Phybots: A Toolkit for Making Robotic Things

Jun Kato[1&2] **Daisuke Sakamoto**[1&2] **Takeo Igarashi**[1&2]
[1]The University of Tokyo, Tokyo, Japan [2]JST ERATO, Tokyo, Japan
{jun,kato | d.sakamoto | takeo}@acm.org

ABSTRACT
There are many toolkits for physical UIs, but most physical UI applications are not locomotive. When the programmer wants to make things move around in the environment, he faces difficulty related to robotics. Toolkits for robot programming, unfortunately, are usually not as accessible as those for building physical UIs. To address this interdisciplinary issue, we propose Phybots, a toolkit that allows researchers and interaction designers to rapidly prototype applications with locomotive robotic things. The contributions of this research are the combination of a hardware setup, software API, its underlying architecture and a graphical runtime debug tool that supports the whole prototyping activity. This paper introduces the toolkit, applications and lessons learned from three user studies.

Author Keywords
Toolkits, prototyping, and robotic things.

ACM Classification Keywords
H5.2 [Information interfaces and presentation]: User Interfaces – prototyping. I.2.9 [Artificial Intelligence]: Robotics – commercial robots and applications.

INTRODUCTION
What if you could just relax on the sofa and your speakers would automatically move to the position that sounds best for you? What if your alarm clock runs away from you while it is ringing in the morning? Would not everyday life be more interesting when you can add some degree of animacy to everyday things and have them move around in your everyday space?

In the research area of Human-Computer Interaction and robotics, such applications are respectively called "Physical User Interfaces" and "Robots." There are many toolkits for prototyping physical UIs and robots, but their coverage is usually one-sided (Table 1). Toolkits for physical UIs provide hardware and software building blocks for building physical UIs, but the application programming interfaces (APIs) are usually thin wrappers around hardware components. They do not provide support for the time-consuming task of integrating sensor input and actuator output; e.g., to move to a specified position by controlling differential wheels according to camera input. As a result, most of the resulting applications are not locomotive and are fixed to a specific position in the environment.

On the other hand, while many toolkits exist to create robots that move around in the real world, most of them focus on the development of a reliable robot whose hardware components tend to be expensive, with many precise sensors and actuators. They are mainly interested in making robots perform tasks automatically and providing support to implement basic capabilities such as localization, mapping, and recognition. Therefore, prior knowledge of robotics is required for the programmer to choose and configure appropriate modules.

The goal of our research is to enable the rapid prototyping of user experiences with locomotive robotic things. By robotic things, we mean those things that have some degrees of freedom to move around in the real world, such as physical icons with mobility on a tabletop interface [15],

Toolkits for:	Physical UIs	Robotic Things	Robots
Target users:	HCI researchers and interaction designers		Robotics people
Focus:	Prototyping		Reliability
Software APIs:	Low-level and static	High-level and extensible	
Hardware size & cost:	Small & cheap (Phidget Kit $200, Ikimo Robot [9] $160)		Medium to large & expensive ($700~)

Table 1. Comparison between toolkits

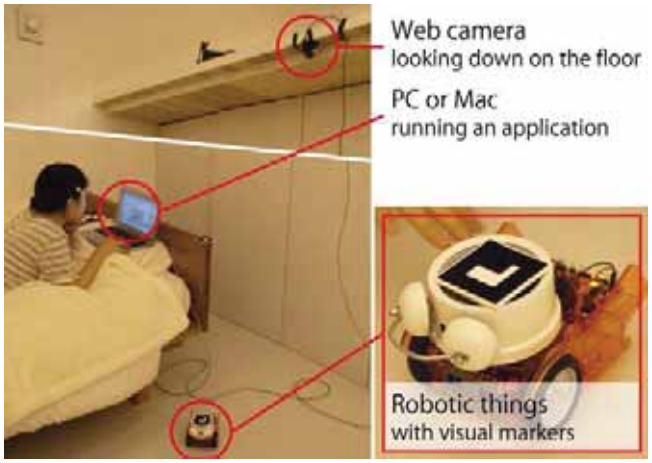

Figure 1. Overview of the prototyping environment

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DIS 2012, June 11-15, 2012, Newcastle, UK.
Copyright 2012 ACM 978-1-4503-1210-3/12/06...$10.00.

and small mobile robots that fold clothes [18] or cook meals [19]. To achieve this goal, we have designed "Phybots" as a simple setup of inexpensive hardware along with a software toolkit. Contributions of this research are the combination of (1) a hardware setup (Figure 1) and software APIs for two-dimensional locomotion, (2) an extensible software architecture, and (3) a graphical runtime debug tool, all of which are validated by three user studies.

First, Phybots assumes a simple and inexpensive hardware setup of a camera looking down on the floor and visual markers attached to the top of physical objects. This hardware setup can easily be deployed to everyday spaces such as a working desk, a dining table or in a kitchen or living room. Given the hardware setup, Phybots provides APIs for two-dimensional localization and locomotion of floor-based robotic things. These APIs are similar to those for graphical applications in which object positions are defined by screen coordinates, and support moving to a goal, pushing an object toward a goal, tracing a path, and other custom behavior by specifying a vector field.

Second, Phybots APIs are built on top of a centralized and extensible software architecture that runs on one personal computer and manages all physical objects of interest including cameras, robotic things, and entities with visual markers. The programmer has direct access to these objects as instances of the *Camera*, *Robot* and *Entity* interfaces, respectively. Time-consuming tasks including locomotion APIs are represented by *Task* interface. The toolkit can be easily extended on top of these software stacks to support other types of sensors, roobts, and tasks. In addition, a *Workflow* class represents a directed graph consisting of a set of *Robot* and *Task* instances, and enables higher-level task management such as handling multiple tasks by one robot in serial or by two robots in parallel.

