SpatialViews Language Specification*
Version 1.0

DCS-TR564

Ulrich Kremer Yang Ni Adrian Stere

November 2004
(updated January 2005)

1 Summary

The goal of SpatialViews (SV) is to provide a high-level programming model that allows application programmers to easily develop and maintain their mobile ad-hoc network applications. Each node in the network is assumed to have substantial computation and communication capabilities, and is aware of its spatial location. The location of a node can be queried by the user level program, i.e., a particular location may determine the actions performed by a program. Resource-constrained sensor networks are not the main target of SpatialViews . Examples for target network nodes are state-of-the-art smart phones, PDAs, notebook, and laptop computers.

The language is a vehicle to study different language, compiler optimizations, and performance trade offs. Clearly, not all conceivable applications may be implemented within the framework of this programming model. Our goal is to provide a high-level programming model for a large class of mobile applications for ad-hoc networks that hides many details of the underlying volatile target networks. In this sense, SpatialViews is complementary to lower level languages such as nesC [4], SP [2] and SM [7]. A programming language approach was choosen for SpatialViews instead of a library (API) approach in order to allow more aggressive optimizations and more efficient program analyses.

The SpatialViews language is an extension of a Java subset. The main program abstraction in SpatialViews is that of a space-time region of virtual network nodes that provide specific services. A SpatialViews can be thought of as a type specification that describes the desired network type and properties. This space-time region or type is called a **spatial view**. The *spatial view* definition is declarative in nature, i.e., does not instantiate any mapping between virtual and physical nodes. This is done through the "iterator" as briefly discussed in the next paragraph.

An iterator allows the programmer to specify actions on nodes in the spatial view. The operational semantics of the iterator is based on migration, i.e., the programmer can imagine that each iteration is performed on a different virtual node, with program migration performed implicitly between the iterations. The spatial view is instantiated by discovering and migrating to network nodes that match the type definition of the spatial view. Constraints on the discovery and migra-

*This work was partially supported by NSF-ITR/SI award ANI-0121416.

tion process may be specified as part of the iterator. A **time limit** is a required constraint for each iterator. Each virtual node in a spatial view can be instantiated, i.e., visited at most once.

Due to the volatile and mobile nature of the underlying target network, SpatialViews does not support any explicit mapping between a virtual network node and its corresponding physical node. In other words, the language does not provide names (e.g. "addresses") of a physical node across iterations. Any physical node that matches the virtual node type as specified in a *spatial view* can be an instance of that virtual node. The motivation for this restriction is the fact that any particular binding of virtual and physical nodes cannot be guaranteed to be valid during the program execution due to the volatility of the underlying network.

The programming model introduced by SpatialViews could be implemented as an extension to the Java programming language through additional syntax and semantics, or through a Java package that contains class definitions that may be imported into "regular" Java programs. There are advantages and disadvantages to both approaches. We chose the programming language approach which allows a better control of the semantics of the newly introduced programming model concepts.

2 Location Awareness

The location awareness is one of the main features of the SpatialViews programming model, i.e., every node in the network is assumed to know its location through the access to a location service or even multiple location services. Such location services may be based on GPS, Berkeley's Mote beacons, 802.11, or MIT's Crickets ([5, 8, 9, 1]). Location services may have different characteristics in terms of their applicability (e.g.: indoor or outdoor), their accuracy (e.g.: accurate within 20 meters or a few centimeters), and the time needed to determine the location after a location request, ranging from hundreds of milli-seconds to several seconds.

The programmer has to be aware of the precision of the location service since it determines whether the application will perform as desired. For instance, some applications may require a location resolution within meters or fractions thereof, rather than multiple meters. An example for such an application is one that tries to find sensors within a small space such as a car. The desired location resolution Δs has to be supported by the location resolution Δa provided by the location service, i.e., $\Delta s \geq \Delta a$ has to hold. For example, two locations specified as cartesian coordinates x $= (x_1, x_2, x_3)$ and y $= (y_1, y_2, y_3)$ are considered the same if

$$\forall i \in \{1, 2, 3\} : |x_i - y_i| \leq \Delta s$$

If the underlying location service cannot provide the location accuracy as specified by the programmer, the semantics of the program is not defined. A runtime check verifies that every call to the SpatialViews runtime system method `location`, whether implicit or explicit, results in a location with the required precision.

