Semi-Automatic Generation of Monitoring Applications for Wireless Networks

André Lins*, Eduardo F. Nakamura*†, Lincoln S. Rocha‡, Antonio A.F. Loureiro*,
Claudionor J.N. Coelho Jr.*
*Department of Computer Science, Federal University of Minas Gerais, Brazil
Email: {alla,nakamura,loureiro,coelho}@dcc.ufmg.br
†Department of Technological Development, FUCAPI – Research and Technological Innovation Center, Brazil
Email: nakamura@fucapi.br
‡Department of Computer Science and Statistics, Federal University of Piaui, Brazil
Email: lincoln@ufpi.br

***Abstract*— In this paper we present a new tool called BeanWatcher. This tool allows the semi-automatic generation of monitoring and management applications for wireless networks such as WLANs and WPANs. 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.**

I. Introduction

A Wireless Network is a special type of network that uses radio waves to communicate between nodes. Recent standard approvals for WLANs – Wireless Local Area Networks (IEEE 802.11), WPANs – Wireless Personal Area Networks (IEEE 802.15 and Bluetooth), and Broadband Wireless Access (IEEE 802.16) [1] allowed users to wirelessly extend their networks in different scenarios such as schools, enterprises, and industries. It is estimated that 16 million people used 802.11 in 2001, and this number will grow to 60 million by 2006 [2].

Networks and devices are important development considerations when developing an application for a wireless environment. On the other hand, if an application will be used in different wireless networks and portable devices, it is important to design it in such a way that code can be reused, and network and hardware requirements be encapsulated, so its development and maintenance will be easier.

In this paper we will use the following example, depicted in Fig. 1. Consider a factory equipped with a wireless network that gathers different types of data from the environment such as machinery temperature and speed, and power consumption. A walking employee can receive data from those equipments through a wireless network interface into a portable device. The management application running on a portable device could be based on J2ME [3] or SuperWaba [4] platform. In this case, a multimedia interface, with graphics, animations and audio/video stream capability, to exhibit data collected from different equipments could be provided to support the network monitoring and management.

The application described above is usually designed to solve a specific issue, and thus, no model is applied and no code reuse is foreseen. 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 a wireless network in a semi-automatic fashion that takes into account the main characteristics of both the wireless environment (e.g., lower bandwidth and higher error rate) and the portable device (e.g, screen size and computational capacity). These applications are intended to be run on portable and mobile devices.

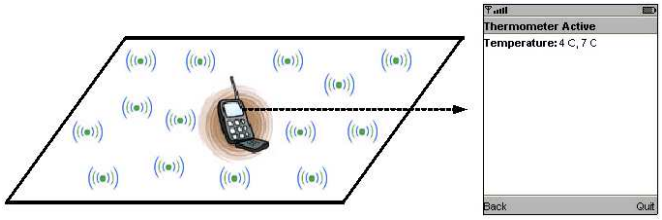

Fig. 1. A wireless network example.

Some commercial tools like LabView [5] and HP VEE [6] were designed to develop applications to monitor and act on instruments. However, these tools are proprietary solutions hardly integrated with other tools and languages. Furthermore, these tools do not allow the development of applications to portables devices neither monitoring applications for wireless networks. Other related tool is the PECOS Component Environment [7] that allows the development of monitoring and actuation applications. Unfortunately, PECOS does not support remote communication and it was not designed for applications to portable devices. In BeanWatcher, applications are modelled and built based on the PECOS component model. In addition, a communication component (based on the standard socket API) is provided to allow the application interaction with other computing devices.

This paper is organized as follows. In Section II we present the component model used in BeanWatcher. In Section III we

present the tool BeanWatcher discussing its architecture and use model. Section IV shows how BeanWatcher can be used to develop applications and new components. Section V presents our conclusions and future work.

II. Component Model

BeanWatcher adopts the PECOS component model proposed by Genssler et al. [8]. However, BeanWatcher adds a communication component to allow the monitoring of remote applications.

A. PECOS Component Model

The PECOS component model aims the design of embedded systems, more specifically field devices, 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 Figures 2 and 3 which will be further discussed.

1) 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 that is used to organize both data and computation of the generated application. In the example shown in Fig. 2, 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 Fig. 2, that shows the output ports *msecs* and *started*, and input ports *time*, *time_milsecs*, *draw*. Connectors describe a data sharing relationship between two ports and are represented by lines connecting the ports.