Third, Phybots provides a *runtime debug tool* that supports a test phase, beyond the programming phase, as an essential part of the prototyping process. In the robot application test phase, when the robot fails to accomplish its task, the user usually has to order the same task to be performed again. Because it is too costly to restart a test run for each error, the programmer often wants to fix the error right away without restarting. To support this workflow, a graphical *runtime debug tool* enables API parameter reconfiguration, restart, pause, or resume of the interaction at run-time. For example, when the robotic thing fails to move to a desired destination, the programmer can change the threshold of the marker detection algorithm by dragging a slider and then give the robot another try.

RELATED WORK

APIs, software architecture, and a runtime debug tool of Phybots aim to solve the deficiencies of existing toolkits for robots and physical UIs, and we borrow good points from each.

Toolkits for Physical User Interfaces

Recent trends in HCI places an emphasis on physical interaction that requires new hardware configurations with sensors and actuators. Papier-Mache [11] is a toolkit for building tangible interfaces consisted of well-defined APIs, software architecture, and graphical tools for development support. Our toolkit takes this approach in the field between physical UIs and robotics. Phidgets [6] encapsulates a sensor or actuator as a package with a USB interface and provides software APIs to control the package. Phybots also abstracts hardware but provides higher-level task centric APIs. Arduino [1] lowers the threshold for embedded programming, which was used in our user study to build robots. While these toolkits only support the programming phase of the prototyping process, d.tools [7] additionally supports the test and analysis phases by recording and playing video of the user interaction, synchronized with transition visualization in the program components. Our toolkit also supports the test phase through runtime debug tools and differs from d.tools in that it allows configuring API parameters and testing the interaction scenario many times without restarting the application.

Toolkits for Robots

There are many robot programming toolkits with much overlap in their functionalities. Most of them can be categorized into one of three types according to their purposes.

The first type is middleware that abstracts hardware and provides functionality for message passing between software components in a distributed environment. Robot Operating System (ROS) [16] constructs a peer-to-peer network in the environment and abstracts general-purpose services including robots and sensors, which is similar to Decentralized Software Services of Microsoft Robotics Developer Studio (MRDS) [12]. On the other hand, Player [5] works as a proxy server to the robots. The programmer has virtually direct access to the connected devices with these toolkits. However, the connection requires some prior configuration that is not as easy as our case of presuming one host computer. Our toolkit is not designed to become a software platform in a distributed environment, but is intended to provide a lightweight development environment with less configuration.

The second type is a collection of algorithms studied in the specific research domain of robotics. For example, the Carnegie Mellon Robot Navigation Toolkit [13] provides basic localization and navigation functionality OpenCV [14] provides implementation of many vision-based algorithms, which is included in ROS. Though these toolkits free the programmer from detailed implementation, they are still required to learn related topics beforehand. In contrast, our toolkit supports fewer algorithms related to marker detection and two-dimensional locomotion, but provides enough functionality for the prototyping of robotic things. In addition, our APIs are a similar to those for

programming graphical UIs since we regard HCI researchers and interaction designers as our target users.

The third type are toolkits for educational and entertainment purposes, which aim to allow easy development of robot applications. These toolkits are similar to ours in that they enable non-experts to program robots, where the difference lays in the target users, applications, and the resulting methods of supporting them. For instance, LEGO Mindstorms [4] provides a set of hardware building blocks and a visual programming language (VPL) that enables young children to learn programming. It contains support for parallel execution of multiple tasks similar to our *Workflow* abstraction, but then cannot benefit from other software libraries since it is specially designed for the Mindstorms hardware. Topobo [17] is a construction toy with built-in actuators whose motion can be specified by demonstration with the user's hands and so does not even need a development environment on; but then lacks extensibility just as the Mindstorms VPL. Our toolkit provides Java APIs for a desktop programming environment and can easily be connected with other input and output devices, such as a mouse, keyboard, camera, gamepad, voice, physical UIs, display, projector, and speaker. Pyro [2] helps students learn programming and artificial intelligence algorithms while providing locomotion APIs for Python that can specify the relative movement of a robot from a current position, but does not support the prototyping and debugging of practical applications as our toolkit does.

PROTOTYPING SCENARIO

Before describing the concrete features of our toolkit, we introduce an example prototyping scenario to provide an overview of Phybots development. This scenario is based on one of our user study applications that makes an alarm clock run away from the user while it is ringing. Please note that the programmer often went backward and forward among these steps to iteratively complete their project.

The programmer first prepares the hardware, including a webcam, visual markers, a robotic thing, and a personal computer (PC) (Figure 1). An alarm clock is broken down and connected to Ikimo [9], an open-source mobile robot kit based on Arduino [1]. Basically, the programmer needs to prepare the robotic hardware, which is supported by physical UI toolkits or cheap robots. Specifically, Phybots has built-in support for Ikimo and LEGO Mindstorms NXT whose components can be assembled with other actuators and sensors to build a robotic thing without soldering. The robot can then be connected to the PC via Bluetooth and controlled by our toolkit.

The programmer next starts coding (Code 1). They write a few sentences in the main function to task the Ikimo robot with simply going forward and then run the program to test the connection between the robot and PC. After they confirm the connection, they write more code to test if the

```
Robot r = new Ikimo("btspp://deadbeaf");

// Test connection.
r.connect();
Task goForward =
  new GoForward();
goForward.assign(r);
goForward.start();

// Test a motor for ringing bells.
r.addExtension("IkimoMotor", Port.DC3);
r.connect();
IkimoMotor motor =
  r.requestResource("IkimoMotor", this);
motor.drive();

// Make the robot run away from the user while the bell rings.
VectorField vf = new VectorField() {
  // Define vector field responsive to the user's position.
  public void getVectorOut(Position robotPos, Vector2D v) {
    Position personPos = markerDetector.getPosition(person);
    vector.set(
      robotPos.getX()-personPos.getX(),
      robotPos.getY()-personPos.getY());
  }
};
Task runAway = new VectorFieldTask(vf); Task ring = new Ring();
runAway.assign(r); ring.assign(r);
runAway.start(); ring.start();
```

Code 1. Code snippets written during the development

hardware configuration is correct and the alarm rings properly. They also design a vector field that causes the alarm clock to run away from the user. Since the alarm clock was broken down, they need to prepare a graphical user interface to allow the user to set the alarm time. Though we omitted the details, the programmer iteratively writes this code and tests whether each code snippet works properly.

