# BeanWatcher: A Tool to Generate Multimedia Monitoring Applications for Wireless Sensor Networks*

André Lins[1], Eduardo F. Nakamura[1,2], Antonio A.F. Loureiro[1], and Claudionor J.N. Coelho Jr.[1]

[1] Department of Computer Science
Federal University of Minas Gerais – UFMG
Belo Horizonte, MG, Brazil.
{alla,nakamura,loureiro,coelho}@dcc.ufmg.br

[2] Department of Technological Development
Research and Technological Innovation Center – FUCAPI
Manaus, AM, Brazil.
eduardo.nakamura@fucapi.br

**Abstract.** In this paper we present a new tool called BeanWatcher that allows the semi-automatic generation of multimedia monitoring and management applications for wireless sensor networks. Thus, particularities of multimedia management and wireless sensor networks were taken into account. The architecture of the tool is based on a component model flexible enough to allow the creation of new components and the optimization of the components currently provided. BeanWatcher was designed to offer a development environment suitable for both expert and beginner users allowing them to choose the programming language that better fits the application requirements.

## 1 Introduction

A Wireless Sensor Networks (WSN) is a special kind of *ad-hoc* network with distributed sensing and processing capability that can be used in a wide range of applications, such as environmental monitoring, industrial applications and precision agriculture [1–3]. As a recent practical example, Intel Research and Agriculture and Agri-Food Canada (equivalent to the U.S. Department of Agriculture) are working on a project in which a WSN of motes [4] is used to measure air temperature on a 50-acre vineyard [5] to enhance the growth of grapes and the quality of the wine produced.

Despite their potential applications, such networks have particular features imposed by resource restrictions, such as low computational power, reduced bandwidth and specially limited power source. Current research efforts have followed different lines such as network establishment [6, 7], data dissemination [8,

* This work has been partially supported by CNPq grant number 55.2111/2002-3.

9], network and resource management [10–13]. Therefore, tools to assist designers with the development of management applications in such networks are very useful.

Usually, WSNs are composed of *source* and *sink* nodes [8, 9]. Sources are data generators that detect events and provide observations or measurements of a physical phenomena. Sinks are designed to receive data sent by sources. Therefore, such nodes can monitor and act in the network performing some management function. Besides, sinks can act as gateways between the WSN and a infrastructured network. Thus, sinks may also provide an interface to the user allowing a manager to decide and act based on the data provided. This interface can be textual or multimedia becoming a useful tool to network managers.

In this paper, we use a precision agriculture example as our guiding application. In this scenario, an orchard or plantation can be equipped with several sensors gathering different types of data, such as soil temperature, humidity and acidity or alkalinity. A walking employee can receive data from those sensors through a wireless network interface into a portable device that acts as a sink node as depicted in Figure 1. The application in the portable device can be implemented in Java running on a portable platform such as J2ME, or SuperWaba. In this case, a multimedia interface, with graphics, animations and audio/video stream capability could be provided to exhibit data collected from different sensors and to support the network monitoring and management.