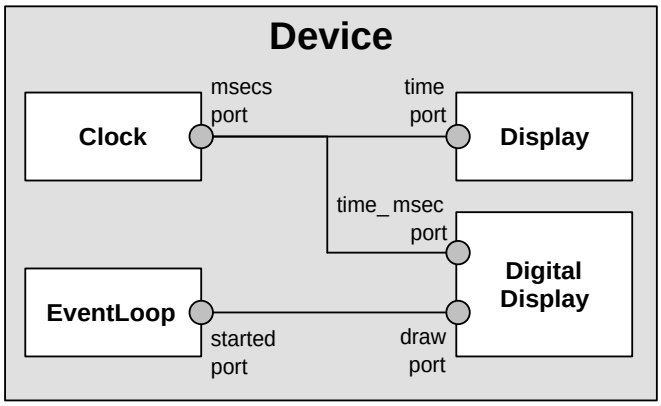

Fig. 2. PECOS component model for a clock application (structural sub-model).

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 other component being executed synchronously. In Fig. 2, the passive components are *clock*, *display* and *digital display*. Active components control their execution which is trigged by a system request. In Fig. 2, 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 specifying the exact execution order of its sub-components. In Fig. 2, 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 components 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).

2) 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 the 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 Fig. 3 we can see the private data space in the *device* and *eventloop* components.

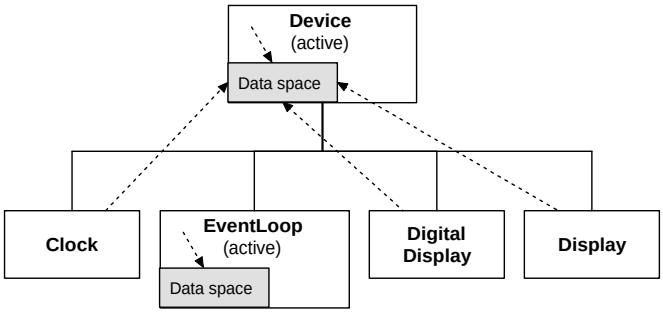

Fig. 3. PECOS component model for a clock application (execution sub-model).

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 Fig. 3). 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 into their own behavior using a control thread;
- Each component has a scheduler for its children.

B. PECOS in BeanWatcher

As mentioned before, BeanWatcher adopts PECOS as its component model adding a communication component to support remote monitoring applications for wireless networks. As an example, consider a wireless network and a temperature application with three components: one to present the temperature, one to perform data fusion and one for communication, as depicted in Figures 4 and 5.

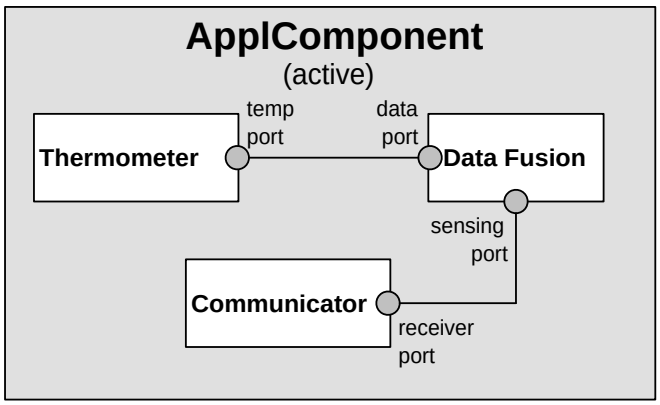

Fig. 4. Using PECOS in BeanWatcher (Structural sub-model).

In the structural sub-model of BeanWatcher, every component that monitors data provided by the wireless network (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 Fig. 4, 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 wireless network.

The execution sub-model of BeanWatcher does not use the thread concept. We adopted a pipeline execution, i.e., only one component can be executed at a time. This is illustrated in Fig. 5 where ApplComponent executes sequentially its sub-components.