Finally, the programmer integrates all of their code and starts testing the whole application. While they test if the alarm clock works as intended, they use the runtime debug tool to monitor its status and configure API parameters until its behavior stabilizes.

PHYBOTS: HARDWARE AND SOFTWARE

We now explain PhyBots' hardware requirements and the implementation of our software APIs. The building stacks of the software architecture are shown in Figure 2.

Supported Hardware

The toolkit runs on a PC with a Java development environment. Depending on the operating system, cameras are controlled by DirectShow or QuickTime. Phybots has built-in support for LEGO Mindstorms NXT, Ikimo, iRobot Roomba and Create, but the programmer can add support for any robot that can be controlled via Bluetooth, TCP/IP, or the serial port.

APIs for 2D Locomotion

Two-dimensional (2D) locomotion in the environment is one of the fundamental functions of a robot and is achieved through localization and navigation. Our strategy in

designing the APIs is to be as similar to graphical UI applications as possible because they are well understood by many programmers. New concepts are introduced only when necessary. This strategy was evaluated in the first user study as described later.

Localization in Global Coordinates

Although the 2D GUI coordinates are defined with respect to the pixels of display devices, coordinates in the real world are not defined a priori. Phybots defines the area of interest by the field of view of the camera looking down on a flat surface, while visual markers define the entities of interest and whose positions are detected from images captured by the camera. We use ARToolKit [10] markers for the visual markers, whose detection algorithm is available as an open source library.

With our toolkit, the location information of an entity, or robotic thing, is provided by a *MarkerDetector* instance implementing *LocationProvider* interface. The programmer can simply call *getLocation* to get the location immediately or *addEventListener* to get notified when the location is updated. When built-in localization is not satisfactory, another *LocationProvider* can be implemented that possibly uses motion capture or other localization methods, retaining the usability of the APIs. Location information is a 2D coordinate combined with a direction in centimeters and radian. The functionality to convert coordinates between the real world in centimeters and the camera image in pixels is provided by *Camera* instance that captures images, which is expected to work with *LocationProvider* interface.

Navigation by Global Coordinates

The toolkit supports robotic things with differential wheels that can go forward, go backward, as well as rotate left and right in place. Navigating robots to a specified destination on the floor in the real world corresponds to moving the position of a component in a graphical UI. However, unlike UI components, robots cannot instantly "teleport" to this destination; they require running a localization feedback loop that commands the robot until it arrives at the destination.

To provide high-level navigation, the toolkit provides three built-in classes: *Move* makes a robot go to a specified position, *Push* makes the robot bring a specified object to a specified position by pushing it, and *TracePath* makes it visit specified waypoints in order. The instance of these classes has *assign-robot* method to assign the task to the robot instance and *start*, *pause*, *resume*, and *stop* methods to control its status. Event listeners are used for notifications when the task status has changed.

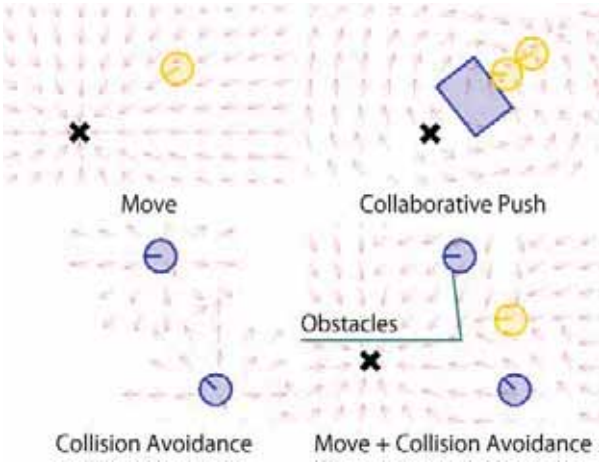

Figure 3. Vector fields for navigation of robotic things.

Vector Field Navigation

The above navigation APIs do not allow specifying dynamic behavior of the robotic thing in responsive to changes in the environment. For example, it is not easy to make a robotic thing follow the user or another thing.

To provide a more flexible way to design interactive behavior, the toolkit abstracts its strategy by means of vector fields, which have long since been used in the field of robotics [3]. Example vector fields are shown in Figure 3. The programmer can define a strategy by providing a vector field as an instance of the *VectorField* interface, on which a robot moves in the direction of a vector at its position. Actually, the *Move* and *Push* APIs are defined using vector fields. A vector field for *Move* is like a whirlpool and for *Push* is like an electric field from a dipole. When multiple *Push* tasks for transporting the same object to the same location are assigned to multiple robots, the robots achieve cooperative transportation of the object [8]. In addition, existing vector fields can be combined into a new one. For example, the combination of the collision avoidance field and *Move* task achieves navigation to the destination without collisions.