3 Spatial View Definition

A spatial view defines a virtual network type. A spatial view is dynamically instantiated during the execution of a Spatial View Iterator. A virtual network type is a collection of virtual nodes that

- reside within a physical space,

- provide a specified set of services, and
- can be distinguished in terms of their location and across time.

Physical spaces may have unique names, called **SpaceID**s. **SpaceID**s may be defined within a space library or may be defined explicitly by the user. For instance,

`Rutgers.Busch.CoRE`

may specify a particular building on Rutgers University's Busch Campus, called the CoRE building. Refinements of the space definition such as `Rutgers.Busch.CoRE.3rdFloor.Room334` may be supported. Such space specifications may be imported as part of predefined space packages.

In general, a SpaceID is a Java class that implements the SpatialViews `Space` interface directly, or indirectly through an ancestor super class that does. The interface requires a single method, namely **contains** that returns an object of class `Location`. It is the responsibility of the user, i.e., programmer to provide an appropriate implementation of class `Location`. For example, the user may choose a two-dimensional or three-dimensional space model. If necessary, a user may need to translate the location service space representation provided by the location service of a network node into the location abstraction used in the space definition, i.e., in the definition of a *SpaceID*. The location service provided by the network nodes are assumed to be of the form (longitude, latitue, and altitude) and are part of the SpatialViews runtime environment.

```
public interface Space {
    public boolean contains(Location l);
}
```

The following code fragment defines the SpaceID `Circle`.

```
public class Circle implements Space {
    private Location center;
    private float radius;

    public Circle(Location c, float r) { center = c; radius = r; }

    public boolean contains(Location l) {
        LocationService ls = SpatialViewsEnvironment.locationService;
        return l==null || ls.distance(l, center) < radius;
    }
}
```

Space expressions allow the construction of composed spaces, for instance only the second and third floor of a building (`CoRE.2ndfloor + CoRE.3rdfloor`). This corresponds to the union of two spaces. Intersections of spaces are also possible using the "&" operator, for example (`HighWay59 & Interstate10`). This is an absolute space expression since it is based on physical objects with absolute coordinates. Relative space expressions may specify spaces that are relative to the location where the space is instantiated. The initial language specification contains only a single relative

specification, namely `cycle`, which allows the definition of a space as all locations within a particular radius.

Each service is named by a particular Java class, i.e., by its **ServiceID**. There is the implicit assumption that services which provide the same interfaces are semantically equivalent. The requirement for a set of services can be expressed through a service expressions.

The density of the virtual network in terms of space and time must be defined using a **Space-Time** expression consisting of a space and time component, `(radius, time)`. Two virtual nodes are considered distinct if they are not located within `radius` meters from each other or are visited more than `time` seconds apart. Both, space and time granularities as specified by floating point numbers. It is the responsibility of the programmer to select appropriate values for this constraint. The property of the target network, in particular the quality and resolution of the location service will have an impact on the granularity selection. For instance, if the location service has a precision of determining a location within 20 meters, a location granularity of less than 20 meters, for instance 1 meter, is not meaningful.

3.1 Syntax

<SpaceExpression> ::= <RelativeSpaceExpression> | <AbsoluteSpaceExpression> | **anywhere**

<RelativeSpaceExpression> ::= **cycle(FloatConst)**

<AbsoluteSpaceExpression> ::= **SpaceID** + <AbsoluteSpaceExpression> |
SpaceID & <AbsoluteSpaceExpression> |
(<AbsoluteSpaceExpression> **)** |
SpaceID

<ServiceList> ::= **ServiceID** , <ServiceList> |
ServiceID

<SpaceTimeExpression> ::= **(FloatConst**, **FloatConst secs)**

<SpatialView> ::= <ServiceList> **@** <SpaceExpression> **%** <SpaceTimeExpression>

<SpatialViewDefinition> ::= **spatialview SpatialViewID** = <SpatialView>

3.2 Examples

The first example below defines a virtual network that consists of nodes with cameras in the CoRE and Hill buildings on Rutgers' Busch Campus. The location specification is based on cartesian coordinates. If two cameras nodes are less than 2 meters apart, then only one of the nodes may be a member of the defined network. If a node has not been visited for at least 10.0 seconds, it may be visited again, i.e., may represent another, distinct virtual network node.

