A Java-Based Programming Language Support of Location Management in Pervasive Systems

Sherif G. Aly
The American University in Cairo

Sarah Nadi
The American University in Cairo

Karim Hamdan
The American University in Cairo

Summary
In this paper, we present an effort towards incorporating programming language support for pervasive systems development activities through extending on an already existing and stable programming language, namely Java. Pervasive systems research indicated the need for five main cornerstone features in support for conducting efficient pervasive systems development activities, namely programming language support for context, location, actors, sensors, and events. Herewith this article, we illustrate how we utilized the extensible markup language (XML), along with the Jini Javaspace technology, access rights definition, as well as supporting libraries to incorporate the representation and management of location in pervasive systems development activities. Beyond presenting the details of this approach, we demonstrate an actual example showing how location topology can be defined, along with their associated fine-grain access rights, how such topology can be stored and retrieved from Javaspaces, how the Javaspaces themselves can be managed, and how to query for location information taking into consideration the enforcement of defined rules for access rights to various locations.part of summary.

Key words:
Pervasive, location, Javaspace, XML, access.

1. Introduction

With the rapid spread of pervasive computing research, computational devices have actually started to interweave themselves into our daily lives, and are affecting the daily lives of many people. Pervasive systems in the form of intelligent environments rich in sensing and computational ability are adapting and reacting to the daily lives of many. It is Mark Weiser, the founder of pervasive computing, who actually anticipated this phenomena back in 1991 [1], and only time proves the correctness of such prediction.

The development of various infrastructures, languages, tools and environments that support the development of pervasive systems must however be facilitated to the greatest possible extent. Such support must be made available in such way that may allow for the further advancement of this field.

Unfortunately, a programming language with solid features that can cater to the direct needs of pervasive systems does not still exist. Much pervasive systems research indicated the need for five essential features that should exist in any programming language that will be used in the development activity of pervasive systems. Such features include the support for: Context, location, actors, sensors and events. Many research efforts have tackled one or more of these needs separately such as that presented by Heutelbeck in [2] which suggests a data structure for location information in the form of tuples. The work proposed in [3] also states the requirements of a programming language supporting context, and the work in [4] provides a high-level programming language for prototyping pervasive applications and also focuses on modeling context information and events. Finally in [5], location management in pervasive systems is discussed.

Despite these research efforts, and many others, that have produced valuable contributions to the pervasive computing domain, a programming language that encapsulates all such features together is still missing. It is thus necessary to provide a programming language that integrates all required features supporting the development of pervasive systems. However, designing a completely new programming language is not necessary if existing solid and widespread programming languages can be extended in a streamlined way to support the programming of pervasive systems. A programming language such as Java, one of the most widely used programming languages in the world nowadays, already has basic and abstracted features such as architecture independence, networking, and multithreading abilities that make it a very appealing candidate for the support of pervasive systems and promote it as a solid baseline for adding new data structures and features in support of pervasive systems.

The advantage of choosing Java as well includes the extensibility features that the language offers. Java offers

Manuscript received June 5, 2008.
Manuscript revised June 20, 2008.

a considerable set of libraries and technologies that can be extended to easily incorporate more capabilities into the language. For example, Jini, a set of Java networking APIs, could be considered a very appropriate networking technology that may be used to support pervasive systems [6]. More details about how Jini is incorporated in this research will be given in Section 3.

Out of the five necessary aforementioned features that need to exist in pervasive programming languages, this article focuses on specifying appropriate data structure support for location that meets the needs of pervasive applications. It is worthy to mention that the support for location identification, management and storage is an essential requirement for the incorporation of context awareness and reactivity in pervasive systems. Location information is typically used as input to deducing context, and as information for the actors themselves.

As indicated in [12], there are three major issues that must be considered when dealing with context, namely: Places, people, and things. The various attributes that may describe these issues include: Identity, location, status and time. Location is thus one of the key corner stores of information used by context-aware systems.

Considering the following example that illustrates the need for location support: A user with a pre-scheduled meeting; their personal digital assistant (PDA) automatically alerts them with an appointment ahead of time, and provides the user with directions to an indicated location of the meeting room. In order to provide the needed directions in such a scenario, the location topology of the area must be stored in an easily accessible location, and in a way that considers the distributed nature of pervasive systems. Eventually, an application that maps a route to the indicated location can use this information while visualizing the route to the user.