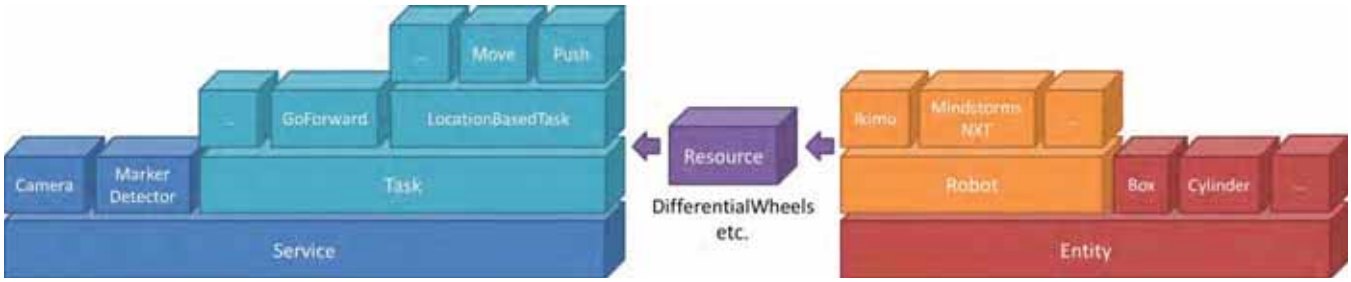

Figure 2. Phybots software building stacks

Centralized and Extensible Architecture

As shown in Figure 2, APIs for 2D locomotion such as *Move* and *Push* are provided on top of Phybots software stacks. A *Phybots* singleton provides direct access to all of the instantiated software components related to Phybots such as cameras, robots, tasks, etc. The programmer can define his own class on top of these stacks to extend the toolkit to their own needs.

Robot, Resource, and Task Abstraction

Our toolkit abstracts both robots and tasks through the *Robot* and *Task* interfaces and their abstract implementations. A *Robot* instance has a set of function units named *resources*. A *Task* instance requests one or more *resources* of a *Robot* instance to carry out the task such as pushing an object, cleaning the floor with brushes, putting on and off a pen on the paper, etc.

The *Resource* interface serves three purposes. First, it enables the dynamic building of a robot by composing several hardware components. For example, a LEGO Mindstorms robot has multiple ports to connect building blocks such as actuators and sensors. If the capability of the robot was statically described in a class definition, the programmer would have to rewrite it whenever they changed the hardware configuration. The *resource* abstraction allows the programmer instead to construct a robot instance dynamically with a different set of *resources*.

The second purpose is to improve the portability of the task definition so that it can be used with various kinds of robots. For example, navigation APIs introduced in the previous section request *WheelsController* for mobility functions. As long as a robot has a *WheelsController* resource, the robot can be used in any application that uses the navigation APIs.

Third purpose is to enable the safe assignment of multiple tasks to one robot at the same time. Some types of tasks need to obtain resources of a robot exclusively, which is managed by the toolkit. For example, an output to an actuator should be controlled from at most one task at a time, while observation of the output does not need such exclusive control. In this case, interfaces for observation and control are defined separately as *Wheels* and *WheelsController*. An interface that requires exclusive control must extend the *ExclusiveResource* interface, whose instance is managed properly by the toolkit to avoid any conflict.

With these abstractions, backend code that runs the feedback loop to accomplish a certain type of task is written as a class definition and is instantiated and controlled from user interface code without needing to learn its implementation.

Workflow for Higher-level Task Management

The toolkit provides a data structure named *workflow* for higher-level task management based on the task abstraction, which represents the procedure of tasks by one or more robots. A workflow is a kind of directed graph whose specification is drawn from the activity diagram of the Unified Modeling Language (UML) 1.0 [21]. The diagram is used to show the flow of discrete stepwise activities, which is similar to the traditional flowchart but different in that the diagram allows concurrent execution of multiple nodes. The *workflow* data structure consists of nodes and directed edges. An *action* node represents a task and a robot assigned to it. A control node such as a *fork* or *join* node coordinates program flow. Each directed edge represents a *transition* from one node to another that is usually activated when the process in the source node is finished, such as on task completion. Since robot applications must often handle timeouts, we also provide a convenient "*timeout transition*" that activates when a specified time has passed since the process in the source node was started.

Each *workflow* instance is constructed through a user interface that relieves the programmer from writing many event listeners to manage the status of multiple tasks. Since the *Workflow* class and *Task* interface share their major purpose of encapsulating time-consuming processes, both of them have methods to control their status and add event listeners to receive status change notifications.

Runtime Debug Tool

During user tests of robot applications, the programmer usually stands by the running system to check whether the robot runs properly, and if not, to diagnose the cause of the error and sometimes restart the system. However, such debugging is not easy since there are many possibilities for the cause including hardware (unstable wireless connection, low battery, broken actuators, etc.) and software issues (incorrect parameter configurations such as threshold for binarizing images in the marker detection process, maximum distance to the destination to judge *Move* task as finished, etc.). In addition, restarting the whole system and recovering the previous state takes time.

To address these issues, the toolkit provides an interactive tool named the "runtime debug tool" (Figure 4). The tool allows the programmer to monitor and configure the status of the application, and re-run the task or workflow in order to help figure out the error cause and remove the need to restart whole system. It consists of three main components described below, each of which is tightly coupled with the APIs of our toolkit. All of the presented information is accessible through methods of a *Phybots* singleton. These components can be used not only for debugging but also as part of an application so that it benefits from their rich GUIs.

