Automated Software Engineering, 9, 151–165, 2002

XMILE: An XML Based Approach for Incremental Code Mobility and Update

CECILIA MASCOLO c.mascolo@cs.ucl.ac.uk
LUCA ZANOLIN* l.zanolin@cs.ucl.ac.uk
WOLFGANG EMMERICH w.emmerich@cs.ucl.ac.uk
Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK

Abstract. The eXtensible Markup Language (XML) was originally defined to represent Web content, but it is increasingly used to define languages, such as XPL, that are used for coding executable algorithms, policies or scripts. XML-related standards, such as XPath and the Document Object Model, permit the flexible manipulation of fragments of XML code, which enables novel code migration and update paradigms. The XMILE approach that we describe in this paper exploits these mechanisms in order to achieve flexible and fine-grained code updates, even without stopping execution. We describe a Java-based prototype that implements XMILE and our experience in using XMILE in the domain of code updates on mobile devices.

Keywords: code mobility, XML, code update

1. Introduction

Code mobility approaches (Fuggetta et al., 1998) introduce flexibility to the general programming paradigms, allowing code to be dynamically distributed across different hosts of a network. Mobile code can be pro-actively shipped using push-technologies or be fetched on demand, for example using Java's class loading mechanism. Mobile agents (Wong et al., 1999; University Stuttgart, 1999; Gray, 1995) permit the migration of autonomous programs from host to host, opening new possible communication strategies. In terms of physical mobility, communication and synchronisation of *mobile information devices*, such as personal digital assistants, palmtops, mobile phones and wearable computers, introduce new needs in terms of middleware and context awareness.

Due to the novelty of the arisen problems and the speed with which technologies for mobility are being developed, there is still uncertainty about both the level of flexibility and the paradigms for logical and physical mobility. Java-based technologies, for instance, Java Remote Method Invocation (Sun Microsystems, 1998) and Java Virtual Machines, such as those built into Web browsers, offer a logical mobility granularity at a *class level.* Mobile agents can be considered at the coarsest level of granularity achievable in a logical mobility context. The unit of mobility in this case is the agent itself, that is, an element containing code, data, and possibly some state.

Several application domains need a more flexible approach to code mobility than can be achieved with Java and mobile agents. This flexibility can either be required as a result

*This author was on study leave from Dipartimento di Ingegneria Elettronica, Politecnico di Milano, Milano, Italy.

of low network bandwidth, scalability or adaptability requirements. In physical mobility settings, for instance, 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, every time an update is needed. In case of mobile GPRS or UMTS networks, where the bandwidth is much larger, providers will charge by transmitted data volumes; also, in this case, the update of code of the services on the phones needs to be fine-grained to achieve cost effectiveness. Applications on several thousands of mobile information devices have to be kept in sync and be updated with new code fragments: scalability issues rise in this context, due to the number of applications and to the frequency of the updates.

It has been identified that mobile code is a design concept that is independent of technology and can be embodied in various ways (Fuggetta et al., 1998) in different technologies. The eXtensible Markup Language (XML) (Bray et al., 1998) 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 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 XML Schema (Fallside, 2000), XPath (Clark and DeRose, 1999) and the Document Object Model (DOM) (Apparao et al., 1998) to modify programs and even their programming languages dynamically. The availability of these XML technologies considerably simplifies the construction of application-specific languages and their interpreters.

In this paper we describe extensions of our previous work on XML-based code mobility. In Emmerich et al. (2000) we explain how we use XML to achieve more fine-grained mobility than in the approaches that are based on Java and we demonstrate that the unit of mobility can be decomposed from an agent or class level, if necessary, to individual statements. The paper (Emmerich et al., 2000) also describes how we 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) and mobile phones, and active networking.

This paper describes how we extended and evaluated our previous work. In particular, we now support incremental updates to code without the need for stopping any execution, which supports changing applications that need to be active at all times. Moreover, we now no longer use XML Document Type Definitions to define XML programming languages but, use XML Schema instead. This means that the definition of the language grammar is itself contained in an XML document and therefore amenable to the same code migration and updates. This allows us to also change the syntax of an XML programming language. Such changes demand subsequent changes to the language interpreter. We explain how successive updates to interpreters of XML programming languages can be achieved by structuring interpreters according to the syntax and then use Java class loading primitives to update the interpreter code. We have applied the extended XMILE in a number of case studies, some of which in collaboration with the local software industry, and we report on our experience to date.

XMILE is built on the formal foundation for fine-grained code mobility that was established in Mascolo et al. (1999). 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. The paper underlines the need of flexible methods for analysis and description of what needs to be mobile, so that designers can decide what to move depending on the applications they are targeting. In this paper, we share that vision and focus on an implementation of fine-grained mobility using standardized and widely available technology.

