VIPLE: Visual IoT/Robotics Programming Language Environment for Computer Science Education

Yinong Chen and Gennaro De Luca
School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287-8809, U.S.A.

Abstract— **Microsoft released its Robotics Developer Studio (MSRDS) and Visual Programming Language (VPL) in 2006. Microsoft VPL is service-oriented, uses workflow-based visual programming, and has strong support for parallel computing. It is a milestone and flagship in software engineering and in computer science education. Many universities and high schools have adopted VPL as a tool for teaching computing and engineering concepts and for programming robots. Unfortunately, as part of Microsoft's restructuring plan, the robotics division of Microsoft Research was suspended on September 22, 2014, leaving the Microsoft VPL community without updates and support. Arizona State University (ASU) is among the schools that adopted VPL since its first release in 2006. We started to find a solution to our VPL-based curriculum in 2014. This paper presents our research and development of a new visual programming language and its development environment: ASU VIPLE (Visual IoT/Robotics Programming Language Environment). ASU VIPLE extends the discontinued Microsoft VPL to sustain our curriculum and to help the community with their VPL projects. ASU VIPLE supports LEGO EV3 and all IoT devices based on an open architecture. ASU VIPLE integrates engineering design process, workflow, fundamental programming concepts, control flow, parallel computing, event-driven programming seamlessly into the curriculum. It has been pilot tested at Arizona State University in summer 2015 and in spring 2016, as well as in several other universities. ASU VIPLE software and documents can be freely downloaded at: http://venus.eas.asu.edu/WSRepository/VIPLE/**

Keywords - MSRDS VPL, visual programming, computer science education, Internet of Things, robot, parallel computing

I. INTRODUCTION

The Internet of Things (IoT) refers to uniquely identifiable objects (things) and their virtual representations in an Internet structure [1]. The concept was initially applied in the Radio-Frequency Identification (RFID) tags to mark the Electronic Product Code (Auto-ID Lab). The concept of the IoT is extended to refer to the world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes [2]. The Internet of Intelligent Things (IoIT) deals with intelligent devices that have adequate computing capacity. Distributed intelligence is a part of the IoIT [3]. According to Intel's report, there are 15 billion devices that are connected to the Internet, in which 4 billion devices include 32-bit processing power, and 1 billion devices are intelligent systems [4].

The development of cloud computing pushed the desktop-based computing platform into an internet-based computing infrastructure. It also changed the concept of physical products or things into services. The endorsement and commitment to building and utilizing cloud computing environments as the integrated computing and communication infrastructure by many governments and major computing corporations around the world have led to the rapid development of many commercial and mission-critical applications in the new infrastructure, including the integration of physical devices, which gives the Internet and IoT their seemingly unlimited computing capacity.

In addition to IoT, a number of related concepts and systems have been proposed to take advantage of Internet and cloud computing. A cyber-physical system (CPS) is a combination of a large computational and communication core and physical elements that can interact with the physical world [5][6][7]. CPSs can be considered the extended and decentralized version of embedded systems. In CPSs, the computational and communication core and the physical elements are tightly coupled and coordinated to fulfill a coherent mission. The U.S. NSF (National Science Foundation) issued a program solicitation in 2008 on CPSs, envisioning that the cyber-physical systems of tomorrow would far exceed those of today in terms of adaptability, autonomy, efficiency, functionality, reliability, safety, and usability. Research advances in cyber-physical systems promise to transform our world with systems that respond more quickly (e.g., autonomous collision avoidance), are more precise (e.g., robotic surgery and nano-tolerance manufacturing), work in dangerous or inaccessible environments (e.g., autonomous systems for search and rescue, firefighting, and exploration), provide large-scale, distributed coordination (e.g., automated traffic control), are highly efficient (e.g., zero-net energy buildings), augment human capabilities, and enhance societal wellbeing (e.g., assistive technologies and ubiquitous healthcare monitoring and delivery) [8].

An autonomous decentralized system (ADS) is a distributed system composed of modules or components that are designed to operate independently but are capable of interacting with each other to meet the overall goal of the system. ADS components are designed to operate in a loosely coupled manner and data is shared through a content-

oriented protocol. This design paradigm enables the system to continue to function in the event of component failures. It also enables maintenance and repair to be carried out while the system remains operational. Autonomous decentralized systems have a number of applications including industrial production lines, railway signaling and robotics [12][13].

