application for incremental code mobility. The departure control system of airlines that are used to handle check-ins are good examples. For large airlines or alliances, the clients implementing the user interface of such systems have to be deployed on several thousand machines, distributed across the globe. The machines are not necessarily owned by the airlines but are rather temporarily rented from airport authorities, which want to keep tight control on updates of software. Thus airlines cannot frequently update the software that is installed on these machines.

It would be possible to accommodate changes by deploying a Java Virtual Machine on each of these systems and downloading front-end applications from centralized servers. The Java approach, however, has two disadvantages. First it requires code of substantial size to be downloaded from a server, possibly through slow dial-up networks. Second, the Java code needs to be changed whenever the user interface needs to be changed. These limitations can be overcome by installing a general-purpose user interface engine onto each of the client machines that interpret high-level user interface descriptions.

XwingML is a DTD for a user interface description language [27]. It provides markup tags for all Java Swing user interface components and also provides an interpreter for XwingML documents that generates the desired user interfaces. Applying our approach of code mobility with XML, the high-level descriptions of user interfaces can be sent from a centralized server to all distributed client hosts. As the user interface descriptions are rather small compared to the size of the Java byte code of the full user interface application, we avoid the first of the problems above. The second limitation is overcome because the user interface description is just an XML document, which can be generated by server applications.

Incremental mobility can be applied successfully in this context. If the displayed window needs to be updated, for example by adding or replacing some buttons, an XML code increment can be sent to the user interface engine. The program increment can be dynamically integrated with the original XML code for the window, thus making the window change its appearance.

Application Management on Mobile PDAs
An interesting application for our XML-based approach to code mobility arises when logical mobility meets physical mobility. Lightweight computing devices, such as Personal Digital Assistants (PDAs) are starting to be used for mission critical computing and, therefore, need to be integrated into enterprise computing environments. In these settings, it is important for all PDAs to run the same set of applications. An example for such a PDA deployment is the New York Stock Exchange (NYSE). NYSE has equipped its traders with PDAs, that are used for trade data entry and automated transmission between trade data and back office trade settling systems.

The applications that are used in financial markets have to evolve rapidly. Financial analysts invent new products known as derivatives on a regular basis. Once such a product has been created, the trading applications need to be adjusted and be updated to support trading in these new derivatives. If the machines used were wired workstations it would be feasible to transfer and replace the complete code when needed, though the incremental approach described in this paper could also be applied. Our approach becomes essential when the devices used are thin clients such as PDAs; in this case incremental code updates are a valuable option, considering occasional disconnection of PDAs and slow IRDA or radio network connectivity.

To take advantage of our approach application developers have to devise an XML-based scripting language for developing trading applications. They also have to build an interpreter for this language which is then deployed on each PDA. Whenever an application needs to be changed, a program increment can be added to a list of updates that are kept on the server to which PDAs connect. Once a PDA physically enters the trading room and establishes connection to the server, the server first checks the patch-level of the applications on that PDA. The server will then incrementally send all application updates that are not yet deployed on the PDA.

The definition of an application-specific language and its implementation in an interpreter may sound difficult to accomplish. It is, however, well supported. The application-specific language can refer to XWingML or MoDAL [1] for user interface definition purposes. The implementation of an interpreter is simplified by the availability of light-weight XML parsers. Moreover, Java Virtual Machines have already been developed for PDAs, such as 3COM's Palm Pilot [26] and the Symbian operating system that will run on the next generation of mobile phones.

Consistency Management of Distributed Documents
In [10], we describe an approach for managing the consistency of distributed documents. We assume that documents are represented in XML themselves. This is a fair assumption, because Microsoft's Office 2000 can save documents in XML format, IBM's Visual Age environment uses XML as representation scheme for its project repository and most CASE tools can export UML diagrams in XMI.

```
<ConsistencyRule id="r1" type="CT">
 <id>r1</id>
 <Description>
 For every instance in a collaboration
 diagram there must be a class in
 a class diagram with the same name.
 </Description>
 <Source>
  <XPointer>
   root().child(all,Package).
    (all,CollaborationDiagram).
    (all,Collaboration).(all,Instance)
  </XPointer>
 </Source>
 <Destination>
  <XPointer>
   root().child(all,Package).
    (all,ClassDiagram).(all,Class)
  </XPointer>
 </Destination>
 <Condition expsource="origin().attr(CLASS)"
  op="equal"
  expdest="origin().attr(NAME)"/>
</ConsistencyRule>
```

Figure 7: A Consistency Rule in XML Format.