This paper is further structured as follows. In Section 2 we describe the XMILE approach to code mobility, the definition of domain specific languages, and the incremental mobility approach. In Section 3, we outline the implementation of the XMILE system. In Section 4, we discuss how we use the XMILE approach in applications of different domains, in particular, mobile phone services update, programmable networks and thin clients application service provision. We describe in detail the first application scenario. Section 5 evaluates the approach and identifies strengths and weaknesses comparing with other existing approaches. Section 6 contains a summary of the work done and some future work.

2. The XMILE approach

XML provides a flexible approach to describe data and document structures. We now show how XML can be used to describe code and explain how XML can be used to define *programs* and how specific control flow tags can be defined and used in XML documents.

XML Schemas define the grammar for XML programs. The structure of all the element that can be put in an XML program are defined in a schema. XML Schemas (or DTDs (Bray et al., 1998), Data Type Definition which are the predecessors of Schemas) are very similar to attribute grammars (Knuth, 1968). Each element in an XML Schema corresponds to a production of a grammar. The complex type of an element defines the right-hand side of the production. Contents can be declared as enumerations of further elements, element sequences (i.e., `<xsd:sequence>`) or element alternatives (i.e., `<xsd:choices>`). These give the same expressive power to Schemas as BNFs have for context free grammars. Elements of XML Schemas can be attributed. These attributes can be used to store the value of identifiers, constants or static semantic information, such as symbol tables and static types. Thus, XML Schemas can be used to define the abstract syntax of programming languages. We refer to documents that are instances of such Schemas as XML *programs*. XML programs can be interpreted and in Section 3 we discuss how such interpreters can be constructed using XML technologies. By sending XML programs or fragments of them from host to host we achieve a very fine granularity for the unit of code mobility.

We use a simple example to demonstrate this idea. Consider a simple calculator application installed on a mobile information device. The initial calculator installation is able to add numbers and display the result. Figure 1 shows the XML program implementing the calculator functions. The program is written in an XML programming language whose syntax is defined by a Schema, an excerpt of which is shown in figure 2. The program starts with the definition of the program name in element `properties` (line 3). The program itself begins with the element `program` (line 4) and it then defines variables in order to store the number in the display, the number in memory and the value of the button pressed. The element at lines 9–26 implements a *while* loop. As will become clear later, the `block`

```
<?xml version="1.0"?>
<general>
 <properties name="calculator"/>
 <program>
    <var name="mem" value=""/>   <!-- number shown in the calculator memory -->
    <var name="cur" value=""/>   <!-- number stored in the calculator display -->
    <var name="button" value=""/><!-- button pressed -->
    <var name="loop" value="1"/>
    <While var="loop" value="1">
      <block>
        <GetInput var="button"/>
      </block>
      <block>
       <CaseOf var="button">
       <case ="plus">
          <Add firstNumber="mem" secondNumber="cur" saveIn="cur"/>
         </case>
         <case ="enter">
          <Enter number="cur" saveIn="mem"/>
         </case>
         <default>
          <AppendDigit digit="button" number="cur" saveIn="cur"/>
         </default>
         </CaseOf>
      </block>
    </While>
  </program>
</general>
```

Figure 1. XML code for the calculator.

element is used for defining control flow and supporting updates in non-reboot conditions. For simplicity we omit the user interface definition code and just show the element `GetInput`, which handles the user input on the calculator. Element `CaseOf` manages the different inputs: either a digit, or the plus or the enter buttons are pressed. Specific elements for each of these events are used. For instance, the `Add` element has three attributes (i.e., parameters), namely the two numbers to be added and the name of the variable where to store the result.

The root of the grammar in figure 2 is the definition of the `general` element, which contains the `properties` (which we do not show) and the `program` element. The `program` element (line 2) can contain a list of choices of other elements. The grammar showed in figure 2 is a simplified version of the actual one. We just show some of the elements we used in figure 1, namely the `While`, `var`, `block` and `Add` tags. `While` (line 16) can contain all the possible tags (a part from `var`) and it has two attributes, namely a counter for the loop and an actual variable. The `var` tag (line 26) defines a variable with a name and a value. The `block` tag (line 33) is used to structure the program in order to be able to have updates in non-reboot conditions. Blocks essentially are monitors for concurrent access to the code and they may act as scopes for variables. Blocks can contain all the possible elements.