Another scenario relates to deducing context information. If a fire emergency occurred in some location, a relevant pervasive application may need to deduce the actors who are in danger of getting hurt, and then the relationship of the locations in that area must be known. The application may need to know if the user in a certain room is in the same floor as that on which the fire occurred for example. All such information can be deduced with the support of a data structure that can contain location information and the relationships between locations in an environment. Thus, to develop a pervasive system, this information must be present in some data structure that can be used by all participating entities.

In our support for location, and as we will illustrate later in the article, we used XML to represent the location topology of a pervasive system, and provided appropriate supporting libraries for manipulating and interpreting such location topology. XML was chosen for creating the location topology data structure support. As stated in [4], “It provides a more open, flexible and scalable solution for a common location representation format in pervasive systems”. Furthermore, as stated in [7], one of the problems with current approaches to developing pervasive systems is the lack of separation between data and functionality. It is highly recommended to have location information data separate from the functionalities that will use it. Representing location information using XML is thus a very appealing option.

Moreover, Jini was used in particular for the presence of its JavaSpaces services to support the dynamic and distributed nature of pervasive systems [8], and make the location topology information available to all actors of the system.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 describes our approach in detail. Section 4 gives an example on how to use the different classes that we created. Section 5 has the references that we used in order to complete our work.

2. Related Work

The distributed nature of pervasive systems enforces some requirements on the data structures used to develop those systems. These requirements include supplying the data structures with features which support the various needs of the different entities collaborating in this distributed environment. Location management is one of the main critical functionalities in any pervasive system. The research done in [2] suggests that the location data structure used in pervasive systems should be scalable, distributed and robust against failure of hosts. In order to support those features, a data structure containing tuples, each representing an object-location relationship is used.

Although this architecture provides useful information about the location of each object in the system, redundant information about the same location is stored in different tuples. This architecture does not take into consideration the limited storage available on mobile devices and the extensive need for accessing location information. This deficiency will be overcome as will be explained later through separating location information from entity information. Thus, the entity can simply have a reference to the location where it resides without storing both sets of information together.

Another important concept in pervasive computing applications, which depends on location information, is context. Context can provide us with information about the surrounding various physical spaces, situations, and the location of persons and devices [4]. Location information is one of the attributes of complex context

information. Location identification and management is also a core functionality in a typical pervasive system, providing users with information specific to user location, activity and nearby objects [5, 6]. Typically, all such information will need to be transferred across various networks, thus raising difficulties such as firewall rules that may block this critical information from being delivered. Using XML to transfer information is a good option since XML documents will be easy to transfer from one network to another bypassing most firewall obstacles

In [9], a survey of the approaches made towards modeling context information is presented. One of such ways is indeed done using XML. The representation has a root element denoting the *context* that is represented. Information describing the location of the *context* is included as attributes in a sub-element called *spatial* where the coordinates of the location, zone are represented.

3. Supporting Location through Java

3.1. XML

The location topology is represented as an XML structured document that contains all required information necessary to describe a topology. The Document Type Definition (DTD) of this XML structure is shown in Figure 1. The information is organized in a way that makes the document self-expressive and easy to parse. Our XML representation of location has seven main elements:

- **LocationTopology:** Is the root of the document. It has one attribute, *name*, that identifies the name of this location topology. It should have an instance of each of *read*, *write*, *description* elements and one or more *location* elements.
- **Description:** Carries the description information of the location topology. It should have five attributes that describe this location: *country*, *state*, *suburb*, and *building*.
- **Location:** Represents the information of a certain location in the topology. It has four required attributes: *floor*, *name*, *type* and *inheritParentAccessRules*. *Floor* represents the floor to which this location belongs in a topology. *Name* is the unique identifier for a location, so it should be unique among all the location elements in the same document. *Type* represents the type of that location; its value is bound to the following list: *room*, *meetingroom*, *office*, and *floor*. This list may be modified later to accommodate for other room types in different topologies. A location element may have nested child location elements.

```
<?xml version='1.0' encoding='UTF-8'?>

<!ELEMENT locationTopology
(read,write,description,location*)>
<!ATTLIST locationTopology
   name CDATA #REQUIRED>

<!ELEMENT description EMPTY>
<!ATTLIST description
   country CDATA #REQUIRED
   state CDATA #REQUIRED
   suburb CDATA #REQUIRED
   street CDATA #REQUIRED
   building CDATA #REQUIRED>

<!ELEMENT location (read? , write? , location*) >
<!ATTLIST location
   floor CDATA #REQUIRED
   name ID  #REQUIRED
   type (room | meetingroom | office | floor | kitchen)
#REQUIRED
   inheritParentAccessRule (true | false) "true" >

<!ELEMENT write (allow , deny?)>

<!ELEMENT read (allow , deny?)>

<!ELEMENT allow EMPTY>
<!ATTLIST allow
   users CDATA "*"
   roles CDATA #IMPLIED>

<!ELEMENT deny EMPTY>
<!ATTLIST deny
   users CDATA "*"
   roles CDATA #IMPLIED>
```

Figure 1 DTD Representation of the XML Location Topology

- **Write:** Represents the write access rules to a specific node in the document. It should have an *allow* element, while a *deny* element is optional.
- **Read:** Is similar in structure to the *write* element but it defines the read access rules to a node.
- **Allow:** Lists the allowed users/roles. It has a required attribute *users* that defines the list of the users granted the access right. However, *roles* is an optional attribute that may be used in case of the need to define a list of roles granted by the access right. *Users* can be listed as a comma separated list of users or "*" denoting all users. *Roles* are treated the same way.
- **Deny:** Has the same structure of the *allow* element but it defines the users/roles denied by the access right to a node.

3.2. The Location Topology Parser

A set of functionalities is implemented in a utility class called LocationTopologyParser that is used to parse the XML document representing the location topology. Any communication with the XML document is done through this utility class. Each of the following functions requires the name of the user initiating the function call so that appropriate access rules may be enforced as indicated later. The following functions are available in the LocationTopologyParser:

- **Location getLocation (String name, String userName):** Gets the location whose *name* holds the given value. If no matching location is found a *null* reference is returned.
- **Location getParentLocation (String locationName, String userName):** Gets the parent location of the location with the given *name*.
- **Location findNearest(String locationName, String locationType, String userName):** Finds the nearest location of a given *type* to the location with the given *name*.

Each of the three methods above returns a Location object that represents the requested location node in the XML document. All three functions check for the access rights granted to the requesting user before returning any information. The username is passed to each of these functions. For example, if findNearest should return location X to a certain user while this user does not have read access rights to this location, then location X will not be returned and the next nearest location to which the user has access to will be returned. If the user has no access to any of the locations in the document then null will be the return value.

To aid in the parsing of the XML document, five classes have been implemented to represent the various nodes in the XML document. These classes are Read, Write, Allow, Deny, and Location. The Location class represents the location node in the XML and contains references to objects of each of the previous four classes to resemble the structure in the XML. This methodology has been followed to facilitate the usage of the location information in the by developers to avoid the need to communicate with the XML document redundantly.

3.3. Access rules

Implementing access rules for the different locations in a pervasive system is of significant importance. Various users may not be allowed to access information related to certain locations. Typical known access rules have been incorporated in the chosen XML structure. The following strategy has been implemented to ensure that only allowed users can access location information:

- Child locations inherit access rules by default from their parent. An attribute named *Inheritparentaccessrules* determines whether a child location entry should inherit the access rules of its parent or not. The default value of the attribute *Inheritparentaccessrules* is true.
- *Inheritparentaccessrules* should be set to false if the child location elements are not meant to inherit the access rules of their parent location.
- Access rules of a child location, if present, override that of the parent. It is optional for a *location* element to have a *read* element and a *write* element. *Read* and *write* elements allow a child to override access rules set by its parent.

Each *read* or *write* element can contain allow or deny elements. The following access rules apply:

- An empty *allow* or *deny* element means that its value is “*” which means that all users are allowed or denied.
- If an *allow* element with the value "*" for users is present without a *deny* element, then all users are allowed.
- If an *allow* element with the value "*" for users is present, and a *deny* element with a list of users is present, then all users but those in the user list of the *deny* element are allowed.
- If a list of users exists in the *users* attribute for the *allow* element, then only these users are allowed and all other users are denied even if there is a list of users in the *deny* element.