Entity Monitor

The entity monitor allows the programmer to monitor the status of robots and objects, as well as change their configuration and even instantiate a new robot instance. For example, the programmer can monitor the status of the output to the actuators of a robot. He can also change the

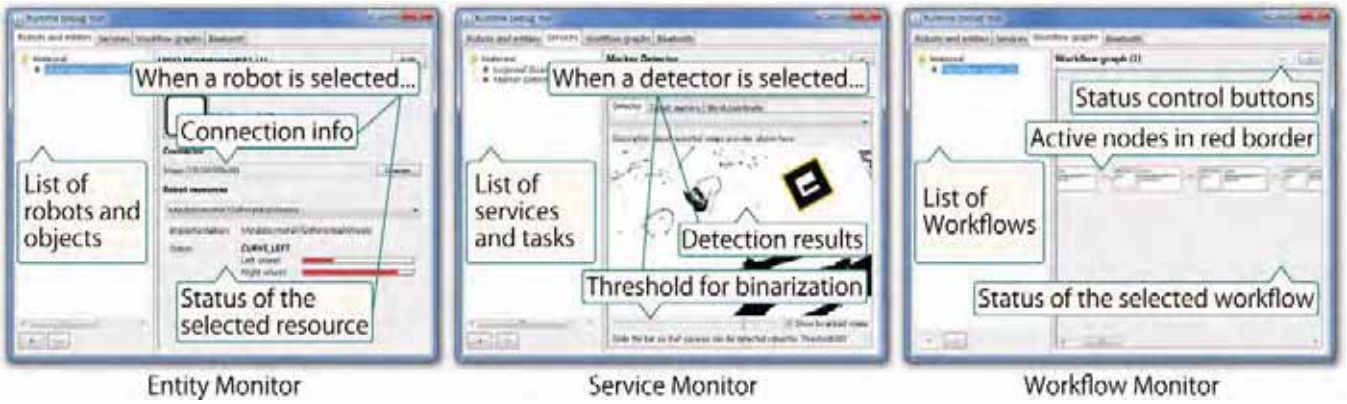

Figure 4. Runtime debug tool.

Bluetooth address and reset the connection to the robot. All of the robot and object instances are registered to our toolkit during their instantiation process and can be retrieved by the *getEntities* method of the *Phybots* singleton. The results can be filtered by specifying a class object such as *getEntities(Roomba.class)*. Each GUI component is retrieved by *getConfigurationComponent* of a robot and object instance, and the use of the component is not limited in the runtime debug tool but can be used in the application to allow parameter configuration by the end user.

Service Monitor

The service monitor allows the programmer to monitor, configure parameters, and control the status of services, which are active functions of our toolkit that include tasks, such as capturing images from a camera, detecting markers, and navigating a robot to the destination. All of the service and task instances can be retrieved by the *getServices* method of the *Phybots* singleton. As is the case for a robot and object instance, every GUI component can be retrieved for further use from a service and task instance.

Workflow Monitor

The workflow monitor allows the programmer to monitor and control status of workflows. The programmer can save the workflow and re-run it afterwards in the application to test the same scenario many times. He can also solve the hardware problems of a robot and re-run its workflow by: stopping the workflow; restarting the robot; re-establishing connection to the robot; and restarting the workflow.

USER STUDIES

We have iteratively developed, evaluated, and refined our toolkit. Evaluations were carried out at two points during a three years period. First, we provided the alpha version of the toolkit with limited functionality to fourteen graduate students who took HCI lecture courses in order to obtain feedback about the navigation APIs. Second, after we revised our toolkit to provide the complete functionality described in this paper, we provided the toolkit to seven graduate students majoring in HCI. We wanted to see the breadth of the robot applications that can be prototyped with our toolkit, as well as to know its limitation. At the same time, we provided our toolkit to two graduate students who are majoring in robotics and have development experience with well-known robotics middle-ware; and had discussions with them to compare this toolkit with other robotics toolkits.

Providing an Alpha Version to HCI Students

Eleven groups formed of fifteen graduate students enrolled in an HCI course attended this user study, which consisted of a five times weekly one-and-half hour course lecture. They are allowed to play with robots outside class hours. We asked each student group to create an original robot application with an alpha version of the toolkit that only provides two-dimensional localization and navigation APIs. We told the students that their deliverables would not affect their course scores. Fourteen students had never written a program for a robot before, while one student had experience with writing a program for a LEGO Mindstorms robot. We gave them our homebuilt mobile robots, visual markers, web cameras, and the toolkit with a robot class for the robots. The robot is wirelessly connected to a host computer via Bluetooth. It measures 7.5 x 8.8 x 6.5 [cm] and is driven by two stepping motors. Visual markers were 5.5 [cm] square. Cameras could capture images with 800 x 600 [pixels] at 30 [fps]. Along with the hardware, we provided a one-and-half hour lecture to explain the overview of the toolkit and provided sample codes and API documents. The participants could ask any questions during their development process.

After four one-and-half hour lectures, the student groups spent four to seventeen days in total for the development

Figure 5. Applications developed by the students.

and all of them had successfully developed robot applications; three of eleven results are shown in Figure 5. The applications vary among new user interfaces for a robot, multi-player games, and practical tools. Eight of eleven groups reported that the programming with the toolkit were easy. The rest three groups reported that the difficulty came from parameter configuration, which shows that the proposed set of localization and navigation APIs were used effectively for rapid prototyping. The hardware setup could be easily deployed on the desktop with cameras attached at heights of about 50 [cm].

Lessons learned

First, the applications seemed to be limited by the hardware and software specifications. Since we provided the robots and the toolkit as one package for simplicity, they were not designed to be extended by the students, and their technical specifications were not open. As a result, the robots were never extended mechanically, though one group attached paper-made arms to a robot to gather objects. It was not possible to define new locomotion strategies to be used other than those predefined as APIs. These observations revealed to us the importance of the extra features of the robot besides the basic locomotion features. Therefore, we were motivated to do a code refactoring of the toolkit so that it can expose public interfaces and abstract classes to provide enough extensibility for the programmer.

Second, the students appeared to avoid dealing with complex workflows for robots. They simply called navigation APIs in event listeners of GUI components. This observation led us to provide *workflow*, a data structure to relieve the programmer from high-level task management.

Third, the students complained that the API parameters could not be easily configured. They tweaked the parameters in source code and evaluated their performance by restarting their applications many times. We noticed the lack of support for the prototype test phase and planned to implement the *runtime debug tool*.