The `Add` tag (line 43) has three attributes which are the first and second numbers to add and the resulting number.

```
1  <!-- The element program contains tags such as While, var, block, add and so on -->
2    <xsd:element name="program">
3     <xsd:complexType>
4      <xsd:choice minOccurs="0" maxOccurs="unbounded">
5       <xsd:element ref="While"/>
6       <xsd:element ref="var"/>
7       <xsd:element ref="block"/>
8       <xsd:element ref="Add"/> ...
9      </xsd:choice>
10    </xsd:complexType>
11   </xsd:element>
12   </xsd:complexType>
13  </xsd:element>
14 <!-- The element While has 2 attributes (lines 18 and 19), var and value for the counter,
15 and may contain the elements defined on line 21 -->
16  <xsd:element name="While">
17   <xsd:complexType content="elementOnly">
18    <xsd:attribute name="var" use="required"/>
19    <xsd:attribute name="value" use="required"/>
20    <xsd:choice minOccurs="0" maxOccurs="unbounded">
21     <xsd:element ref="While"/> <xsd:element ref="block"/>...
22    </xsd:choice>
23   </xsd:complexType>
24  </xsd:element>
25 <!-- A variable has name and value attributes (lines 28 and 29) -->
26  <xsd:element name="var">
27   <xsd:complexType>
28    <xsd:attribute name="name" use="required"/>
29    <xsd:attribute name="value" use="required"/>
30   </xsd:complexType>
31  </xsd:element>
32 <!-- block has the same structure as While -->
33 <xsd:element name="block">
34  <xsd:complexType>
35   <xsd:attribute name="name"/>
36   <xsd:choice minOccurs="0" maxOccurs="unbounded">
37 ... <!-- Same choices as in While -->
38   </xsd:choice>
39  </xsd:complexType>
40 </xsd:element>
41 <!-- Add has 3 attributes, the first two for the numbers to be added
42 and the thrid to store the result -->
43 <xsd:element name="Add">
44  <xsd:complexType>
45   <xsd:attribute name="firstNumber" use="required"/>
46   <xsd:attribute name="secondNumber" use="required"/>
47   <xsd:attribute name="saveIn" use="required"/>
48  </xsd:complexType>
49 </xsd:element>
50 ...
51</xsd:schema>
```

Figure 2. XML schema for the calculator.

Note that an XML schema is an XML file itself, and we show below how we exploit this property for dynamic updates of language grammars.[1]

The calculator application is running on a mobile information device. The XMILE approach allows us to update the code by transferring fragments of new code from a server to the phone and then dynamically patching the original code. Unlike Java programs, which are sent in a compiled form, XML programs are transferred in source form and then interpreted on the 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 grammar, 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. The XML increments will be shipped together with some information on how to modify the remote XML program. For instance, let us assume that we want to update the calculator adding the ability of subtracting numbers. For this we need to add some code to the calculator. Figure 3 shows the program increment that is shipped to the mobile phone to update the application.

The program increment that we need to update can be sent separately without the need to re-send the complete program. As we will describe in detail in Section 4, this is required in situations where slow network connections, or pay by volume charges are involved.

The XML program structure makes the dynamic manipulation of the code a lot easier. An XML program can be seen as a tree and the DOM (Apparao et al., 1998) API provides operations for the navigation and modification (adding/deleting/changing) of branches of the program tree. The addressing of the particular branch that needs to be modified is performed using the XPath language (Clark and DeRose, 1999). Going back to our example, figure 4

```
<case value="minus">
  <Minus firstNumber="mem" secondNumber="cur" saveIn="cur"/>
</case>
```

Figure 3. XML program increment.

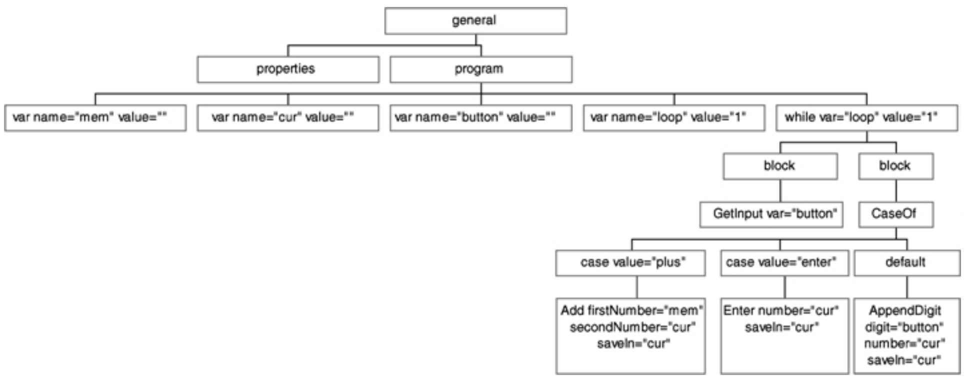
general
properties
program
var name="mem" value=""
var name="cur" value=""
var name="button" value=""
var name="loop" value="1"
while var="loop" value="1"
block
block
GetInput var="button"
CaseOf
case value="plus"
case value="enter"
default
Add firstNumber="mem" secondNumber="cur" saveIn="cur"
Enter number="cur" saveIn="cur"
AppendDigit digit="button" number="cur" saveIn="cur"

Figure 4. The calculator program tree.

```
xpath="/general/program/while/block[2]/CaseOf
```

Figure 5. The XPath expression for the addressing of the insertion point.