For example, the XML snippet represented in Figure 2 shows that all users can read from and write to the location topology “Main Campus”. However, the location named “Blue Room” will override the write access rules of its parent by denying the user “username” from modifying the information of this location while inheriting the read access rules of its parent.

```
<locationTopology name="Main Campus">
  <read><allow /></read>
  <write><allow /></write>
  <description
        country="Egypt"
        state=""
        suburb=""
        street="Main" building="14"/>
  <location floor="1"
        name="Blue Room"
        type="meetingroom">
    <write>
      <allow />
      <deny users="username" />
    </write>
  </location>
        …
</locationTopology>
```

Figure 2 Sample Section of a Location Topology

3.4. JavaSpaces

As indicated in [2], a pervasive computing environment should support user mobility, adapt to runtime changes, and provide a clean separation of concerns between data and functionality [8]. Upon reviewing the various Java technologies that support this behavior, the JavaSpace services of the Jini technology of Java proves itself to be the technology that best fits the required pervasive nature.

Jini is a Java based technology that is built on top of the Java remote method invocation system, RMI [12]. A Jini system is a distributed system based on the idea of federating groups of users and the resources required by those users [12]. The goal of such system is to make the resources easily accessible by different users of the system. Jini is totally independent of the underlying network, thus it can work with wired as well as wireless networks, which in return extends the area that a pervasive system will serve.

In an attempt to exploit such features of Jini, we shall store the location topology using the JavaSpace technology, thus ubiquitous access to this information is guaranteed. It is worthy to note that Jini deals with a JavaSpace as a service that is offered through the Jini network. Security is an important feature that also adds to the advantages of using Jini, as it extends Java, thus providing some level of protection against execution of malicious code on the JVM.

The JavaSpaces service, along with other services, represents one of the main three segments of the architecture of Jini. A JavaSpace can hold any kind of data as long as this data can be serialized while writing it to the JavaSpace, and deserialized while taking/reading it from the JavaSpace.

3.5. Storing Location Topology using JavaSpaces.

The XML documents representing the various location topologies in a system will be stored in JavaSpaces using the Jini technology. Fetching location information from the XML structured document is done through utilizing the dynamic and distributed nature of JavaSpace services. There are two classes that we created and are responsible for managing the JavaSpace component in the system.

- ***LocationTopologyEntry:*** Represents the XML document of the location topology that will be saved in the JavaSpace, and implements the net.jini.core.entry. The entry interface in JavaSpaces holds various types of entries [12], and it also has data as an attribute that is used to create a search template to search for a topology in a JavaSpace.

- ***JavaSpaceManager:*** Manipulates the pool of available JavaSpaces. The user can add a new JavaSpace to the pool or remove an existing JavaSpace. There is, however, always a default JavaSpace available in the pool called JavaSpace that uses the default Jini service for JavaSpaces.

The user can add a new topology to a specific JavaSpace. If no JavaSpace is specified, it will be added to the default JavaSpace. A specific topology can either be retrieved from any of the available JavaSpaces or in specific from a particular JavaSpace. A topology can also be removed completely from the JavaSpace where it resides, and hence, it will no longer be available for the actors in the pervasive system.

A set of functions offered by the classes we created can be used to manage the JavaSpace and extract information from a location topology.

Figure **3** shows a set of function that can be used to manage the different JavaSpaces that the application may have access to. Such methods provide functionalities to add and remove Java Spaces, and to add, remove, and retrieve topologies from the Java Spaces.

```
public void addJavaSpace(String javaSpaceName)

public void removeJavaSpace(String
javaSpaceName)

public void addTopology(LocationTopologyEntry
locTopEntry)

public void addTopology(LocationTopologyEntry
locTopEntry, String spaceName)

public void
removeTopology(LocationTopologyEntry
locTopEntry)

public void
removeTopology(LocationTopologyEntry
locTopEntry, String spaceName)

public LocationTopologyEntry
retrieveTopology(String name)

public LocationTopologyEntry
retrieveTopology(String name, String
spaceName)
```

Figure 3 Functions Created for Managing Available JavaSpaces

4. Example

After discussing how we supported location for pervasive systems through Java, a concrete example will help visualize and comprehend how the various classes we created can be used. A sample XML document that represents the topology of some hypothetical university is described in Figure 4.