The consistency management approach of [10] is based on a language that can express consistency rules. This language is, in fact, an XML programming language and Figure 7 shows an example consistency rule expressed in the language.

The rule is based on the UXF DTD [29] for UML and demands that for each object in each collaboration diagram, there is a class in a class diagram whose name equals the type of the object. In [10], we also discuss the interpreter that executes these consistency rules in order to check the consistency of XML documents. The result of such a check for a rule is a set of XLink expressions that link consistent document fragments with each other and inconsistent document fragments to an indicator of such inconsistency.

The approach as described in [10] uses one set of rules and one rule interpreter. This is rather inflexible as every member of a software development team has to work against the same set of rules. Moreover, the centralized interpretation of rules creates a bottleneck that can be avoided if we have multiple rule interpreters on each developer's machine. The rule interpreter would then only have those sets of rules that the developer needs to check consistency of the documents she produced locally. We can even have dedicated interpreters for particular subgroups of the development team in order to check consistency between documents produced by dif-

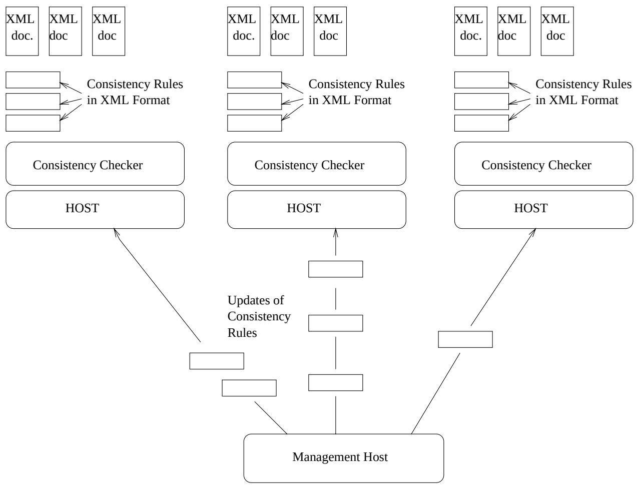

Figure 8: Consistency Management Architecture.

ferent team members and then at a higher level there can be rule sets that check for project-wide consistency.

A significant number of rules will be defined for realistic projects. However, for each developer, only a subset of rules will be relevant. This subset is determined by the types of documents that the developer currently modifies and also by the state of the project. Closer to deadlines more rules may be enforced than during initial stages. Thus, the sets of rules that are active at each of our rule interpreters cannot be static but has to change during the course of the project. In order to accommodate such changes, the set of rules that are active at each interpreter have to be changed. As each interpreter runs a different set of rules, this cannot be achieved using a broadcast or a shared storage. New rules have rather to be added and existing rules may have to be deleted from the rule sets of individual interpreters. Using our approach, these changes can be triggered by a consistency supervisor component that uses incremental code migration to pass the XML-encoded consistency rules to the different rule interpreters involved.

Figure 8 shows the overall architecture of this approach. Each developer's workstation and group and project servers run an interpreter for XML-consistency rules. The consistency supervisor manages the rule set of consistency rules that are active for each of these interpreters and moves new rules incrementally to these interpreters, if necessary.

We have so far shown how we can use XML to define programs and how we can define the update of code in an incremental fashion. In the next section we describe how we can utilize off-the-shelf XML technology in order to implement interpreters for application specific languages and how these interpreters implement incremental code updates.

5 IMPLEMENTATION OF THE APPROACH

After a programming language has been specified, an interpreter for this language needs to be implemented. We first show how significantly off-the-shelf XML technology, most notably XML parsers and the implementations of the Document Object Model (DOM) [3], simplify the construction of such interpreters. Then, we explain how the communication between sender and receiver can be achieved using distributed object technology. Finally, we focus on the implementation of incremental code mobility, demonstrate how XPointer processors support locating the increment to be updated, and how the DOM supports incremental syntax tree modifications.

```
import org.w3c.dom.*;         //DOM API
import com.ibm.xml.parser.*;  //XML Parser

public void execute(String program,
                    String update_location){
 ...
 //create a new parser for Karel Programs
 Parser parser=new Parser("Karel.dtd");
 InputStream is;
 // parser to read input stream from program
 is=new StringBufferInputStream(program);
 // root of parse tree for program in inc
 Document inc=parser.readStream(is);
 ...
}
```

Figure 9: Translating XML Program into an AST.