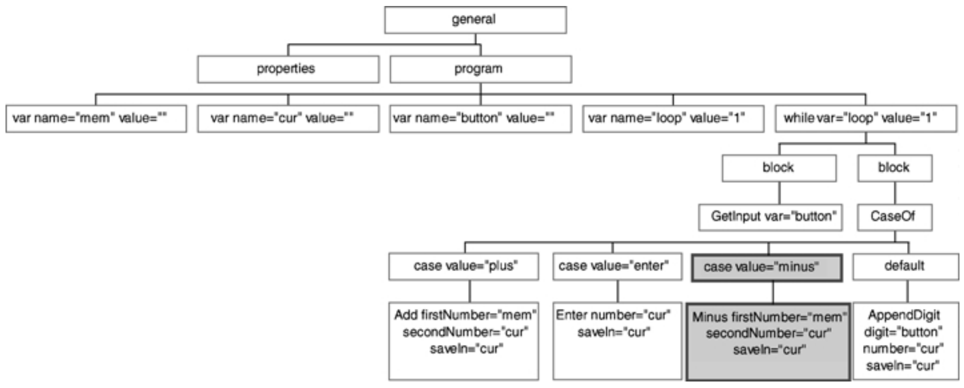
general
properties
program
var name="mem" value=""
var name="cur" value=""
var name="button" value=""
var name="loop" value="1"
while var="loop" value="1"
block
block
GetInput var="button"
CaseOf
case value="plus"
case value="enter"
case value="minus"
default
Add firstNumber="mem" secondNumber="cur" saveIn="cur"
Enter number="cur" saveIn="cur"
Minus firstNumber="mem" secondNumber="cur" saveIn="cur"
AppendDigit digit="button" number="cur" saveIn="cur"

Figure 6. The new DOM tree after the code update.

shows the DOM representation of the program in figure 1, while figure 5 shows the XPath expression addressing the point where the new increment (the minus operator) needs to be added in the program.

Figure 6 shows the new tree structure after the migration and insertion of the program increment for the minus operation into the calculator program.

Given the fact that most of the applications we are targeting (on mobile devices and in network settings) need to be updated at run-time, XMILE was conceived with the requirement that code updates need to be performed on-the-fly. We exploit the DOM tree structure to determine when and how the updates and the code interpretation may be interleaved. For example, in the calculator example, the user is not aware of the code migration and update until the new "minus" button pops up on his/her window. The user is allowed to perform operations during the code update. As we will show, the XMILE engine handles the concurrency of events defining "blocks" of code which act as *monitors*.

As described earlier, XML Schemas are themselves XML files. This allows us to dynamically modify the grammar of an XML programming language. In the above example, we did not have to change the grammar and assumed that our XML programming language already has a `Minus` operator that we then used in the XML code increment. If we, however, want to modify the language, for example by adding a "multiplication" operator so that in a subsequent update we can also include a multiplication button to the program, we could transmit the schema update shown in figure 7 with a specific XPath expression to define the update point.[2] To implement the richer semantics of the language, we then need to also evolve the interpreter. We rely on Java code mobility and class loading of new interpreter classes for that purpose and discuss that in the next section.

```
<xsd:element name="Multiply">
 <xsd:complexType>
  <xsd:attribute name="firstNumber" use="required"/>
  <xsd:attribute name="secondNumber" use="required"/>
  <xsd:attribute name="saveIn" use="required"/>
 </xsd:complexType>
</xsd:element>
```

Figure 7. XML Schema update code.

3. Implementation of the approach

In this section we outline the architecture and the implementation of our approach. Every XMILE enabled machine runs Java and our XMILE engine. The XMILE engine is able to parse and interpret XML programs such as the one in figure 1 for the calculator. The engine is able to perform the concurrency control of updates to XML programs that are being executed, so as to allow dynamic modification of XML programs through other XML programs. Figure 8 describes the architecture of two XMILE hosts and their interaction procedure.