Kakuda presented the concepts, technologies, and case studies of the next generation of assurance networks [11]. The study made use of the available redundant computing and communication resources for dependability purposes. The paper laid out the roadmap to the design and implementation of the new generation of assurance networks. These networks inherit different features from a number of systems, including cyber-physical systems, scalability and service orientation from cloud computing, the adaptability and autonomy from autonomous decentralized systems [12], fault tolerance and real-time computing from the responsive systems [14], and distributed real-time and embedded (DRE) systems. DRE systems are based on a model driven architecture and model integrated computing [15], are applied in the situation where application requirements and environmental conditions may not be known *a priori* or may vary at run-time, and mandate an adaptive approach to management of quality-of-service (QoS) to meet key constraints such as end-to-end timeliness. Different DRE middleware systems have been developed by the members of the Distributed Object Computing (DOC) Group, which is a consortium consisting of universities and industry partners [16]. The new generation assurance networks also include many other common features such as trustworthiness and mobility [17][18]. Scheduling tasks and allocating resources in such environments require new strategies and techniques. The gang scheduling and real-time scheduling techniques in ad hoc distributed systems and on the elastic cloud environment ensure that related tasks are scheduled in groups and are running simultaneously [19][20][21], which can be used to coordinate the intelligent devices.

Robot as a Service (RaaS) is a cloud computing unit that facilitates the seamless integration of robots and embedded devices into Web and cloud computing environments [3][9]. In terms of the service-oriented architecture (SOA), a RaaS unit includes services for performing functionality, a service directory for discovery, and service clients for the user's direct access [10]. The current RaaS implementation facilitates SOAP and RESTful communications between RaaS units and the other cloud computing units. Hardware support and standards are available to support RaaS implementation. For example, the Devices Profile for Web Services (DPWS) defines implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices between Web services and devices. The recent Intel IoT-enabled architectures, such as Galileo and Edison, make it easy to program these devices as Web services. From different perspectives, an RaaS unit can be considered a unit of the Internet of Things (IoT), Internet of Intelligent Things (IoIT) (which have adequate computing capacity to perform complex computations [3]), a Cyber-physical system (CPS) (which is a combination of a large computational and communication core and physical elements that can interact with the physical world [4]), and an autonomous decentralized system (ADS).

This paper uses the concepts of IoT and Robot as a Service as examples to study the programming issues of all the aforementioned systems that are connected to the Internet or can be accessed through the Internet.

On the software development environment side, there are a number of great visual programming environments for education. MIT App Inventor [22] uses drag-and-drop style puzzles to construct phone applications. University of Virginia and Carnegie Mellon's Alice is a 3D game and movie development environment [23]. It uses a drop-down list for users to select the available functions in a stepwise manner. App Inventor and Alice allow novice programmers to develop complex applications using visual composition at the workflow level. Microsoft Robotics Developer Studio (MSRDS) and Visual Programming Language (VPL) are specifically developed for robotics applications [24], which is a milestone in software engineering, robotics, and computer science education from many aspects. Microsoft VPL is service-oriented; it is visual and workflow-based; it is event-driven; it supports parallel computing; and it is a great educational tool that is simple to learn and yet powerful and expressive. Sponsored by two Innovation Excellence awards from Microsoft Research in 2003 and in 2005, Dr. Yinong Chen participated in the early discussion of a service-oriented robotics developer environment at Microsoft. MSRDS and VPL were immediately adopted at ASU in developing the freshman computer science and engineering course CSE101 in 2006. The course grew from 70 students in 2006 to over 350 students in 2011. The course was extended to all students in the Ira A. Fulton Schools of Engineering (FSE) at ASU and was renamed FSE100, which is offered to thousands of freshman engineering students now.

Unfortunately, Microsoft stopped its development and support for MSRDS and VPL in 2014 [25], which leads to our FSE100 course, and many other schools' courses using VPL, without further support. Particularly, the current version of VPL does not support LEGO's third generation of EV3 robot, while the second generation NXT is out of the market.

To keep our course running and also help the other schools, we take the challenge and responsibility to develop our own visual programming environment at Arizona State University (ASU). We name this environment VIPLE, standing for Visual IoT/Robotics Programming Language Environment.

ASU VIPLE is based on our previous eRobotics development environments [26]. It is designed to support as many features and functionalities of MSRDS and VPL as possible, in order to better serve the MSRDS and VPL communities in education and research. To serve this purpose, VIPLE also keeps a similar user interface, so that the MSRDS and VPL development communities can use VIPLE with little additional learning. VIPLE does not replace MSRDS or VPL. Instead, it extends MSRDS and

VPL in their capacities in multiple aspects. It can connect to different physical robots, including EV3 and any robots based on off-the-shelf processors.

The rest of the paper is organized as follows. Section II discusses IoT standards, protocols and the stack of the technologies related to IoT. Section III outlines the main functionalities of ASU VIPLE and how it is used to visually program IoT devices and services. VIPLE is compared and contrasted with Microsoft VPL in this section. Section IV presents a number of examples implemented in VIPLE to show its applications in computer science education. Section V discusses the interface and connection from VIPLE to IoT devices. Section VI concludes the paper.