The document should comply with the rules and specifications set in the Document Type Definition (DTD) aforementioned in this article. Consequently, using the classes we created to manage the JavaSpaces present in the Jini network, this topology can be written to the JavaSpace thus making it available to all the users who can access this network.

A LocationTopology object should be created as such. This object will include the XML document along with a name to identify the document. The code in Figure 5 illustrates the creation of a JavaSpaceManager, followed the creation of a LocationTopology object, binding it to a name, populating it with the required topology defined in the XML document, and then eventually storing it onto the JavaSpace.

```
<locationTopology name="Main Campus">
  <read><allow /></read>
  <description
        country=" "
        state=""
        suburb=""
        street="Main" building="14"/>
  <location floor="1"
        name="Blue Room"
        type="meetingroom">
    <write>
      <allow />
      <deny roles="guest" />
    </write>
    <location floor="1"
        type="kitchen"
        name="Buffet">
      <read>
        <allow />
        <deny users="user" />
      </read>
      <write><allow /></write>
    </location>
  </location>
  <location floor="2"
        name="Manager Room"
        type="office">
    <write>
      <allow users="manager"/>
      <deny users="*" />
  </location>
</locationTopology>
```

Figure 4 Sample Location Topology for a University

```
JavaSpaceManager manager =
        new JavaSpaceManager();

LocationTopologyEntry locationTopology =
        new LocationTopologyEntry();

locationTopology.data = "name";

locationTopology.document =
        DocumentBuilderFactory.newInstance().
        newDocumentBuilder().parse(new
        File("path/docment.xml"));

manager.addTopology(locationTopology);
```

Figure 5 Creating a LocationTopology object and Writing it to a JavaSpace

In order to retrieve the *LocationTopology* object that has just been written onto the JavaSpace, we need to pass its name to one of the search functions in the *JavaSpaceManager* class. We can either retrieve the topology by taking a copy from it, or we can totally remove it. Figure 6 shows how we can retrieve the *LocationTopology* object that was written above.

```
manager.retrieveTopology("name");
```

Figure 6 Retrieving a Location Topology from the JavaSpace Pool

Once the topology is retrieved, all the functions that the *LocationTopologyParser* class offers can be used in order to extract location related information. Assuming the following scenario: A user with the user name "user1" who is present in a kitchen of the first floor, wishes to retrieve the location information regarding the nearest office to his current location. All what needs to be done to get such information is to call the function *findNearest*, and passing the proper arguments as shown in Figure 7.

Typically, finding the nearest location could be achieved using multiple approaches. For one, it may include finding the Euclidean distance between two points. However, although two locations may be geometrically close to one another, their accessibility to one another may make them very far away from one another. According to the topology we have in the XML representation of the location information we can know given a certain location what the nearest location is of a given type to that location by parsing this XML document. Locations on the same level are considered closer to each other than locations in sub-levels or previous levels.

```
LocationTopologyParser parser =
        new LocationTopologyParser
        (locationTopology.document);

Location location = parser.findNearest("Buffet",
LocationType.OFFICE, "user1");
```

Figure 7 Finding the Nearest Office

Access to the nearest location of type office is restricted to the user having the user name "manager" and there are no other offices in the topology. Therefore, the return value of the location will be null denoting that no such location can be found in the given topology. However, if "user1" wishes to know where the nearest meeting room is, an object containing the information of the "Blue Room" (See Figure 4) will be retrieved given it is the nearest location of type meeting room that such user has access to. Figure 8 illustrates this kind of query.

```
LocationTopologyParser parser =
        new LocationTopologyParser
        (locationTopology.document);

Location location =
        parser.findNearest ("Buffet",
        LocationType.MEETING_ROOM,
        "user1");
```

Figure 8 Finding the Nearest Meeting Room

Another scenario assumes that the user "user" wants to get the location information of the Buffet that is present on campus. However, it happens that such user is denied any read access to this location, so a null object will be the result. Figure 9 shows how to get the location information of the Buffet.

```
LocationTopologyParser parser =
        new LocationTopologyParser
        (locationTopology.document);

Location location =
        parser.getLocation("Buffet", "user");
```

Figure 9 Finding a Location

5. Conclusion

In this article, we presented an approach for building programming language support for pervasive systems development activities by extending the Java programming language. Typically, support for context, location, actors, sensors, and events is needed in the efforts made towards building complex pervasive systems. We demonstrated in this article our programming language support for location and its associated management and retrieval through utilizing the extensible markup language, Javaspaces, access rights, and eventually supporting libraries. We finally demonstrated an example that illustrates the utilization of this approach for supporting location related development activities.