XML programs with different purposes can be built and run on the same XMILE engine. The XMILE engine spawns an interpreter for each XML program that a host receives (for example for the calculator program of figure 1). As we mentioned earlier, the XML programs are structured using *blocks*, which act as monitors for concurrent accesses, exploiting the XML program tree structure. This means that whenever an update to any statement contained in a block is not possible whenever an interpreter executes statements that are contained in the block. This permits the dynamic modification of programs through other programs at

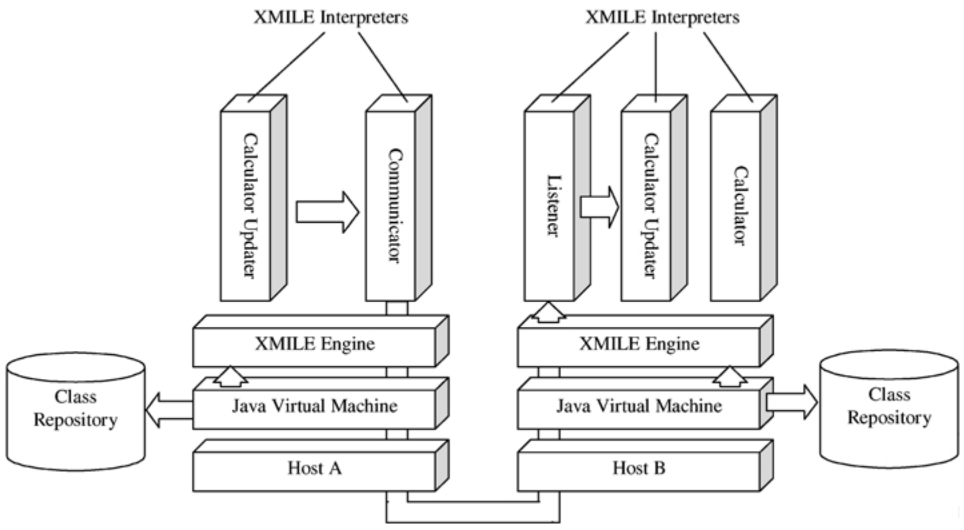

Figure 8. The architecture of two XMILE hosts and their interaction.

```
<?xml version="1.0"?>
<general>
        <properties name="comm"/>
         <program>
                <var name="savedPrg" value=""/>
                <var name="loop" value="always"/>
                <CommLoop var="loop" value="always" port="1972" saveIn="savedPrg">
                        <Create program="savedPrg"/>
                </CommLoop>
         </program>
</general>
```

Figure 9. XML code for the listener.

run-time. Programmers can optimize concurrency control adding blocks (using the primitive `<block> </block>`. Hierarchical locking is used to guarantee consistency in the locking.

We apply the same mechanism that is used for executing XMILE programs for updating these programs, which means that the program updates are sent and received by XML programs, too. We refer to the sending program as a communicator and to the receiving program as a listener. The XMILE engine contains built-in interpreters for communicator and listener programs. They use TCP/IP through a socket interface for their communication between each other, however any other middleware could be used (in a previous prototype we used Java RMI). An example of a listener program is shown in figure 9.

The `CommLoop` loop in the program makes the program wait on a specific port for some input (the Java class for the tag implements sockets communication) and when received it calls the executor of the root class for the program received through the tag `Create`.

We used Sun JDK1.3, and the Apache Xerces (for XML parsing and DOM API level 2) and Xalan (for XPath processing). Let us now describe more in detail the interpretation of an XML program.

3.1. Interpreter

Whenever a program is received on a host a XMILE Interpreter is spawned and begins to parse the program. XMILE Interpreters are structured in such a way that there is a class for every element type of the XML language and that class has an `execute` method, which is called when control is given to an element of that type. Revisiting our calculator example of figure 1, when element `<While ...>` is found the class `While.class` is loaded into the interpreter and its `execute` method is called. The interpretation and execution of the XML program follows the syntax tree structure and the DOM API is used to traverse the child elements of a particular node and to read attribute values. Whenever a Java class is not present locally *dynamic class loading* is used to retrieve it remotely. The tag syntax can specify the host from where the class loader should fetch the class. An example of this is in figure 10: the tag `Minus` specifies that if the class is not found locally it can be loaded from `host="sushi.cs.ucl.ac.uk"`.[3] Figure 8 shows the class loading from the local repository and the remote fetching of classes.

```
<general>
 <properties name="update"/>
 <update>
  <case value="minus">
   <Minus host="sushi.cs.ucl.ac.uk" firstNumber="mem" secondNumber="cur" saveIn="cur"/>
  </case>
 </update>
 <program>
  <InsertChild nameProgram="calculator"
xpath="/general/program/while/block[2]/CaseOf"/>
 </program>
</general>
```

Figure 10. An XML program for updating the calculator.

3.2. Code update

A program for the update of other programs can be written and shipped to a host as a normal XML program. When received an interpreter is spawned as usual. As described in Section 2, an XML program fragment and an XPath expression can be migrated and these are sufficient to specify how to modify a program. In order to standardize the way we update programs we write the XML code update and the XPath expression in a single program that is shipped to the remote host. An example of an update program is the one in figure 10.