The second example describes a relative space that describes all nodes with motion and light sensors on the second floor of the CoRE building, which have to be within 20 meters of the node that initiates the iteration of the defined space. The space/time constraint is the same as in the previous example.

```
import SpaceDefinition.Rutgers.*;
```

```
spatialview SV1 = Camera @ Busch.CoRE + Busch.Hill % (2.0, 10.0 secs)

spatialview SV2 = MotionSensor, LightSensor @ cycle( 20 ms ) % (2.0 , 10.0 secs)
```

4 Spatial View Iterator

A spatial view iterator instantiates a network as defined by a spatial view. During the iterator's execution, nodes are discovered, and nodes that match the spatial views definition are visited. The visiting process typically requires program migration. The programmer may not make any particular assumptions about the order in which the nodes are visited, i.e., the order in which the particular iterations are performed. In fact, several nodes may be visited "at the same time", which does not correspond to any possible "sequential" visiting order. Spatial view iterators may be nested.

Each iteration is executed on a unique member of the virtual network defined by the spatial view definition. An iteration may not be performed on the same virtual node more than once. After an iteration has been performed, the program implicitly migrates to another virtual node while respecting the iteration constraints. The iterator guarantees that at all times the visited physical node is a valid instance of a virtual node defined by the spatial view. For example, if an iteration contains a spatial view iterator itself, it is not guaranteed that on "return" from the embedded iterator the same physical node is still bound to the virtual node.

A required iteration constraint is the upper bound on the time that the iterator is allowed to execute. If this time limit is exceeded, active iterations will terminate after reporting any possible partial results. The time constraint is not a real-time constraint, but only a soft constraint. The main goal is to make sure that the iterator terminates. Without a time constraint and due to the dynamic nature of the underlying network, new nodes could always appear, resulting in an infinite execution.

A "visited" constraint allows the programmer to limit the number of virtual nodes visited during the execution of an iterator. Once the limit is reached, no more nodes will be visited and the iterator terminates. However, it is important to note that the constraints do not guarantee that the iterator terminates within the time constraint, or that the specified number of nodes in the visited constraints are actually found.

The mapping between a virtual node and a particular physical node is only guaranteed during the execution of a code segment that is part of a single iteration and does not contain any embedded spatial view iterators. In other words, the execution of an iteration is assumed to be *location atomic* until the iteration terminates or a nested iterator is encountered. This means that the programmer can assume that migration is not performed while executing a single iteration that does not contain any nested iterator. This is important in order to allow reasoning based on location. For instance, if the program instructs an available camera to take a picture and then determines the location of the camera node in order to store it with the picture, the program should not migrate between taking the picture and requesting the location. Otherwise, the semantic link between data, events, and/or location may be broken.

4.1 Syntax

<statement> ::= ... | <SpatialViewIterator>

<SpatialViewIterator> ::= **visiteach** (<ServiceObjectDecls>) **: SpatialViewID**

with <Constraints> <block>

<ServiceOjectDecls> ::= **ServiceID ServiceObject ,** <ServiceOjectDecls> |
ServiceID ServiceObject

<Constraints> ::= **(** <TimeConstraint>, <NodeNumberConstraint> **)**

<TimeConstraint> ::= <= **FloatConst secs**

<NodeNumberConstraint> ::= **unlimited nodes** | <= **IntConst nodes**

<block> ::= **{** <declarations> **;** <statement_list> **}**

The declaration of service objects have to match the services specified by the used spatial view.

4.2 Examples

The following code segment describes an iterator that tries to visit at most 10 nodes in the spatial view of type `SV1`. The iterator will terminate after 100 seconds, which is a soft time deadline. On each of the visited nodes, the program takes a picture, processes the picture, and reports results. The `reportResults` method should modify a global data structure that represents the final answer of the SpatialViews program.

```
import SpaceDefinition.Rutgers.*;

spatialview SV1 = Camera @ Busch.CoRE + Busch.Hill % (2.0, 10.0 secs)
visiteach (Camera cam) : SV1 with constraints (<= 100.0 secs, <= 10 nodes) {
  Picture pic;
  cam.TakePicture(pic);
  process(pic);
  reportResult();
}
```

---

5 SpatialViews Program

A SpatialViews program is a Java program that contains Spatial View definitions and Spatial View Iterators. The machine that executes the `main` method of the program is called the **injection point**. The injection point itself does not define any interesting space, but has importance with respect to the output of the SpatialViews program. Once all spatial view iterators have terminated, the program has to reinstate the effects of the program execution on globally visible variables and data structures in the injection point machine. This will require some form of process and data migration back to the machine that initiated the SpatialViews program execution.