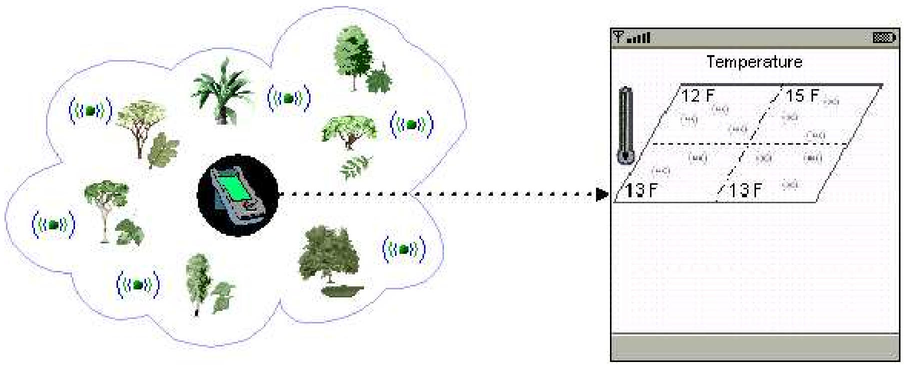


**Fig. 1.** A WSN example.

The application described above is usually designed to solve a specific issue, not considering a more general model, in which we could have, for instance, code reuse. In this work we present a tool called BeanWatcher that aims the code reuse proposing a standardization to the development of such applications. BeanWatcher allows the development of management applications in different programming languages such as Java, C/C++ and Embedded C. Our tool generates a management application for WSNs in a semi-automatic fashion. These applications are intended to be run on portable and mobile devices acting as a sink node monitoring raw sensory data and multimedia data streams.

Some commercial tools like LabView[3] and HP VEE[4] were designed to develop applications to monitor and act on instruments. However, those tools are proprietary solutions hardly integrated with other tools and languages. Furthermore, those tools do not allow the development of applications to portables devices neither monitoring applications for WSNs. Other related tool is the PECOS Component Environment[5] that allows the development of monitoring and actuation applications. In this paper, we show how it can be used to develop applications running in portable devices.

This paper is organized as follows. Section 2 presents some challenges in the design of multimedia management applications for WSNs. In Section 3 we present the component model used in BeanWatcher. Section 4 presents the BeanWatcher architecture and Section 5 shows how we can use BeanWatcher to develop a simple monitoring application. Finally, Section 6 presents our conclusions and future work.

## 2 BeanWatcher Challenges

In the following, we discuss the challenges that we must address when developing a multimedia management application for WSNs.

### 2.1 Multimedia Management for WSNs

Multimedia management faces new challenges in WSNs concerned with provision of scalable quality of service (QoS) through the management of metrics, such as coverage [14–16], exposure [17, 18], energy consumption [12, 13], and application specific metrics (e.g., for target detection, miss detection and false detection ratios). Due to the ad-hoc nature of WSNs, which might be deployed in hostile environments with fairly unpredictable conditions, management must be scalable, self-configurable and adaptive to handle such challenges. A classic approach is the data-centric design of WSNs [8, 9], which aims the integration of application-level and network-level operations to provide power-efficient solutions.

In addition, WSNs can be composed of unrelated sensors that measure different physical properties (e.g., temperature, pressure, image, video and audio streams) that are semantically and/or structurally distinct. In such networks, incommensurate data can be understood as different types of media. In this case, multimedia management encompasses resources, theories, tools and techniques to manipulate data provided by different sources (multiple medias) with the goal to extract relevant information.

[3] National Instruments – Labview 6.1. http://www.ni.com/
[4] Agilent – VEE Onelab 6.1. http://www.agilent.com/
[5] PECOS Composition Environment. http://www.pecosproject.org/software.html

From this perspective, data fusion [19–21] should be used by multimedia management to combine different medias (e.g., video streams and temperature) to obtain relevant information (e.g., fire detection) or to improve the quality of the data provided (e.g., noise reduction). Therefore, fusion of multimedia data is equivalent to cooperative fusion proposed by Durrant-Whyte [22] where different nodes contribute with different data (e.g., video and audio streams) to accomplish a common task (e.g., intrusion detection). In addition, simple data aggregation schemes [11, 23, 8] can be used to summarize, organize and retrieve data in a power-efficient manner.

Figure 2 depicts some possible components in a multimedia management application for WSNs. The Stream Organizer receives multiple data streams through the same communication channel, then it organizes and directs them to the appropriate processing module. Three levels of processing are identified. The first level is the Stream Processing, which can perform low level data fusion. The second level is represented by the Feature Extractor which fuses several data streams to obtain relevant features from the environment. The last level is represented by the Decision Making where action plans are formulated in response to a identified situation. According to the application requirements or computational restrictions, the application can encompass only the desired processing levels (stream processing, feature extraction or decision making). This is also depicted in Figure 2 where the user interface is connected to all processing levels. In addition, the user can set properties and parameter related to the components responsible for each level of processing.