The program contains two main parts. In between the two `update` tags the update code (i.e., the "Minus" already shown in Section 2) is contained. The program itself is contained in the `program` element and it instructs the interpreter to insert a child into the program `calculator`, under the indicated XPath expression. This approach allows us to specify more complex program updates, such as duplication or movement of already existing lines of code, removal, insertion in different positions. Once again, the *blocks* defined in the code act as monitor for concurrency control of the updates and the interpretation. Several updates can be performed by different programs at the same time. This allows us to handle code modification on-the-fly. Also a caching system for the block to be updated is also implemented for performance reasons.

4. Evaluation

XMILE has been applied in a variety of settings, such as code update on mobile devices, programmable networks and reconfiguration of application servers. In each of these settings flexibility and changeability were quite important. We now describe an application to the management of user interfaces of mobile phones in more detail. An account of a XMILE application to programmable network (Tennenhouse and Wetherall, 1996) is described in Meer et al. (2001) and Mascolo et al. (2001) and we refer the interested reader to those references for more details. A XMILE application to the reconfiguration of application servers is described in Zanolin (2001).

The calculator that we used throughout this paper is an example of a simple application that might be deployed on mobile phones or PDAs. Mobile phone operators might want to be able to update these applications. To date phone operators, for example, deploy about

20 different user interfaces on each handset to cater for different languages. If the user interface were described in an XML language, operators could use XMILE to download a new user interface whenever the language preferences of the user changes and in that way safe significant amounts of memory. The ability to flexibly update the behaviour of a large number of phones might also appeal when it becomes necessary to fix software faults without having to recall a large number of handsets, such as recently experienced by Nokia or NTT DoCoMo.

We implemented the example of the calculator shown in Section 2 on the Symbian mobile phone operating system (Tasker, 2000), which includes a Java Virtual Machine. In order to evaluate the performance, we installed XMILE on an Ericsson MC 218, which has 16 MB of memory and a 37 MHz ARM processor. We now report on the performance of XMILE on this phone with respect to memory consumption, performance of the application and performance and costs of updates.

The Java Virtual Machine on the MC 218 requires about 2 MB of main memory. XML parsers, the XPath processor, the DOM tree of the calculator and the AWT user interface consume a further 1 MB bringing the total memory used to about 3 MB of main memory. Figure 11(a) shows the calculator installed on the phone.

In order to evaluate the run-time performance of the XMILE engine, we compare the XMILE calculator with a functionally equivalent application that is written and compiled into byte code using Java 1.1.6. To start the Java byte-code calculator takes about 6.3 seconds, while the launch of the XMILE calculator takes about 14.6 seconds. This total time breaks down into initialization of the XMILE engine (4.8 seconds), start of the calculator (2.6 seconds) and 7.2 seconds to display the AWT user interface. Thus, the initialization of the XMILE calculator takes a little more than twice as long as the startup of the byte code calculator. We note that booting of the XMILE engine only needs to be done once (e.g., when the phone is switched on) and then it can execute a number of applications and therefore the results are comparable. Once the startup is completed, the delays incurred when using the application to perform calculations are not noticeable neither in the byte code nor the XMILE calculator application because the operations performed are not computationally intensive.

Figure 11(b) shows the new user interface after the code update has happened. With respect to the example we used in Section 2 the calculator program for this application also handles the user interface update for the "minus" button. The total size of the XML program

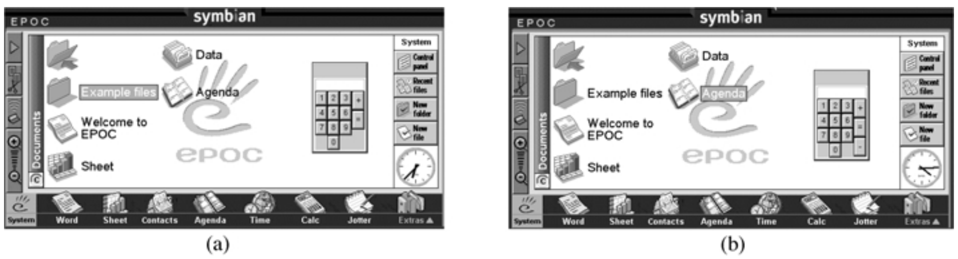

Figure 11. (a) Calculator on Symbian phone (b) Updated calculator.

update for both the functionality and the additional button that had to be transmitted to the phone in this example is 519 bytes. The updates can be sent to the phone either using IRDA or using a network protocol for example over GSM. Transmitting the above code using a 9,600 baud GSM link takes about 500 milliseconds when reception is good and the update time drops to about 100 milliseconds with GPRS. Thus, the cost at the time of writing this paper for a GSM transmission would be 0.083 pence (5 pence per minute connection) and for GPRS (1 pound sterling per Megabyte transmitted) it would amount to 0.05 pence. When changing the Java byte code calculator, we would assume that the Java Archive (jar) that includes the calculator code needs to be sent to the phone and for adding the same functionality this amounts to a size of about 57,000 bytes, which is two orders of magnitude larger than updating the XMILE calculator. To perform this update over a GSM connection would take about 60 seconds and would incur a cost in the UK of about 5 pence and over a GPRS connection it would take about 10 seconds and incur a cost of 5.5 pence.