Interpreter Implementation

The first stage of interpreting a program involves the validation of the syntactic correctness. As a result of that stage, interpreters produce an attributed abstract syntax tree (AST) of the program. If the program is written in XML, both tasks can be entirely delegated to a validating XML parser. We use IBM's XML4J [2] but many other validating XML parsers exist. Figure 9 shows the use of the XML4J parser in our Java-based Karel interpreter. When invoking `parse` on the Karel code of Figure 2 the XML parser will construct the parse tree that is graphically represented in Figure 10.

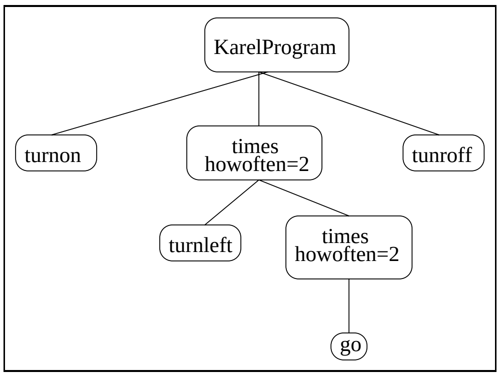

Figure 10: Abstract Syntax Tree for Karel's Program.

The next stage of the interpretation is a static semantic analysis that checks, for example, the uniqueness of identifiers or the correct typing of expressions. This is often done while the interpreter is executing the code in order to avoid several traversals of the abstract syntax tree. Thus, while traversing the tree and visiting each node, the interpreter first checks for violations of the static semantics and then executes the operation that the node represents. Operations for traversals through ASTs that have been constructed from XML documents are standardized by the Document Object Model (DOM) [3] and are implemented in off-the-shelf products, such as IBM's XML4J. The DOM traversal operations support obtaining all the children of a node, querying the type of the node, obtaining values of node attributes and so on.

Figure 11 shows an excerpt of the Karel interpreter that traverses the abstract syntax tree and executes statements for each AST node. The actions usually modify some state variables. In the case of Karel, these state variables indicate whether the robot has been switched on, its current position and direction and the number of items that it has picked up. The interpretation is then performed as a recursive method `execute`, which is initially passed the root node of the AST tree. It then examines the type of node and performs the appropriate action. For the root node, it recursively calls execute for all its child nodes. For instance, for a node of type `go`, it adds the current direction to its co-ordinates. We note that for Karel, the implementation of each command requires a few lines of code and in total is about 50 lines of Java code.

Code Mobility
In order to support code mobility, an XML program is sent from a remote host rather than being read it from a local file system. Any transfer protocol could be used for this purpose, however in our example we have used Java/RMI [16] due to its simplicity. XML programs are passed as parameter of remote invocations. The object invoked implements the interpreter for the XML program. The interpreter understands all possible XML programs written according to the DTD of Karel's language. Figure 12 shows the migration of the code to the remote host.

In order to facilitate the remote communication that transmits the mobile code, the Karel Interpreter declares the remote interface `Karel` as shown in Figure 13. That interface is implemented by the Karel interpreter. This enables a controller that resides on one host to send Karel programs for interpretation on a different host. Note that we do not transfer the DTD together with the code but rather assume that the DTD is stored locally. This choice derives from the observation that the interpreter implementation is very tightly linked to the DTD, because the DTD is the grammar of the language and every interpreter is dependent on the grammar of the language that it executes.

Incremental Code Mobility
So far, we have shown how to parse and interpret the program, which is passed as the first parameter to the `execute` method in Figure 13. The second parameter is an XPointer expression. If this XPointer expression is not empty and is well-formed, it will identify a node in the abstract syntax tree that needs to be replaced with the program increment that is

```
import org.w3c.dom.*;        // DOM API
import com.ibm.xml.parser.*; // XML parser

class KarelExecutor {
 //the position and direction of Karel:
 private int x_pos=5, y_pos=5;
 private int x_direction=1, y_direction=0;
 private int num_beepers=0; //items #
 private boolean on=false;// status

 public void execute(Node n) {
  if(n.getNodeName().equals("KarelProgram")){
   NodeList children=n.getChildNodes();
   Node command;
   for (int i=0; i<children.getLength();i++){
     execute(children.item(i));
   }
  } else if (n.getNodeName().equals("go")){
   if (on) {
    x_pos=x_pos+x_direction;
    y_pos=y_pos+y_direction;
   }
  } else ...
 }
}
```

Figure 11: Traversing the AST during Interpretation.