References

[1] M. Weiser. "The computer for the 21st Century. *Scientific American*, September, 1991.

[2] D. Heutelbeck and M. Hemmje. "A peer-to-peer data structure for Dynamic Location Data." In Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06), 2006.

[3] S. Fritsch, A. Senart, and S. Clarke. "Addressing dynamic contextual adaptation with a domain-specific language." 29th International Conference of

Software Engineering Workshops (ICSEW'07), 2007.
[4] T. Weis, M. Knoll, A. Ulbrich, G. Muhl, and A. Brandle. "Rapid Prototyping for Pervasive Applications." IEEE Computer Society. pp. 76-84, 2007.
[5] J. Indulska and P. Sutton. "Location management in Pervasive Systems." Appeared at Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing, Adelaide, Australia, 2003.
[6] S.W. Loke. "Logic programming for context-aware pervasive computing: Language support, characterizing situations, and integration with the web." Proceedings of the IEEE/WIZ/ACM International Conference on Web Intelligence (WI'04), 2004.
[7] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall. "Systems Directions for Pervasive Computing." Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII'01), 2001.
[8] K. Harihar and S. Kurkovsky. "Using Jini to enable pervasive computing environments." 43rd ACM Southeast Conference, March 18-20, 2005.
[9] G. K. Mostefaoui, J. Pasquier-Rocha, and P. Brézillon , "Context Aware Computing: A Guide for the Pervasive Computing Community", Proceedings of the IEEE/ACS International Conference on Pervasive Services(ICPS'04), 2004.
[10] M. Baldauf, S. Dustdar and F. Rosenberg. "A survey on context-aware systems." Int. J. Ad Hoc and Ubiquitous Computing, vol 2, pp 263 – 277, 2007.
[11] "Jini Specifications Archive - v2.1." Sun Developer Network (SDN). 2007. Sun Microsystems, Inc. 29 Sep 2007 <http://java.sun.com/products/jini/2.1/doc/specs/html/js-spec.html>.
[12] "Jini Technology Architectural Overview." Sun Microsystems. January 1999. Sun Microsystems, Inc.. 20 Oct 2007 <http://www.sun.com/software/jini/whitepapers/architecture.html>.

Sherif G. Aly received his B.S. degree in Computer Science from the American University in Cairo, Egypt, in 1996. He then received his M.S. and Doctor of Science degrees in Computer Science from the George Washington University in 1998 and 2000 respectively. He worked for IBM during 1996, and later taught at the George Washington University from 1997 to 2000 where he was nominated for the Trachtenberg prize-teaching award for his current scholarship and scholarly debate. He spent two years as a guest researcher for the National Institute of Standards and Technology at Gaithersburg, Maryland from 1998 to 2000. Dr. Aly also worked as a research scientist at Telcordia Technologies in Morristown, New Jersey, in the field of Internet Service Management Research, and as a Senior Member of Technical Staff at General Dynamics Network Systems. He also consulted for Mentor Graphics and taught at the German University in Cairo. He is currently a faculty member at the Department of Computer Science and Engineering at the American University in Cairo, and recipient of the Egypt National State Prize for research. Dr. Aly published numerous papers in the area of distributed systems, multimedia, digital design, and programming languages. His current research interests include pervasive systems, programming languages, multimedia, directory enabled networks, and image processing. Dr. Aly is a member of IEEE.

Sarah Nadi and Karim Hamdan received their B.Sc. from the Computer Science and Engineering Department of The American University in Cairo. Both have graduated with high honors, and are different recipients of the best senior project awards during their respective semesters. Their research interests include programming languages, and mobile and pervasive systems. They are currently Masters degree candidates in Canada.