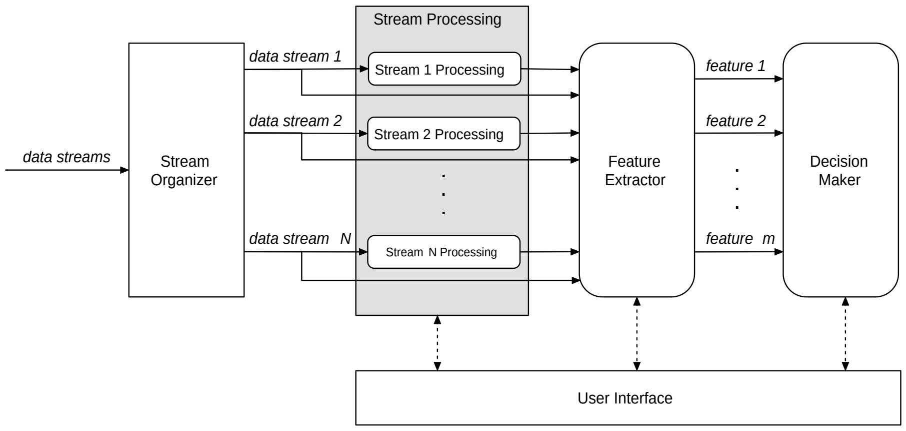


**Fig. 2.** Possible application involving multimedia management.

Considering our precision agriculture example with different data streams (soil temperature, acidity and humidity), which can be received in any order through the same communication channel. Thus, we need a component to organize the data streams properly. Further processing can be executed on these streams, such as data fusion algorithms like Kalman Filter [24], Particle Filter [25] or the Marzullo function [26] to obtain more accurate values. If desired, these data can be fused again to extract characteristics, like *the soil conditions that are favorable to cultivation of "X"* or *whether the soil is dry*. Yet further data fusion can be applied to obtain decisions, such as *"X" should be harvested in one week* or *irrigation required* or *freezing threat is eminent*. Multimedia fusion (concerned with feature extraction and decision making) is strongly coupled to the application and the type of data provided by sensors. Thus, the reusability of a multimedia fusion component might be limited to a specific domain of application and sensors. In addition, the development of such applications, and a tool to generate them, is still subject to the particularities of WSNs.

## 2.2 Attending WSN Requirements

Traditionally, system design can be organized in a Logical Layer Architecture (LLA) were the system is divided in different abstraction levels ranging from a physical layer, that deals with computing devices, to an application layer, that deals with business requirements. Usually, the LLA model is used in a bottom-up approach. However, WSNs are application-driven, and as discussed in [10], a top-down approach is preferable since once the business issues are understood, the requirements of lower levels become clearer.

Clearly, different applications tend to present distinct features and restrictions. Thus, it is not feasible to provide a unique Application Programming Interface (API) that is self-contained. Instead, we should think of providing small software components that represent elementary functionalities useful for various applications.

Regarding the applications illustrated in Figure 2 and using the top-down approach, we consider first the elements of the user interface that are related to multimedia management applications for WSNs. As we consider physical measurements acquired by sensors, it is reasonable to provide visual components appropriate to display common measurements, such as, temperature, pressure, acidity and humidity. Thus, BeanWatcher provides some visual components (e.g., thermometer, speedometer, gauge, and valued maps) to cover different types of sensory data.

The next steps should try to identify features in data streams. As mentioned before, this task depends on the application objectives, and, thus, the tool does not provide special components for them, since its reusability is restricted. For stream processing, there are some popular fusion algorithms that can be applied such as Kalman Filter, Particle Filter or Marzullo function. The current version of the tool provides two components that implement the Kalman Filter and the Marzullo function.

At the lower levels we need a component to receive data streams and to properly organize them. To get data from sensors we implement a communication component that receives and sends data without worrying about their semantics. Again, as we might have different types of sensors, we cannot provide a unique stream organizer. Besides, same sensors (data streams) can be semantically distinct in two different applications.