Therefore, the use of XMILE is beneficial and cost effective whenever applications do not perform any computationally intensive tasks (as is often the case in front-end components of distributed systems), there is a need to update the application and the data links between sender and receiver are either slow or expensive.

5. Discussion and related work

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

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.

Mobile agents (Wong et al., 1999; University Stuttgart, 1999; Gray, 1995; Johansen et al., 1995) provide mobility at the agent level with some finer degree of mobility when a Java class is fetched using Java class loading after an agent moves, or when a Tacoma folder contains code which is patched into another agent. Other mobile code based approaches, particularly μ-CODE (Picco, 1998), allow the designer to decide on the granularity of the mobility depending on the application. However, for the approaches based on Java, the finest granularity of the unit of mobility is at the class level. Other approaches not Java based have also been developed (Levy et al., 1988; Tschudin, 1994; White, 1996; Cardelli, 1995). However, in general, the focus has not been on trying to provide a flexible granularity for the unit of mobility, but instead, by fixing a specific granularity in providing a specific paradigm, hiding details from the application layer. The XMILE approach is novel in that it supports both Java class loading of interpreter classes but also incremental code mobility of individual interpreted statements.

We have examined the application of incremental and fine-grained code mobility to application management on mobile devices, programmable network settings, and mobile devices application service provision and we found strong evidence that they are both needed and sufficient.

The calculator example has shown how to migrate and add a code fragment to the program and how to expand the grammar. However, code update and deletion are also possible using update programs like the ones shown in Section 3 (specific tags for different operations you would want to perform have been defined in XMILE). XMILE allows the modification of code at three different levels. First, XML tags can be added/modified/deleted from the XML program. We can imagine that Java classes corresponding to tags are implemented so that users just have to write an XML program or program increment. However, in some infrequent cases we still can imagine to have another level of mobility driven by the Java class loader on each XMILE enabled machine. In the rare event in which a Java class corresponding to an element needs to be modified, the class loading mechanism can be instructed to download the new class from a remote repository. Finally, XMILE supports the modification of the grammar of our domain specific language as we showed in Section 2.

Moreover, XML files can be validated against a Schema during parsing. This operation can be useful when we need to ensure that the used tags respect certain conditions and a particular syntax. The validation offers some levels of integrity checks, which could be strengthened by additional also defining static semantic checks, such as adding types to variables. The validation is optional and can be turned off if not needed. However, even with validation, security is an issue when dealing with code modification and mobility and we plan to analyze this issue further, especially in the context of programmable networks. Some more security could be obtained constraining the communication sockets to be open only at specific times.

As we briefly mentioned in Section 3, XML programs that send XML programs to other hosts can be easily written. It is also possible to use XPath to address specific parts of the same program in order to ship them. These two features combined support *pro-active mobility* (in the same way as mobile agents do), or with the fine granularity, to mobility of parts of the same programs to other hosts. We have not investigated in this direction further, however we believe this is an interesting capability that underlines the flexibility of XMILE.

One of the novelties of XMILE is the ability to change code, while the code is being executed. This can be done by exploiting the tree structure of the programs and the DOM API. Specific elements are specified in XMILE to define blocks in the XML programs. The blocks are of monitors which synchronize concurrent access by the interpreter and the updater to guarantee consistency. As the blocks need to be defined by the programmer, the programmer is also able to define more efficient and less efficient programs (in the updating perspective) as with larger blocks are updates of elements contained in that block can only happen less frequently.

As we said, for every application, a domain specific language must be defined in XML. For every new tag a corresponding Java class needs to be implemented. XMILE comes with a set of common already defined tags. In our experience, only a small number of new tags need to be defined from scratch for every application. The definition of the Java classes for these tags is usually quite easy. XPL (VBXML, 2000) is an XML based extensible programming language. The aim of it is to be able to allow programming, mainly scripting, through a set of XML tags. XMILE follows the same line, however the motivations of XPL are still very vague. XMILE aims at a fine-grain way to update remote programs, possibly defined with a specific extensible grammar linked to a domain, versus th XPL vision of a generally defined grammar with which to program.

6. Conclusion and future work

We have described an XML based system for fine-grained code migration and update. We have illustrated the main ideas behind the approach and the implementation details of XMILE. We have shown how we are using XMILE in different application domains, such as mobile devices application update, programmable networks, and mobile devices application service provision. We plan to continue to apply XMILE in different context to prove its flexibility. We are defining a policy XML based language and we want to use XMILE to distribute and update different kind of policies such as security policies.