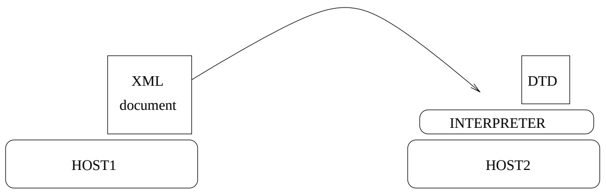

Figure 12: XML Program Migration to Remote Interpreter.

```
import java.rmi.*;
import java.io.*;

public interface Karel extends Remote {
 void execute(String program,
              String update_location)
      throws RemoteException,
             UnambiguousInsertException;
} // Karel
```

Figure 13: Remote Method Invocation for Karel.

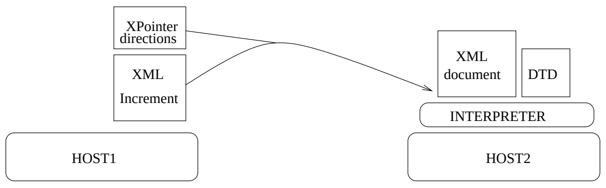

Figure 14: Increment Migration to Robot Site.

passed as the first parameter to `execute`. Figure 14 shows the migration of the increment code. The strategy for implementing incremental code mobility is then as follows: we first parse the program increment passed as the first parameter and construct an syntax tree for the increment, we evaluate the XPointer expression and replace the node addressed in the expression with the root node of the syntax tree of the increment. This replacement is shown in Figure 15.

In order to implement this strategy for incremental updates, we again take advantage of the DOM. Parsing the program increment and constructing the AST for it is achieved in the same way as for the full program. This time, the parser just creates a tree whose root node type is different from the root type of the DTD. In case of our Karel increment, a root increment node of type `times` is created.

The evaluation of the XPointer expression for the replacement node can be fully delegated to an XPointer processor. Again there are several of those processors available and we use the one that comes with XML4J. Figure 16 shows how we use the XPointer processor in order to locate the node that needs to be replaced. The replacement of the code increment is shown at the bottom of Figure 16. We navigate to the parent node of `replace` and use standard DOM operations to substitute it with the root node of the syntax tree of the increment that was sent.

6 EVALUATION

In this section we discuss the advantages and current disadvantages of the approach. We also hint at how the disadvantages may be overcome.

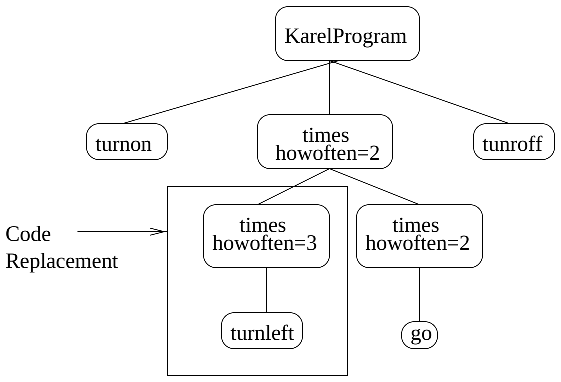

Figure 15: Result of Incremental Code Update on AST.

```
import org.w3c.dom.*;        // DOM API
import com.ibm.xml.parser.*; // IBM parser
import com.ibm.xml.xpointer.*;//IBM xpointer
...
public void execute(String program,
                    String update_location)
       throws RemoteException,
              AmbiguousInsertException {
 ...
 // create an XPointer object from
 // the update location that is passed
 XPointerParser xpp=new XPointerParser();
 XPointer xp=null;
 xp=xpp.parse(update_location);
 // Interpret XPointer object from the
 //root node of the previously parsed doc
 Pointed nodelist=xp.point(root);
 if (nodelist.size()!=1) {
  throw new AmbiguousInsertException();
 } else {
  Node replace=(nodelist.item(0)).node;
  Node parent=replace.getParentNode();
  //we get the parent node
  if (parent==null)
   throw new AmbiguousInsertException();
  //child replacement with the new code
  parent.replaceChild(
       inc.getDocumentElement(),replace);
 }
}
```

Figure 16: Evaluating XPointer Expression.

We have demonstrated how XML and its related technologies can be used for both specifying and implementing incremental code mobility at any granularity. By not fixing a particular granularity for mobile code, we enable complete programs as well as individual lines of code to be sent across the network. The combination of fine-grained and incremental mobility achieves a degree of flexibility previously unavailable. We have examined the application of incremental and fine-grained code mobility to user interface management, application management on PDAs and management of distributed documents.