III. BeanWatcher Tool

BeanWatcher is a graphic tool based on components designed to the development of monitoring and management applications for wireless networks. The tool allows the development of applications by a user with limited programming experience. Furthermore, BeanWatcher allows the generation of applications for different platforms if the appropriated component repository is available. In our case study we used

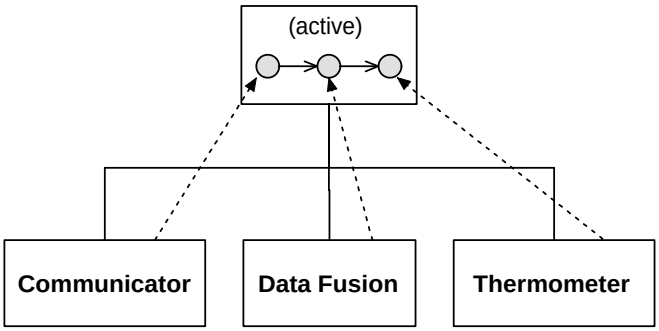

Fig. 5. Using PECOS in BeanWatcher (Execution sub-model).

a J2ME [3] component repository to develop monitoring applications for wireless networks to portable devices such as palms and cell phones.

A. BeanWatcher Architecture

Figure 6 shows the BeanWatcher architecture which has three different modules: repository, processing and presentation.

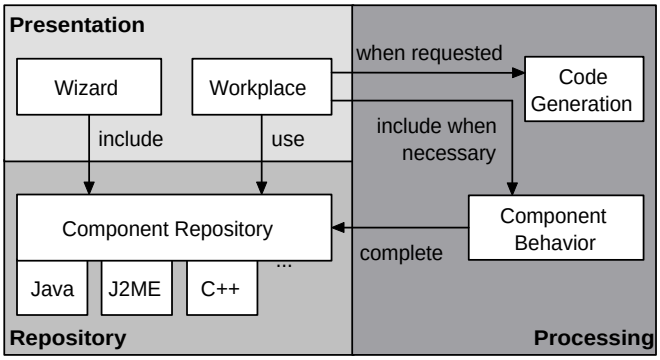

Fig. 6. BeanWatcher architecture.

The repository includes all the components, generated by a wizard, that can be used to develop the application. For the sake of flexibility, Beanwatcher allows the addition of several target languages to the repository, such as Java, C/C++ and Embedded C. These target languages are associated with the repository module and enables the creation of applications in different languages. The repository uses the component model discussed in Section II.

The presentation module specifies the interface provided by BeanWatcher to the user and it is composed by 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 connectors 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 is an important part of the presentation module that 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, both the behavior and show functionalities must be implemented in the code edition. It is important to mention that when a new target language is added to the tool, the components currently in the repository can be replicated by the wizard so the components can be available in the new language. This is possible because the wizard stores all the information necessary to create the components in a XML file. Thus, to add a new language the user only needs to create a new template for that language Fig. 7 shows a template for Java.

```
public class <_component_name>{

     <atribute_in>
    /** This variable input for this component
     *  used when component receiver some data.*/
     private <_type_input_port_attribute>
             <_name_input_port_attribute>InPort<_dimension_input_port>;
     </atribute_in>

    ...

     <methods_in_pv>
    /** This method is to get output for this component, is private because
      * the component has one outport.
      * @return return the data in port value.
      */
     private <_type_input_port_attribute><_dimension_input_port>
             get<_name_input_port_method>InPort (){
         return(this.<_name_input_port_attribute>InPort);
     }
     </methods_in_pv>

     ...

     public void behaviour() {
     }

     public Object show(){
       return new Object();
     }
}
```

Fig. 7. Template to create Java applications.

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.

B. Application Development in BeanWatcher

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 create an application, the user must go through the three steps: building, implementation and generation.

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 attributes and methods of this component are generated automatically. 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.

Once 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 is automatically generated. After that, the user can compile and run the generated application.

IV. Using BeanWatcher

In this section is devoted to the use of BeanWatcher. First, we show how to create a component through the wizard. Then, we describe how to replicate the repository for a new target language. At last, an exemple application is built using BeanWatcher.

The example application monitors the temperature data provided by a wireless network. As we wanted to perform data fusion, we will create through the wizard the data fusion component, which implements the Marzullo function [9]. The application and the components will be created using J2ME.