Providing the Current Version to HCI Students

After we implemented the functionality described in this paper, we ran a study with students again, though all of the members were different from the first user study. With this user study, we aimed to observe the whole process of prototyping including hardware preparation, programming, debug, and user test.

Three undergraduate and four graduate students from six laboratories, all of whom majored in HCI, attended the user study. Five students had experience of programming simple robots. Two of the five had experience of building an original robots and programming their firmware, while three used prebuilt robots such as Roomba. Six students have experience of using toolkits for physical computing such as Arduino, Gainer, and SunSPOT. We asked each of the seven students to create an original robot application with the toolkit. We gave them Ikimo, an Arduino-based open-source mobile robot, visual markers of 5.5 [cm] or 11 [cm] square, a web camera same as the first user study, and the Phybot toolkit. An Ikimo robot is wirelessly connected to a host computer via Bluetooth. It measures 10.5 x 13 x 7 [cm] and is driven by two direct current (DC) motors. Along with the hardware, we overviewed the toolkit and provided sample codes with API documentation.

All students spent two to four days to successfully develop

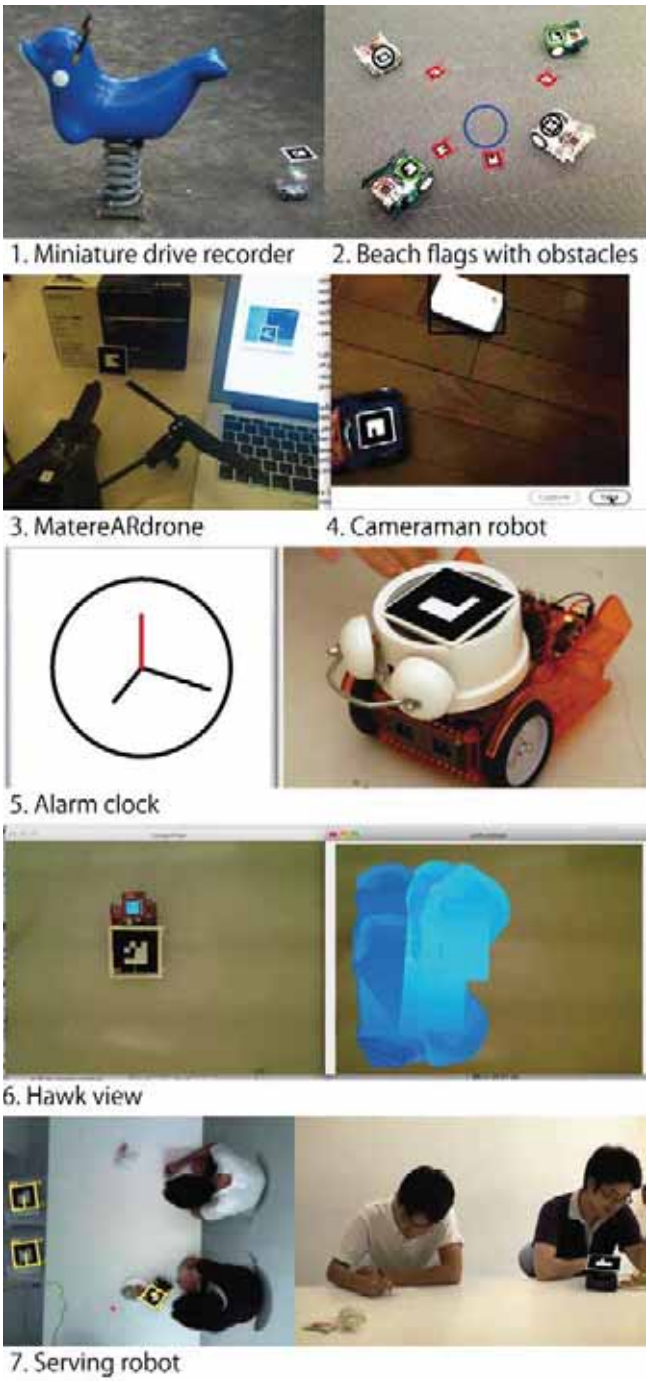

Figure 6. Applications developed by the students.

robot applications as shown in Figure 6. The time spent for the development decreased significantly from the alpha version, which shows the improvement of the overall usability of the toolkit. They especially appreciated the extendibility which enables adding new locomotion strategies and supporting new robots and the runtime debug tool, both of which are the main difference from the alpha version. We will briefly introduce the resulting applications below.

1. Miniature drive recorder: This application makes a robot go around the specified object and take video with its mounted mobile phone. Position and rotation of the camera is also recorded along with the video, which can be used for 3D reconstruction. The *TracePath* task is used to control the robot and GUI on captured images and is used to set the position of the object and the radius of the circle to trace.

2. Beach flags with obstacles: This application is a two-player game to make the player's robots go into the center circle as fast as possible while avoiding obstacles. All robots are controlled by the *Move* task with an additional vector field for collision avoidance. Their positions are watched by an event listener to detect the goal. In its current implementation, game status such as the goal area and the name of the winner is shown in GUI with some special effect.

3. MatereARdrone: This application provides a remote control interface of a quadcopter capable of flying and capturing images with its mounted camera. The programmer wrote a robot class and resource classes for the quadcopter to use its full feature. His future work is to use this quadcopter as a source for *MarkerDetector* and make it possible to navigate mobile robots on the floor without limitation of the static camera position.

4. Cameraman robot: This application takes photos of guests. First, it detects guest positions from the image captured from the ceiling-mounted camera and calculates an ideal position and rotation from which all of the guests could be seen in a photo. Second, it navigates the robot to this ideal position. Third, it takes photo with all of the guests being in the frame. These actions are constructed as a *workflow* and executed in order.