5.1 Program and Service (Node) Variables

SpatialViews is a statically scoped language. Every variable or object in a program is either a **program** variable or a **service** (virtual node) variable. The state or value of a program variable is computed and accessed during the execution of the program and migrates with the program. Service variables may also be accessed and even created by a program, but their values reside on the virtual nodes on which there were created, i.e., service variables do not migrate with the

program execution. By default, all variables are considered program variables. Service variables have to be defined explicitly.

Service variables allow a program to "deposit" and "retrieve" information on virtual nodes, allowing the communication between different SpatialViews programs. In addition, service variables can be used in the definition of a Spatial View, i.e., in defining a set of virtual nodes by specifying that a particular service variable has to be present. It is important to note that service variables do not provide a shared memory abstraction since values of distinct instantiations of a service variable across virtual nodes can be different. In other words, node variables do not provide shared memory with an underlying consistency model, as proposed in languages such as Linda [3] and Spatial Programming [6, 2]. Every variable/object in a SpatialViews program is either a program variable or a service variable. Service variables cannot be part of the implementation of a program variable or vice versa.

5.1.1 Program Variables

There are three categories *program* variables in a SpatialViews program. Each category has specific access constraints, giving opportunities for different optimizations. If no SpatialViews keyword is used, a variable is assumed to be local.

1. local: is read/write within defining iteration, and read-only within nested iterators.
2. container: Collection of objects; is read/write within defining iteration, and write-only within nested iterators. The corresponding abstract data type is that of a set of elements of a particular type. "Write-only" means that objects can only be inserted into the collection, but not read or removed. Container variable declarations start with the keyword `collection`.
3. reduction: reduction variable together with a commutative and associative operation; read/write within defining iteration, and apply-reduction-operation-only within nested iterations. The initial SpatialViews language will only support a rather small subset of reductions, such as sum and product reductions. Reduction variable declarations start with the keyword `reduction`.

The main method of the program, i.e., its injection point, can be considered a single iteration of an iterator over the "dummy" spatial view consisting of only a single node, namely the injection node. Since there is only a single node in this spatial view, the program has to return to it if a side effect such as an output is specified by the program.

5.1.2 Service (Virtual Node) Variables

The main motivation for service variables is the support of cooperation across multiple SpatialViews programs, and the access to services from within an SpatialViews program provided by or residing on physical nodes temporarily bound to a virtual node. User-defined service variables may be used in the definition of a spatial view in addition to system-defined services that allow access to, for instance, hardware on a particular network node. Service variables have to be declared using the `service` keyword as a prefix.

6 SpatialViews Runtime Library

Each node that may be part of the virtual network needs to provide the following SpatialViews Runtime System methods

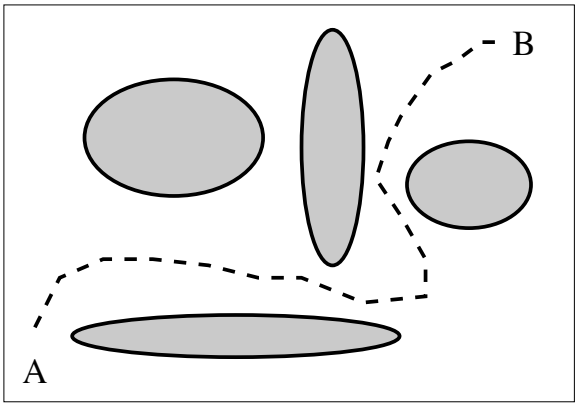

Figure 1: The "glob" game.

---

1. currentLocation(coordinates, accuracy): give the location within a 3-dim space as GPS coordinates of the form (longitude, latitue, and altitude), together with the location accuracy Δa.

2. A time function that provides "wall-clock" times.

3. Some form of "garbage collection" is needed for expired SpatialViews programs and user-defined SpatialViews services.

7 Program Examples

The examples should illustrate different "features" of the language allowing the reader to assess its expressiveness.

7.1 Find a computation service

This application shows that SpatialViews programs can be used to identify services that allows off-loading of work from a handheld to a dedicated server. Here the application would be image rendering. As we move through a space, the QoR may significantly improve if a server is able to perform the rendering. However, the mobile device has to be able to perform rendering as well, which is does if no server was found, or the server did not respond in time.

7.2 The "glob" game

One group of players implicitly define a space that the opponent player cannot enter. Goal: Opponent player has to get from point A to B without getting caught by globs created by the first group of players. A possible path may be the one as shown in Figure 7.2. However, the globs are dynamic, so path planning will involve some prediction about the future positions and movements of the globs.