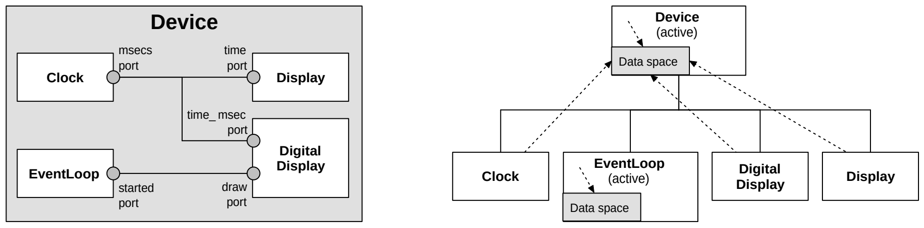
In summary, our API comprises elements for visual display, low level data fusion algorithms and communication operations. We chose to use a component model to implement this API and to generate the applications using our tool. The adoption of a component model allows us to take advantage of code reusability.

## 3 Component Model

BeanWatcher adopts the PECOS component model proposed by Genssler et al. [27], and provides a communication component to allow the monitoring of remote applications. In this section, we briefly describe PECOS and how it is used in BeanWatcher.

### 3.1 PECOS Component Model

The PECOS component model aims the design of embedded systems, more specifically field devices [27], which are executed directly by the instruments. PECOS is divided into two sub-models: structural and execution. Structural sub-model defines the entities included in the model, their features and properties. The execution sub-model defines the semantics of the components execution. An example of this model for a clock application is depicted in Figure 3, which is further discussed.



(a) Structural sub-model. (b) Execution sub-model.

**Fig. 3.** PECOS component model for a clock application.

**Structural Sub-Model** There are three main entities in PECOS structural sub-model: components, ports and connectors. Each component has a semantics and a well-defined behavior. Components form the model kernel and are used to organize both data and computation of the generated application. In the example shown, the components are *device*, *clock*, *display*, *eventloop* and *digital display*. Ports provide an interaction mechanism among components. Output ports are connected to input ports through connectors as illustrated in Figure 3a, that shows the output ports *msecs* and *started*, and input ports *time*, *time_milsecs*, and *draw*. Connectors describe a data sharing a relationship between two ports and are represented by lines connecting them.

Components in PECOS can be primitive and composed. A primitive component can be passive, active or an event. Passive components cannot control their execution, and are used as part of the behavior of another component being executed synchronously. In Figure 3a, the passive components are *clock*, *display* and *digital display*. Active components control their execution, which is triggered by a system request. In Figure 3a, the active components are *device* and *eventloop*. Event components are similar to active components, but their executions are triggered by an event. A composite component is built using connected sub-components, but their internal sub-components are not visible to the user. In addition, a composite component must define a scheduling with the exact execution order of its sub-components. In Figure 3a, the composite component is the *device*.

Ports can be input, output or input/output ports. Input ports just receive data from other components. Output ports just send data to other components. Input/Output ports are bi-directional receiving and sending data from and to other components.

PECOS also defines properties and a parent component. Properties are component's meta-data such as memory usage or execution time. The structure of a component generated by the model is always hierarchical where the top component is always a composite component (parent).

**Execution Sub-Model** PECOS provides a sub-model for the execution of applications, that shows how data are synchronized among components running in different threads and describes their semantics.

Problems of data synchronism may happen in PECOS. For instance, suppose there are two active components connected to each other through a port. Both can read and write data simultaneously by different operations. To solve this problem, active and event components have a private data space where they can update unconditionally and periodically a private data that can be synchronized with a parent component. In Figure 3b we can see the private data space in the *device* and *eventloop* components.

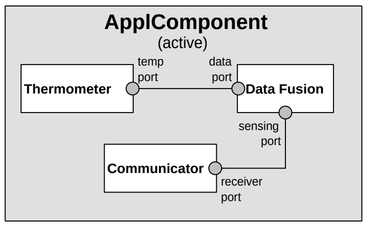
Due to this need for synchronization, active and event components have two possible behaviors: execution and synchronization. The execution behavior defines the actions performed when the component is executed. The synchronization behavior specifies how the private data space is synchronized with the parent component (arrows in Figure 3b). The execution semantics obeys the following simple rules:

- Execution behavior of a passive component is executed by a thread of its parent component;
- Synchronization behavior is executed by a thread of its parent component;

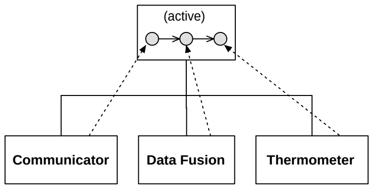
– Active and event components execute their sub-components using a control thread;
– Each component has a scheduler for its children.