5. Alarm clock: This application rings an alarm bell and gets away from the user when the specified time has come. A *vector field* is used to design this behavior and a *task* is defined to ring and stop the bell, which controls an extra actuator connected to the robot. A GUI to set the time to wake the user up is provided. The user's position is tracked by a ceiling mounted camera.

6. Hawk view: This application continuously provides a heat map of the field that represents any kind of data captured by a robot-mounted sensor such as temperature, smell, sound, etc. The heat map is provided as follows. First, the data is continuously retrieved by a mobile robot, which navigates around the field. Then, the color representing the data is displayed on a small LCD display on the robot. The data on the LCD display is captured with its location by the ceiling mounted camera and the result is plotted on the screen.

7. Serving robot: This application monitors people coming in and out of a public space such as a large table in a library. The robot responds to the action of the users: When an adult user sits on a chair, the robot delivers a cup of tea. When a child user sits on a chair, it delivers a candy. When some time has passed since the user sits on a chair, it delivers a piece of cookie. When the user leaves his place, the robot removes the garbage. These actions are detected by the change of seeing and not seeing the marker on the chairs. The robot navigates to the front of the user, waits for a moment, and gets back by *workflow*. Things to deliver are put onto the dish attached to the back of the robot by a person who gets the instruction by the system behind the scenes.

Lessons learned

While typical robot applications often use robots as their primary user interface, it seems that using GUI (and possibly other HCI techniques) as the user interface for robots is also the promising way. Many of the students used GUI as the primary channel to get user instruction to robots. Three students (1, 2 and 6) used a top-down view as the user interface for the robots, while the other three (3, 4, 5) used their own GUI. The last one (7) did not provide any apparent user interface for the end-users but monitored the status of them to provide the service. In addition, it required a person to work behind the scene to help the robot, and the person got instruction from the system through GUI.

Providing extendibility to the toolkit through abstractions such as *vector field*, *robot*, and *task* interfaces was welcomed by the students. One student (5) defined a new vector field (5) while another student (2) combined existing vector fields to make a new one. Two students (3 and 5) extended the toolkit to use his robot and to define a new task to ring a bell

The programmers who have experience usingJava and GUI programming (2, 4, 5, and 7) were familiar with inner classes which are often used for event listeners. They tended to write inner classes for definition of a vector field, a task, and a listener to a task and workflow instance. While these inner classes have access to the local variables and thus allow convenient programming, this tendency led him/her to write the whole code in one file. Since along code in one file is difficult to read, the programmer should divide the inner class to another file, as some students (3 and 6) did . *Workflow* prevented the programmers (4 and 7) from writing many listeners to the tasks and thus prevented the code to be too long.

Providing the Current Version to Robotics Students

Two graduate students from the robotics laboratory attended the user study and partnered together. For their research, they had used Robot Operating System (ROS) [16], an open-source well-known middleware, First, we gave them the same set of hardware as in the previous user study – mobile robots, visual markers, web cameras and the toolkit. Second, we provided the same one day lecture, sample code, and documentations. When they adapted to the toolkit with the sample code, we asked them to implement the same application with ROS. The application is called "click and run," where the user clicks the position on the camera image shown in a GUI window to make the robot move to thatposition. After the implementation, we interviewed the students to compare the toolkits.

The students successfully implemented almost the same application with ROS. The module for the user interface was written in C++ and was capable of showing received images in a GUI window and sending clicked position information to another module. To glue it with robot control modules, they wrote code in Common Lisp, which runs on another module named "euslisp." There was a file to describe dependency on other modules and a file to specify options for running the system, such as the name of the communication port etc. The total code length was 668 lines (586 lines for the GUI module and 82 lines for Lisp), which was longer than our 130 lines sample code for Phybots.

Lessons learned

Regarding the differences between the programming languages, we cannot simply compare the lines of code. 586 lines of code is a relatively large number for the simple user interface, which seems to root in the completely modular architecture of the middleware. Since the user interface module was connected to the other modules only with its publicly defined channels, it must have assumed every possible way to be used from the others and defined those many channels. This is like writing a public class in Java with public getter and setter methods. On the other hand, when we write Java code in Phybots, a GUI class is usually the main code itself, or is extended as an inner class of the main code. It ensures that the GUI class is used from within the main class only, removing the need to write many getter and setter methods.

The user interface code was treated equally to the other control modules in ROS, and it did not have direct access to any specific model of the world including robots, objects, and tasks. This is problematic because the user interface often needs to know what kind of information it is manipulating. Our toolkit provides direct access to the world model from the user interface code, which makes the prototyping easier.

Availability of many modules is thought to be a prerequisite for a good middleware, but was also found to be a shortcoming for novice users. The students complained that it was difficult to learn about modules and that documentation was lacking. The only way would be to throw their questions to the online forum and wait for an answer. On the other hand, our toolkit provides a good starting point for general software programmers to prototype robotic applications, thanks to its compact APIs, event-driven programming, and rich GUI components.

DISCUSSION

Physical UIs, Robots, and Phybots

Phybots aims to provide a prototyping environment for robotic things. While its application domain is between physical UIs and robots, its target users are the same as physical UIs. Therefore, we adopted the approach that weighs more on the physical toolkit side.

First, in the user studies the robotic things were built from building blocks without soldering. There was also no need for microcontroller programming and the user could control the hardware directly in Java code. It was as easy as the physical UIs.

Second, Phybots required the camera as a global sensor set in the environment instead of sensors attached to the robotic things. Making an assumption on the environment is often invalid in the robotics applications since many applications are expected to work in an unknown environment. Though, as shown in the user studies, everyday spaces could be easily equipped with the hardware and turn into a known environment for the applications. Therefore, our approach was valid for the prototyping. Our future work may use a depth camera such as Kinect instead of the normal camera and implement techniques for object recognition to remove the need of the visual markers on top of the object. It would also allow natural interaction between the robotic things and the user.