7.3 Local Spatial Views and Nested Iterators

```
import SpaceDefinition.Rutgers.*;
...
spatialview SV1 = Camera @ Busch.CoRE + Busch.Hill % (2.0, 10.0 secs)
visiteach (Camera cam) : SV1 with constraints (<= 100.0 secs, <= 10 nodes) {
  Picture pic;
  camera.TakePicture(pic);
  {
    spatialview SV2 = FaceDetection @ anywhere;
    visiteach (FaceDetector detector) : SV2 with constraints (<= 60.0 secs, <= 1) {
      detector.DetectFace(pic);
    }
  }
  reportResults();
}
...
```

7.4 Service Installation

This example follows a consumer / producer paradigm. The first program produces information in all nodes about its surrounding neighbors for a week (604800 seconds). The second program reads this information.

The **producer** does not return any values to the injection point since there is not state that needs to be observed at the injection point after the spatial view iterator has terminated.

At any point in time, a physical node can only represent a single virtual node within a particular Spatial View. This allows values of service variables to be "overwritten" if a physical node represents different virtual nodes of the same Spatial View along the time or space dimensions.

```
// ------------- PRODUCER -------------
import SpaceDefinition.Rutgers.*;

public interface NeighborInfo {

public int numberOfNeigbors;
public int averageDistance;
public int averageAvailability;
}
...
spatialview SV1 = Location @ campus % (2.0, 30 secs);

// A service variable can only be declared within an iterator, i.e.,
// ``on'' a virtual node

visiteach (Location loc) : SV1 with constraints
                         (<= 604800.0 secs, unlimited nodes) {
  NeighborInfo neighbors;
  spatialview SV2 = Location @ cycle (20 ms) % (2.0 ms, 1000000 secs);
  collection info;
  visiteach (Location lloc) : SV2 with constraints
                            (<= 2 secs, unlimited nodes) {
    info.put(lloc.mylocation(), lloc.uptime());
  }
```

```
  int averageDist = 0;
  int averageAvail = 0;
  while (info.NotEmpty()) {
    averageDist  = averageDist + (loc.mylocation - info.current.mylocation);
    averageAvail = averageAvail + info.current.uptime;
    info.Next();
  }
  neighbors.numberOfNeighbors = info.NumberOfEntries;
  neighbors.averageDistance = averageDist / info.NumberOfEntries;
  neighbors.averageAvailability = averageAvail / info.NumberOfEntries;
}
...
```

```
// ------------- CONSUMER -------------
import SpaceDefinition.Rutgers.*;

public interface NeighborInfo {
public int numberOfNeigbors;
public int averageDistance;
public int averageAvailability;
}
...
spatialview SV3 = NeigborInfo @ Busch + Cook % (20.0, 1000000 secs);

reduction averageDist;
reduction visited;
visiteach (NeighborInfo node) : SV3 with constraints
                              (<= 60 secs, unlimited nodes) {
  averageDist =+ node.averageDistance;
  visited++;
}

print(``Current Average Distance is `` averageDist / visited);
...
```

---

8 Current Distribution

The current SpatialViews compiler and runtime environment supports only a subset of the SpatialViews language. The current distribution together with a set of example programs can be found on the project web site http://www.cs.rutgers.edu/ ...

In case of questions regarding this document or the current SpatialViews distribution, please send an email message to spatialviews@cs.rutgers.edu.

References

[1] Paramvir Bahl and Venkata N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *INFOCOM (2)*, 2000.

[2] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using Smart Messages: Design and implementation. In *International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[4] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, 2003.

[5] Ivan A. Getting. The global positioning system. *IEEE Spectrum*, December 1993.

[6] L. Iftode, C. Borcea, D. Iyer, P. Kang, U. Kremer, and A. Saxena. Spatial programming with Smart Messages for networks of embedded systems. Technical Report DCS-TR-490, Department of Computer Science, Rutgers University, May 2002.

[7] Porlin Kang, Cristian Borcea, Gang Xu, Akhilesh Saxena, Ulrich Kremer, and Liviu Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal, Special Issue on Mobile and Pervasive Computing*, 47(4), January 2004.

[8] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *MobiCom*, 2000.

[9] Nissanka B. Priyantha, Allen K. L. Miu, Hari Balakrishnan, and Seth J. Teller. The cricket compass for context-aware mobile applications. In *MobiCom*, 2001.