The success of the approach critically depends on the ability to encode a high-level programming language in an XML DTD. Our Karel example has demonstrated that this is possible. The XwingML DTD suggests this can also be achieved in a scalable way. We can envisage that our approach will be used to write XML versions of interpreted languages, such as Javascript. We could then build a compiler that translates between Javascript and the XML encoding and an XML interpreter that wraps an existing Javascript interpreter.

In the Karel example, we have only shown how incrementality can be achieved by replacing existing fragments. We note that this may be overly restrictive. However, the strategies shown here can also be applied to add or delete pieces of code to or from the original program. To address insertion points or identify the fragments of deletion, we could use XPointer in the same way. To implement the changes to the abstract syntax tree, we could use the `insert` and `delete` operations of the DOM.

The interface shown in Figure 13 contains only one possible method (i.e., `execute`) to replace a fragment of code and execute the program. However, this is only a simplification of what is feasible with the approach; the interface can be extended with other methods, one for each operation that can be performed (i.e, replace, add, remove, execute), adding a higher level of flexibility to the ability of manipulating code (and data) of the remote entity. The primitives to implement all these operations are provided by both DOM and XPointer. XPointer enables us to address any node in the tree and DOM supports incremental insertion, removals and modification to nodes in the document's tree representation.

In the example presented in the paper we did not describe the combination of data and code mobility, in a step towards agent mobility. To achieve this in our Karel example, we could change the DTD of Karel's language and add an encoding for the position and other state attributes of Karel. In this way we can write an XML program containing Karel's position initialization. The interpreter would have to be modified as well in order to be able to obtain the information (i.e. the initial position), and to initialize Karel's status correctly.

We used RMI for implementing the migration of the XML program in the example. The approach, however, is independent of the transport protocol, as long as XML is used to encode the moving code. The advantages of using distributed object technologies, such as RMI, together with mark-up languages are reported in [11]. In our context any other transport protocol could have been used. Distributed object technologies add a significant overhead that is however balanced through the use of available middleware services such as transactions, and security. The choice and the trade-off evaluation of the transport technology is extremely application dependent, and beyond the scope of this paper.

The incremental update of the code is done after the robot has terminated an execution. However, in some applications it may be convenient to apply the changes to the program while the program is executing. The user interface application is a good example. This is feasible in our approach as well. Nevertheless, it would raise problems related to the maintenance of the program counter and the updating of operations in a cycle. However, if the language is simple enough this might be feasible.

Furthermore, incremental updating of code raises a series of issues related to access control problems: for instance, what happens if the code is updated twice by different principals? No one of the parties would know the actual status of the program. In our perspective we see applications in "code-distribution" oriented domains, where a single sender has full control of the code and has the right to update it. If we did not use RMI, but CORBA to transmit the code, the CORBA security service could be used to enforce these access rights.

7 SUMMARY AND FURTHER WORK

In this paper we presented an incremental approach to code mobility using the XML language. The novelty of the approach is the ability to send code incrementally instead of re-sending complete updated versions of the code. Java based technologies launched the idea of object and classes mobility, allowing a set of new paradigms for communication to become feasible.

Many theoretical languages have been used to specify and analyze code mobility [7, 22, 8, 12, 21]. The movement is specified with different granularities showing that the Java point of view, where a class is the unit of mobility, was not the only possibility to be explored.

In this paper we have shown a possible embodiment of these ideas, and described a set of potential application. We will now develop one of these applications in order to have a benchmark that we can use to evaluate the approach. In particular, we want to study performance issues and understand the trade-off between space and speed overhead compared to Java byte code transmission.

We are also interested in exploring the security implications of code migration and addressing them with the security services that object-middleware provides. By implementing interpreters as CORBA objects and using the access control interfaces of the CORBA Security service, we can guarantee that only authorized principals are performing changes to code.

In [9] *displets* are used to render special tags defined in XML using Java specific code for displaying formal notation on the Web. We see possible development of our work with the integration of this technique; DTDs and Java fragments could be sent together in order to update the ability of the interpreter to understand new constructs. We are also interested in providing support for proactive code mobility by adding specific XML tags like *go*, that are available in other mobile code languages [14]. They are interpreted as movement commands: this extension introduces many issues related with the dynamic modification of code. However, we believe this would extend the potential of the approach described in the paper considerably.