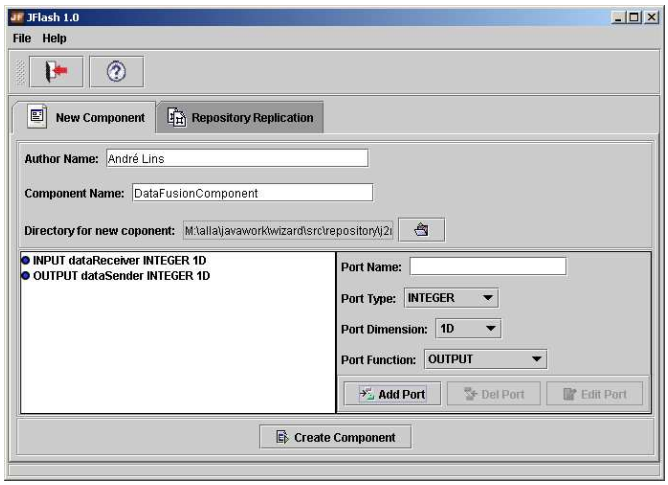

Fig. 8. Creating a new component through the wizard.

A. Creating a New Component

Initially, to create the component we need to specify the interface of the component, i.e., the input and output ports, the associated data types and the proper dimension. This task is performed in the wizard interface depicted in Fig. 8. First, we must specify the author's name and the component's name. Second, the repository file[1] should be indicated. Third,

[1]The repository file contains the specification of all components; it is used to replicate the repository whenever necessary or desired.

the component's ports are specified. Finally, the components are actually created in all target languages available in the repository.

After the creation of the component, BeanWatcher automatically generated every code that was possible. However, we must implement the behavior[2] of the component, which in this case is the Marzullo function. This can be done in the code edition area (Fig. 9).

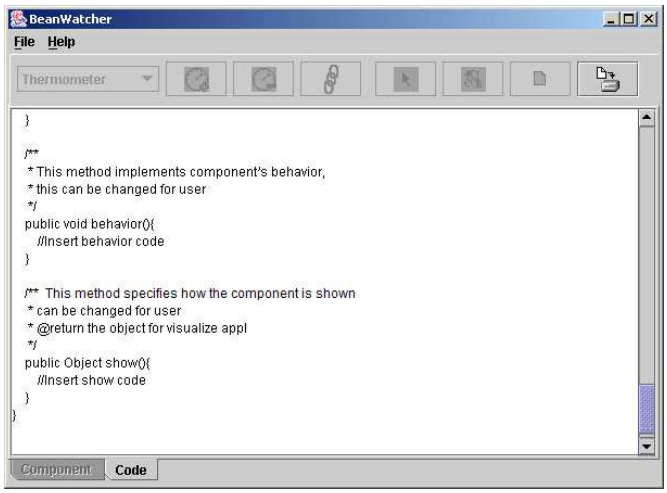

Fig. 9. Code edition area.

B. Replicating The Repository

The specification of all components in the repository is stored in a XML file where the tags are used to describe the interface of a component (Fig. 7). When a new target language is added, the repository needs to be replicated so all components can be available in that language. This repository file is used by BeanWatcher to automatically generate the code for all components in a desired language. However, particularities of each component, such as the behavior and *show* functionalities, still requires implementation.

The replication can be performed in the wizard interface depicted in Fig. 10. First, we have to choose the new target language. Then, the location of the repository file must be specified. Finally, we must indicate the location were the components will be stored and start the replication. Again, the implementation of the behavior and *show* functionalities still needs to be implemented in the new target language.

C. Creating an Application

Now that we have all the necessary components the application can be built. Initially, we chose the components from the menu (thermometer, data fusion and communicator). The thermometer is an active component that shows the temperature provided by the network equipments. The data fusion is a passive component based in the Marzullo function. However, as BeanWatcher allows the optimization of the component

[2]In the case of graphical components, the user also needs to implement the *show* functionality.

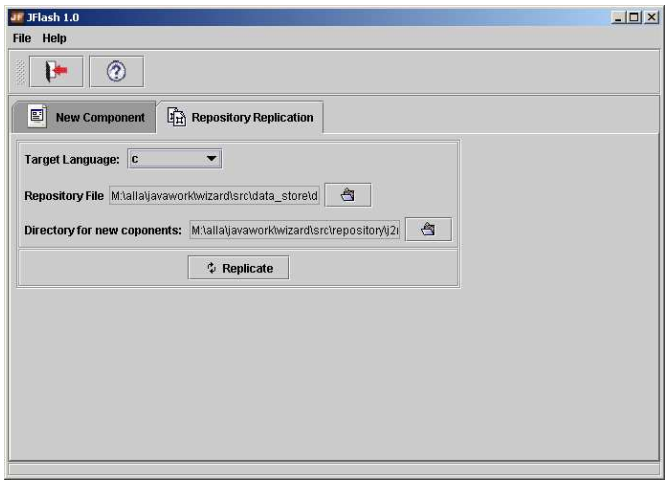

Fig. 10. Replicating the repository.

behavior, new algorithms can be implemented. The communicator is a passive component that allows the communication among the *sink* and *sources*. These components were added to the component edition area.