### 3.2 PECOS in BeanWatcher

BeanWatcher adopts PECOS as its component model adding a communication component to support remote monitoring applications for wireless sensor networks. As an example, consider a WSN and a temperature application with three components: one to present the temperature, one to perform data fusion and one for communication, as depicted in Figure 4a.



(a) Structural sub-model.



(b) Execution sub-model.

**Fig. 4.** Using PECOS in BeanWatcher.

In the structural sub-model of BeanWatcher, every component that monitors data provided by the WSN (e.g., a thermometer) is an active component since it just receives data that is presented to the user. Also the *ApplComponent* is always an active component. Components used as alarm indicators are event components. Internal components (used by a parent component) are passive. In Figure 4a, the *Data Fusion* and *Communicator* components are passive. In addition, we added a *show* functionality to the components because the generated remote applications show raw or pre-processed data from the WSN.

The execution sub-model of BeanWatcher adopted a pipeline execution, i.e., only one component can be executed at a time. This is illustrated in Figure 4b where *ApplComponent* executes sequentially its sub-components.

BeanWatcher gives at least two benefits to the user by adopting PECOS. First, the component model allows the code reuse. Second, PECOS specifies that the interaction of two components must be done through input and output ports; this interaction simplifies the understating of the application that is constructed by connecting input ports to output ports.

## 4 BeanWatcher Architecture

Figure 5 shows the BeanWatcher architecture that has three different modules: repository, processing and presentation.

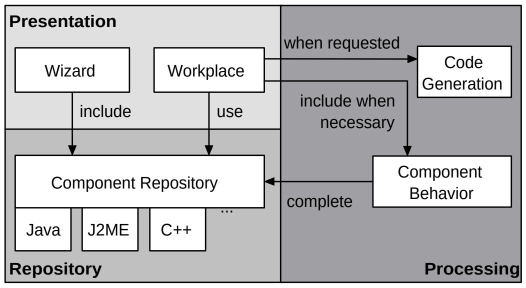


**Fig. 5.** BeanWatcher architecture.

The repository includes all components generated by a wizard that can be used to develop the application. It is important to mention that when a new target language is added to the tool, the components currently in the repository are automatically replicate to that target language so the user does not have to generate them again through the wizard. The repository uses the component model discussed in Section 3.2.

The presentation module specifies the interface provided by BeanWatcher to the user and it is composed of the workplace and the wizard. The workplace is the application building area and is divided into component edition and code edition. The component edition allows the user to choose and place the components according to the application requirements, adding a connector when two or more components demand it. The code edition is used to implement the behavior and show the functionality of the application being created. Furthermore, it allows the re-implementation of the components in the application.

The wizard enables the creation of new components extending the repository. To create a new component it is necessary to label the component and specify its ports. However, the behavior and functionalities must be implemented in the code edition.

The processing module performs the code generation and updates the components repository with new behavior implementations. The code generation obeys the application built by the user on the workplace. To have the complete application, the behavior of the composite component must be implemented.

In summary, the architecture depicted in Figure 5 presents some benefits to the user. First, it generates all the repetitive code to the user through the *code generation* unit. Second, as we mentioned in Section 2, many applications require unique features, so new components can be created through the *wizard* to attend unpredicted requirements; and the behavior of new or current components can be (re)implemented in the *workplace*. Third, it allows code reuse: once components are implemented the user can make it available in the *component repository* for future applications. Finally, the *component repository* allows the development of application in several target languages, such as Java, J2ME and C++.

## 5 Application Development Using BeanWatcher

In this section, we present an example using BeanWatcher.