Technologies, such as XML RPC (2001), SOAP (Box et al., 2000) or JAXM (2001) could potentially be integrated with our approach in order to achieve code mobility in more flexible ways: we plan to investigate this research direction.

From a logical mobility point of view we are interested in extending our variable binding mechanism allowing dynamic downloading of non present variable in addition to the dynamic local binding.

The XMILE web site can be found at `http://pizza.cs.ucl.ac.uk/xmile`.

Acknowledgments

We would like to thank Andrew Bud, Anthony Finkelstein and Richard Gold for participating in the discussion and in the application of XMILE. We thank Jon Crowcroft and Hermann de Meer of UCL network research group for their support using XMILE for programmable networking. We also gratefully acknowledge the support from MBlox Ltd.

Notes

1. The actual grammar used in the application contains some tags grouping which we do not show here that help in the modularization of the grammar update process.
2. The actual grammar update to add a new command also needs to update the list of commands in every other command. As we said the schema in figure 2 is a simplification of the actual one we use. For convenience, we grouped the possible commands in subsets in order to simplify the update process. Also, as will become clear in Section 3, the double update is performed using another XML program, which sequentially executes the two operation on the Schema.
3. The `host` is an optional attribute of the elements. For reasons of brevity the grammar in figure 2 did not show this.

References

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

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S., and Winer, D. 2000. Simple object access protocol (SOAP). Technical Report http://www.w3.org/TR/SOAP/, World Wide Web Consortium.

Bray, T., Paoli, J., and Sperberg-McQueen, C.M. 1998. Extensible markup language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium.

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

Clark, J. and DeRose, S. 1999. XML path language (XPath). Technical Report http://www.w3.org/TR/xpath, World Wide Web Consortium.

Emmerich, W., Mascolo, C., and Finkelstein, A. 2000. Implementing incremental code migration with XML. In M. Jazayeri and A. Wolf, editors, *Proc. 22nd Int. Conf. on Software Engineering (ICSE2000)*. Limerick, Ireland, June 2000, New York: ACM Press, pp. 397–406.

Fallside, D.C. 2000. XML schema. Technical Report http://www.w3.org/TR/xmlschema-0/, World Wide Web Consortium.

Fuggetta, A., Picco, G.P., and Vigna, G. 1998. Understanding code mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361.

Gray, R. 1995. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*.

Java API for XML Messaging (JAXM). 2001. Available at http://java.sun.com/xml/xml_jaxm.html.

Johansen, D., van Renesse, R., and Schneider, F. 1995. Introduction to the TACOMA distributed system. Technical Report 95-23, Department of Computer Science, University of Troms, Norway.

Knuth, D.E. 1968. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145.

Levy, J., Hutchinson, H., and Moore, D. 1988. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

Mascolo, C., Emmerich, W., and De Meer, H. 2001. An XML based programmable network platform. In *ICSE Workshop on Software Engineering and Mobility*, May 2001.

Mascolo, C., Picco, G.P., and Roman, G.-C. 1999. A fine-grained model for code mobility. In O. Nierstrasz and M. Lemoine, editors, *Proc. 7th European Software Eng. Conf. (ESEC/FSE 99)*, Vol. 1687 of LNCS, Berlin: Springer, pp. 39–56.

De Meer, H., Emmerich, W., Mascolo, C., Pezzi, N., Rio, M., and Zanolin, L. 2001. Middleware and management support for programmable QoS-network architectures. In *Short Papers Session of 3rd Int. Workings Conference on Achve Networks (IWAN01)*, October 2001.

Picco, G.P. 1998. μCODE: A lightweight and flexible mobile code toolkit. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, Vol. 1477 of LNCS, Berlin: Springer, pp. 160–171.

Sun Microsystems. 1998. *Java Remote Method Invocation Specification*, Revision 1.50, JDK 1.2 edition.

Tasker, M. 2000. *Professional Symbian Programming*. Wrox.

Tennenhouse, D.L. and Wetherall, D.J. 1996. Towards an active network architecture. *Computer Communication Reviews*, 26(2).

Tschudin, C.F. 1994. *An Introduction to the* MO *Messenger Language*. University of Geneva, Switzerland.

University Stuttgart. 1999. Mobile agent list. Available at http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html.

VBXML. 2000. XPL: The eXtensible programming language. Available at http://http://www.vbxml.com/xpl.

White, J. 1996. Telescript technology: Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press.

Wong, D., Paciorek, N., and Moore, D. 1999. Java-based mobile agents. *Communications of the ACM*, 42(3): 92–102.

Xml-rpc. 2001. Available at http://www.xml-rpc.com/

Zanolin, L. 2001. An XML-based middleware for fine-grained code mobility and update. Technical Report, Politecnico di Milano. Tesi di Laurea.