After the addition of the components it is necessary to connect them. One connector allows the data forwarding from the communicator to the data fusion component. The other connector allows the thermometer to receive and show the data previously fused. The application available in the component edition area is shown in Fig. 11.

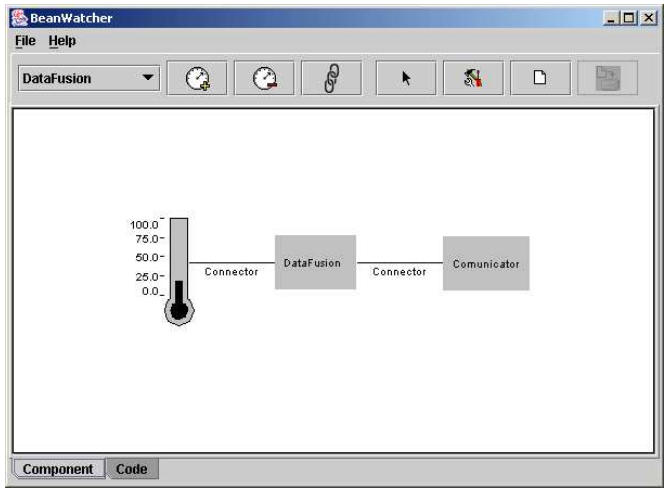

Fig. 11. Workplace.

Once the application is built the next step is the implementation of the behavior and *show* of the *ApplComponent*. This is done through the code edition area shown in Fig. 9. Finally, saving the project we have the application ready. This example was executed in a simulator included in the J2ME Wireless Toolkit [3] and the result can be visualized in Fig. 12. Note that we can add different graphical elements into the *show* method improving the presentation of the application.

(a) Application presentation.

(b) Application waiting for data.

(c) Fused data is presented to the user.

Fig. 12. Execution of the generated application.

V. Conclusions and Future Work

BeanWatcher is a powerful tool that adopts a component model aiming the development of management and monitoring applications for wireless networks. 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.

As a future work, we will develop a module to generate applications based on rules, i.e., if different types of data are received, the application will automatically choose the appropriate interface according to a set of rules simplifying its use. In this case, the application will be composed by a set of synoptic panels that will be automatically chosen based on the data received.

Acknowledgment

This work has been partially supported by CNPq, Brazil, under process 55.2111/2002-3.

References

[1] IEEE, “Wireless standards zone. [online] available: http://standards.ieee.org/wireless/.”

[2] B. Morrissey, “802.11 takes center stage. june, 2002. [online] available:http://www.80211-planet.com/news/article.php/1355221.”

[3] S. Microsystems, “J2me. [online] available: http://java.sun.com/j2me/, access: March 2003.”

[4] G. C. Hazan, “Jsuper waba. [online] available: http://www.superwaba.com.br, access: March 2003.”

[5] N. Instruments, “Labview 6.1. [online] available: http://www.ni.com/, access: January 2003.”

[6] Agilent, “Vee onelab 6.1. [online] available: http://www.agilent.com/, access: January 2003.”

[7] Pecosproject, “Composition enviroment. [online] available: http://www.pecos project.org/software.html, access: January 2003.”

[8] T. Genssler, A. Christoph, B. Schulz, M. Winter, C. M. . Stich, C. Zeidler, P. Mller, A. Stelter, O. Nierstrasz, S. Ducasse, G. Arevalo, R. Wuyts, P. Liang, B. Schonhage, and R. van den Born, “Pecos in a nutshell,” The Pecos Consortium, Tech. Rep., September 2002.

[9] K. Marzullo, “Tolerating failures of continuous-valued sensors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 8, no. 4, pp. 284–304, 1990.