Third, built-in APIs for locomotion of robotic things only provided basic functionality. In the second user study for HCI students, some of them suggested us to implement some algorithms for path planning to support *TracePath* task. We recognize the needs and will provide handful APIs. In the sense of functionality, we should port more useful algorithms from robotics but keep the usability of the exposed APIs. Our future work may implement a software module for robotics middleware such as ROS that works as a bridge between Phybots and the middleware. Then, its plenty of modules will be accessible to our users.

3D World and 2D Workspace

Phybots supports only two-dimensional activities of mobile robots. While it is possible to extend the framework to three dimensions, we decided to focus on two dimensions for several reasons. First, it is much easier to understand two-dimensional coordinates and develop applications in them because software programmers are familiar with GUIs that use two-dimensional coordinates and whose standard input

and output devices such as a mouse and a display are capable of processing two-dimensional information. Second, two-dimensional information is often sufficient to represent the global positions of the most relevant entities including the robots, objects, and users because all of them usually reside on a two-dimensional surface such as the desktop or floor. Our assumption is that global locomotion tasks can be handled in two-dimensional coordinates, while local manipulation tasks require three-dimensional coordinates to be performed robustly, such as picking up an object on the floor and placing it on a shelf. Our future work will explore how to support such manipulation tasks for more advanced robots.

Human as Part of Robotic System

Serving robot in the second user study uses human behind the scenes instead of a complex robot. This approach is thought to be valid when we can assure that it is technically possible to implement the robot. Such a case would rather be encouraged in the prototyping, since the application focuses in its interaction design with the end-user who does not care about the backstage. Regarding the high-level abstractions of Phybots, it is possible to define a *Robot* class that represents human, whose resources would show text message on a display or produce a speech sound to tell what to do. To confirm this possibility, we implemented an application to make human write figures on a piece of paper, which was originally developed for a calligraphy robot.

CONCLUSION

In this paper, we proposed Phybots, a toolkit for making robotic things. Its assuming hardware setup was easily deployed to everyday places. The users could implement various physical applications that showed possibilities of adding mobility to everyday things. While the functionality of the toolkit is limited compared to professional robotics toolkits, its well-designed APIs could be easily understood by the users and was enough for the rapid prototyping. Phybots is open-source and available at http://phybots.com/.

ACKNOWLEDGEMENTS

We would like to thank all participants of the user studies who enjoyed making robotic things with the toolkit.

REFERENCES

1.Arduino. http://www.arduino.cc/.

2.Blank, D., Kumar, D., L. Meeden., and Yanco, H. Pyro: A python-based versatile programming environment for teaching robotics. *JERIC 2005*, 3(4). ACM (2005).

3.Borenstein, J., and Koren Y. Real-Time Obstacle Avoidance for Fast Mobile Robots. *IEEE Trans. on Systems, Man and Cybernetics*, 19(5). (1989), 1179-1187.

4.Erwin, B., Cyr. M., and Rogers, C. LEGO Engineer and RoboLab: Teaching Engineering with LabVIEW from Kindergarten to Graduate School. *International Journal of Engineering Education 16*, 3 (2000), 181-192.

5.Gerkey, B., Vaughan, R. T., and Howard, A. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proc. ICAR 2003*, (2003), 317-323.

6.Greenberg, S., and Fitchett, C. Phidgets: easy development of physical interfaces through physical widgets. In *Proc. UIST 2001*. ACM (2001), 209-218.

7.Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proc. UIST 2006*. ACM (2006), 299-308.

8.Igarashi, T., Youichi, K., Masahiko, I. A Dipole Field for Object Delivery by Pushing on a Flat Surface. In *Proc. ICRA 2010*, IEEE (2010), 5114-5119.

9. InMojo. Ikimo. http://www.inmojo.com/ikimo/.

10.Kato, H., and Billinghurst, M. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *Proc. IWAR 1999*. (1999), 85-94.

11.Klemmer, S. R., Li, J., Lin, J., and Landay, J. A. Papier-Mache: toolkit support for tangible input. In *Proc. CHI 2004*. ACM (2004), 399-406.

12.Microsoft. Microsoft Robotics Developper Studio. http://msdn.microsoft.com/en-us/robotics/.

13.Montemerlo, M., Roy, N., and Thrun, S. Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit. In *Proc. IROS 2003*, IEEE/RSJ (2003), 2436-2441.

14.OpenCV. http://opencv.willowgarage.com/.

15.Pedersen, E., and Homæk, K. Tangible bots: interaction with active tangibles in tabletop interfaces. In *Proc. CHI 2011*. ACM (2011), 2975-2984.

16.Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. ROS: an open-source Robot Operating System. In *Open-source software workshop of ICRA 2009*, (2009).

17.Raffle, H., Parkes, A., and Ishii, H. 2004, Topobo: a constructive assembly system with kinetic memory. In *Proc. CHI 2004*, ACM (2004), 647-654.

18.Sugiura, Y., Igarashi, T., Takahashi, H., Gowon, T. A., Fernando, C. L., Sugimoto, M., and Inami, M. Graphical instruction for a garment folding robot. In *ACM SIGGRAPH 2009 Emerging Technologies*, ACM (2009).

19.Sugiura, Y., Sakamoto, D., Withana, A., Inami, M., and Igarashi, T. Cooking with robots: designing a household system working in open environments. In *Proc. CHI 2010*. ACM (2010), 2427-2430.

20.The Machine Perception and Intelligent Robotics Lab, University of Malaga. The Mobile Robot Programming Toolkit (MRPT). http://www.mrpt.org/.

21.UML Partners. 1997. *Unified Modeling Language v. 1.0*. OMG document ad/97-01-14.