### 5.1 The Three-Step Generation Process

Every application generated by BeanWatcher has a composite component called *ApplComponent*, i.e., every component added to the workplace will be a sub-component of *ApplComponent*. To build an application, the user must go through the three steps: building, implementation and generation (Figure 6).

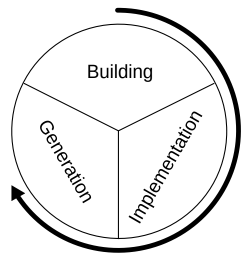


**Fig. 6.** The three steps to create a BeanWatcher application.

In the building phase, the user chooses one or more components to be added to the component edition area in the workplace. Thus, the added component becomes a sub-component of the *ApplComponent*. If necessary, a connector can be added to allow interactions among components.

Once the application is built, it is necessary to implement the behavior and the visualization of the *ApplComponent*. This is done in the code edition area in the workplace. All the functionalities and features of this component are generated automatically and are chosen by the user (e.g., implementation language). At this point the user can modify the behavior of the sub-components selecting the desired sub-component and editing it in the code edition area.

When the user saves the application, the code of each sub-component in the component edition area, the *ApplComponent*, and a default presentation code of BeanWatcher are automatically generated, including repetitive code. After that, the user can compile and run the generated application.

### 5.2 Developing an Application

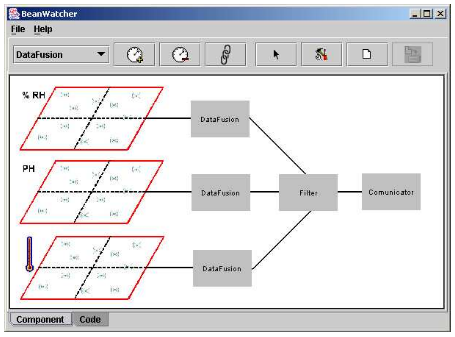
The application presented in this work is for precision agriculture, where a WSN is used to monitor the soil conditions of a orchard of strawberries. Sensors will be used to capture the soil temperature, humidity and acidity of the orchard. For the sake of simplicity, our application will be restricted to stream processing using the Kalman Filter, and no feature extraction nor decision making will be performed by the application (see Figure 2).

Initially, we chose the components from the menu (temperature map, data fusion and communicator). The temperature map is an active component that shows the temperature provided by the sensors and represents the area delimited by the sensors and is divided into four quadrants that show the respective fused values; if the user wants the quadrants can be zoomed in. The data fusion is a passive component based in the Kalman Filter.

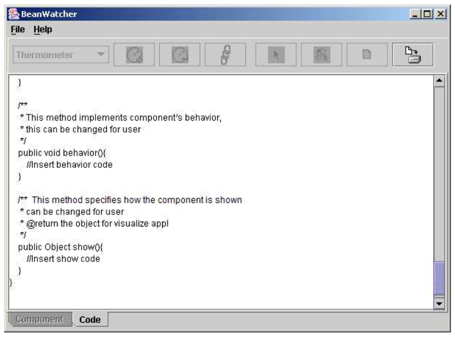
The same procedure is repeated using the humidity and acidity maps that will show the other fused measures.

The communicator must be added to get the data streams and it is a passive component that allows the communication between *sink* and *sources*. Finally, a filter component is added to organize the received streams (this component is implemented by the designer).

After the addition of the components above, it is necessary to connect them. *Connector* allows the data forwarding from the communicator to the filter, and then to data fusion, and so on. The application available in the component edition area is shown in Figure 7a.



(a) Workplace.



```
  }

  /**
   * This method implements component's behavior,
   * this can be changed for user
   */
  public void behavior(){
     //Insert behavior code
  }

  /**  This method specifies how the component is shown
   * can be changed for user
   * @return the object for visualize appl
   */
  public Object show(){
     //Insert show code
  }
}
```

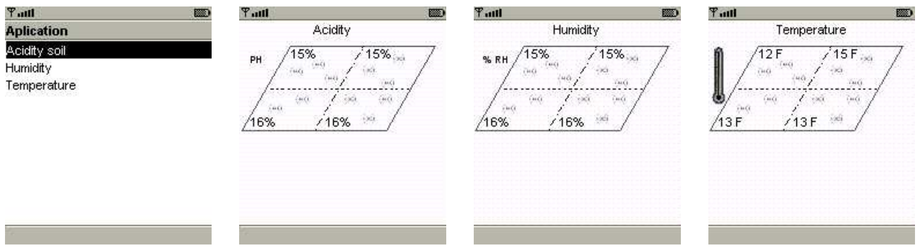
(b) Code edition area.