We intend to explore the use of this approach in real projects involving industrial partners in some of the domains that we mentioned in Section 4. We are currently collaborating with an industrial partner to develop a flexible user interface management for business analysis applications and intend to take advantage of XwingML in this application. Moreover, we are investigating the use of Symbian mobile phones and

PDAs as application platforms with an e-commerce provider. In this setting, we will explore the use of incremental code mobility for application management purposes.

ACKNOWLEDGEMENTS

We are indebted to G.P. Picco for his detailed comments and interesting suggestions on an earlier draft of this paper. We would like to thank P. Ciancarini, E. Ellmer, G.-C. Roman, and A. Zisman for the helpful discussion on the topic. We also acknowledge the support of the European Union through the RENOIR project.

REFERENCES

[1] IBM Alphaworks. MoDAL. http://www.alphaworks.ibm.com/tech/modal, 1999.

[2] IBM Alphaworks. XML4J. http://www.alphaworks.ibm.com/tech/xml4j, 1999.

[3] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium, October 1998.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[6] L. Cardelli. A language with distributed scope. In *Proc. 22nd ACM Symp. on Principles of Programming Languages (POPL)*, 1995.

[7] L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *LNCS*, Lisbon, Portugal, 1998. Springer.

[8] P. Ciancarini, F. Franzè, and C. Mascolo. Using a Coordination Language to Specify and Analyze Systems containing Mobile Components. *ACM Trans. on Software Engineering and Methodology*, page To appear, 2000.

[9] P. Ciancarini, F. Vitali, and C. Mascolo. Managing complex documents over the WWW: a case study for XML. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):629–238, July/August 1999.

[10] E. Ellmer, W. Emmerich, A. Finkelstein, D. Smolko, and A. Zisman. Consistency Management of Distributed Documents using XML and Related Technologies. Research Note 99-94, University College London, Dept. of Computer Science, 1999. Submitted for Publication.

[11] W. Emmerich, W. Schwarz, and A. Finkelstein. Markup Meets Middleware. In *Proc. of the 7th Int. Workshop on Future Trends in Distributed Systems, Capetown, South Africa*. IEEE Computer Society Press, 1999. To appear.

[12] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*. Springer, 1996.

[13] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, 1998.

[14] R.S. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., Dec. 1995.

[15] ISO 8879. Information processing – Text and Office Systems – Standardised General Markup Language SGML. Technical report, International Standards Organisation, 1986.

[16] JavaSoft. *Java Remote Method Invocation Specification*, revision 1.50, jdk 1.2 edition, October 1998.

[17] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–33, 1997.

[18] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[19] General Magic. *Telescript Language Reference*. General Magic, Oct. 1995.

[20] E. Maler and S. DeRose. XML Linking Language (XLink). Technical Report http://www.w3.org/TR/1998/WD-xlink-19980303, World Wide Web Consortium, March 1998.

[21] C. Mascolo, G.P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility . In O. Nierstrasz and M. Lemoine, editors, *Proc. 7th European Software Eng. Conf. (ESEC/FSE 99), Toulouse, France*, volume 1687 of *LNCS*, pages 39–56. Springer, 1999.

[22] P. J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2):97–110, 1998.

[23] Object Management Group. *XML Meta Data Interchange (XMI) – Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF)*. 492 Old Connecticut Path, Framingham, MA 01701, USA, October 1998.

[24] Object Management Group. *XML/Value Request for Proposals*. 492 Old Connecticut Path, Framingham, MA 01701, USA, August 1999.

[25] R. Pattis, J. Roberts, and M. Stehlik. *Karel the Robot*. Wiley, 1994.

[26] BlueStone Software. XML Contact. http://www.bluestone.com/xml/XML-Contact/, 1999.

[27] BlueStone Software. XwingML. http://www.bluestone.com/xml/XwingML/, 1999.

[28] J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Trans. on Programming Languages and Systems*, 12(4):537–565, Oct. 1990.

[29] J. Suzuki and Y. Yamamoto. Making UML models exchangeable over the Internet with XML: The UXF Approach. In P.-A. Muller and J. Bezivin, editors, *Proc. of Int. Workshop on UML '98, Mulhouse, France*, Lecture Notes in Computer Science. Springer, 1999. To appear.

[30] C.F. Tschudin. *An Introduction to the* `M0` *Messenger Language*. Univ. of Geneva, Switzerland, 1994.

[31] D. Wong, N. Paciorek, and D. Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, 1999.