**Fig. 7.** Development environment of BeanWatcher.

Once the application is built, the next step is the implementation. For the filter component, we considered that packets sent by sensors indicate the geographic position, the type of data (temperature, acidity or humidity) and the associated measurements. Thus, the filter only directs the data to the appropriate output port. Also, the behavior and *show* of the *ApplComponent* need to be implemented. This is done through the code edition area shown in Figure 7b. Finally, after saving the project we have the application ready. This example was executed in a simulator[6] and can be visualized in Figure 8. Note that we can

[6] The simulator is included in the J2ME Wireless Toolkit freely available at http://java.sun.com/j2me/.

add different graphical elements to the *show* method improving the presentation of the application.



(a) Menu. (b) Acidity. (c) Humidity. (d) Temperature.

**Fig. 8.** Execution of the application.

The application depicted in Figure 8 is presented as a menu where the user selects the type of data to be collected. All the three measurements are presented in a map, which is divided into four quadrants. The values represent the fusion of the measurements provided by the sensors at each quadrant. This approach is used for temperature, humidity and acidity streams.

## 6 Conclusions and Future Work

BeanWatcher is a powerful and flexible tool that adopts a component model aiming the development of management and monitoring applications for WSNs. BeanWatcher was designed to allow users with limited programming experience to develop a wide variety of monitoring applications. In addition, it supports several target languages.

Currently we are working on an extension module that uses PECOS to develop the applications that are executed into the sensor nodes so we can develop applications to both *sink* and *sources*. Thus, once data fusion and management components are added to the BeanWatcher repository, they can be used to build applications to *sink* and *sources* nodes so data fusion and management can be done in-network.

Finally, another module is being developed to generate applications based on rules, i.e., if the *sink* node is able to receive different sensory data, the application will show such data according to a set of rules so the user do not need to interact with the application.

As a future work we plan to implement a data fusion component to combine different types of media, such as video and audio streams to accomplish a more important task such as intrusion detection in a closed environment. However, as stated in Section 2, the reusability of such components is restricted to specific application domains and sensors.

## References

1. Estrin, D., Girod, L., Pottie, G., Srivastava, M.: Instrumenting the world with wireless sensor networks. In: International Conference on Acoustics, Speech, and Signal Processing, Salt Lake City, USA (2001)
2. Pottie, G.J., Kaiser, W.J.: Wireless integrated network sensors. Communications of the ACM **43** (2000) 51–58
3. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: Scalable coordination in sensor networks. In: Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCom'99), Seattle, Washington, USA, ACM Press (1999)
4. Hill, J., Culler, D.: Mica: A wireless platform for deeply embedded networks. IEEE Micro (2002) 12–24
5. Baard, M.: Wired news: Making wines finer with wireless. [online] available: http://www.wired.com/news/wireless/0,1382,58312,00.html, access: March 2003 (2003)
6. Schurgers, C., Tsiatsis, V., Ganeriwal, S., Srivastava, M.B.: Topology management for sensor networks: Exploiting latency and density. In: 2002 ACM Symposium on Mobile Ad Hoc Networking & Computing (MobiHoc'02), Lausanne, Switzerland (2002)
7. Chen, B., Jamieson, K., Balakrishnan, H., Morris, R.: Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. Wireless Networks **8** (2002) 481–494
8. Krishanamachari, B., Estrin, D., Wicker, S.: The impact of data aggregation in wireless sensor networks. In: Proceedings of the International Workshop of Distributed Event Based Systems (DEBS), Vienna, Austria (2002)
9. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom'00), Boston, MA, USA, ACM Press (2000) 56–67
10. Ruiz, L.B., Nogueira, J.M., Loureiro, A.A.F.: Manna: A management architecture for wireless sensor networks. IEEE Commmunications Magazine **41** (2003) 116–125
11. Zhao, Y.J., Govindan, R., Estrin, D.: Computing aggregates for monitoring wireless sensor networks. In: Proceedings of the 1st IEEE International Workshop on Sensor Network Protocols and Applications (SNPA'03), Anchorage, AK, USA (2003)
12. Zhao, J., Govindan, R., Estrin, D.: Residual energy scans for monitoring wireless sensor networks. In: IEEE Wireless Communications and Networking Conference (WCNC'02), Orlando, FL, USA (2002)
13. Mini, R.A.F., Nath, B., Loureiro, A.A.F.: A probabilistic approach to predict the energy consumption in wireless sensor networks. In: IV Workshop de Comunicação sem Fio e Computação Móvel, São Paulo, SP, Brazil (2002)
14. Tian, D., Georganas, N.D.: A coverage-preserving node scheduling scheme for large wireless sensor networks. In: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications, Atlanta, Georgia, USA, ACM Press (2002) 32–41
15. Chakrabarty, K., Iyengar, S.S., Qi, H., Cho, E.: Grid coverage for surveillance and target location in distributed sensor networks. IEEE Transactions on Computers **51** (2002) 1448–1453
16. Meguerdichian, S., Koushanfar, F., Potkonjak, M., Srivastava, M.: Coverage problems in wireless ad-hoc sensor networks. In: Proceedings of IEEE Infocom 2001. Volume 3., Anchorage, AK, USA (2001) 1380–1387

17. Megerian, S., Koushanfar, F., Qu, G., Veltri, G., Potkonjak, M.: Exposure in wireless sensor networks: Theory and practical solutions. Wireless Networks **8** (2002) 443–454
18. Meguerdichian, S., Slijepcevic, S., Karayan, V., Potkonjak, M.: Localized algorithms in wireless ad-hoc networks: Location discovery and sensor exposure. In: Proceedings of the 2001 ACM International Symposium on Mobile Ad Hoc Networking & Computing, Long Beach, CA, USA, ACM Press (2001) 106–116
19. Luo, R.C., Yih, C.C., Su, K.L.: Multisensor fusion and integration: Approaches, applications, and future research directions. IEEE Sensors Journal **2** (2002) 107–119
20. Brooks, R.R., Iyengar, S.S.: Multi-Sensor Fusion: Fundamentals and Applications. Prentice Hall, New Jersey, USA (1998)
21. Hall, D.L.: Mathematical Techniques in Multisensor Data Fusion. Artech House, Norwood, Massachusetts, USA (1992)
22. Durrant-Whyte, H.F.: Sensor models and multisensor integration. International Journal of Robotics Research **7** (1988) 97–113
23. Dasgupta, K., Kalpakis, K., Namjoshi, P.: Improving the lifetime of sensor networks via intelligent selection of data aggregation trees. In: Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference, Orlando, FL, USA (2003)
24. Brown, R.G., Hwang, P.Y.: Introduction to Random Signals and Applied Kalman Filtering. 2nd edn. John Wiley & Sons, New York, NY, USA (1992)
25. Arulampalam, S., Maskell, S., Gordon, N., Clapp, T.: A tutorial on particle filters for on-line non-linear/non-gaussian bayesian tracking. IEEE Transactions on Signal Processing **50** (2002) 174–188
26. Marzullo, K.: Tolerating failures of continuous-valued sensors. ACM Transactions on Computer Systems (TOCS) **8** (1990) 284–304
27. Genssler, T., Christoph, A., Schulz, B., Winter, M., Stich, C.M.., Zeidler, C., Mller, P., Stelter, A., Nierstrasz, O., Ducasse, S., Arevalo, G., Wuyts, R., Liang, P., Schonhage, B., van den Born, R.: Pecos in a nutshell. Technical report, The Pecos Consortium (2002)