II. IoT Standards and Protocols

The Internet of Things (IoT) is a general concept that can be interpreted in different contexts. Different protocols and standards can be used for the communication between the devices and the Internet. As shown in Figure 1, the IoT connects to the Internet through Internet protocols, such as HTTP, TCP, and IP. The data received from the IoT is represented as Web data in forms such as HTML, JSON, XML, and URI. The data can be further organized into ontology and presented in RDF, RDFS, and OWL for storing, analyzing, and reasoning. The data is typically processed in Service-oriented and Web-based computing environments. If the data amount is big, it can be processed by cloud computing and big data analysis. At this end, the IoT and its data are fully integrated into the Web and the virtual world, and all the technologies and applications developed can be applied to process IoT data and control the physical world connected to the IoT on the other side.

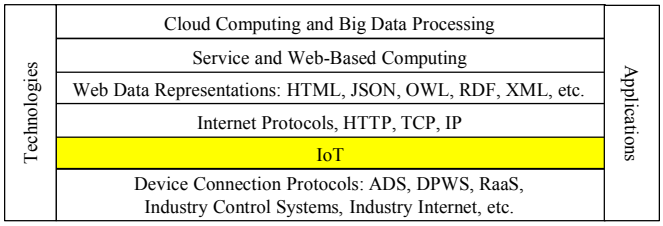


Figure 1. Internet of things and devices

On the other side, the IoT is connected to devices and the physical world through different device protocols and standards. For example, ADS uses a content-oriented protocol to broadcast data to a device bus, and the devices take the data based on a content code, instead of destination address [12][13]. Any other devices that need to delay information will pick up the data by content code and apply the delay information.

The Devices Profile for Web Services (DPWS) applies Web service standards in device interface and communication. It defines implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices between Web services and devices [27]. The DPWS specification is built on the following core Web Services standards: WSDL 1.1, XML Schema, SOAP 1.2, and WS-Addressing. Initially published in 2004, DPWS 1.1 was approved as an OASIS Standard together with WS-Discovery 1.1 and SOAP-over-UDP 1.1 2009. The Microsoft .Net Framework Class Library has defined classes for supporting DPWS device programming. Devices that implement DPWS are already on the market. For example, Netduino Plus is an interface board that wraps a device with a Web service interface, so that the device can communicate with a virtual thing (a Web service) without requiring device driver code. The device is available in major stores, such as Amazon.com.

RaaS applies Web service standards at the device level, just like DPWS. However, RaaS extends DPWS to include the service broker and service clients [3]. The service broker allows the devices (robots) to register their IP address once they are online. The registration is necessary, as many of the devices are connected to the Internet via Wi-Fi, and their IP addresses are dynamically assigned. Once registered, the devices can be discovered and connected by looking up the device broker. As robots can be powerful devices, RaaS allows the robots to install multiple clients, and the robots can execute a different client to perform a different set of functions.

In the next section, we will focus on the design and implementation of VIPLE and its interface to the Internet and to devices.

III. The Definition of VIPLE

ASU VIPLE is based on Microsoft VPL. Figure 2 compares the basic activities between ASU VIPLE and Microsoft VPL. VIPLE implements most basic activities in VPL and implements additional While, Break, and End While activities to facilitate loop building, which can reduce the circular paths in VPL diagrams.

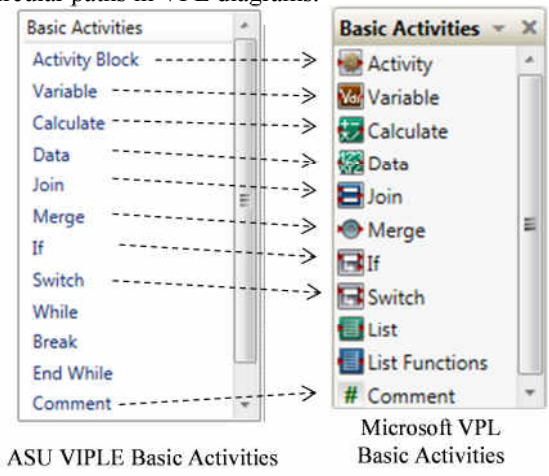
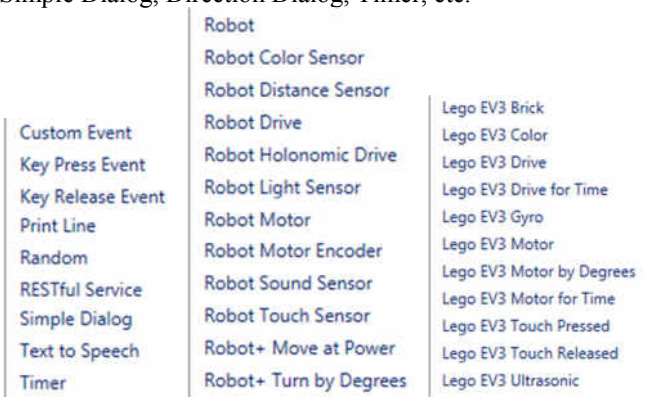


Figure 2. Activities: ASU VIPLE versus Microsoft VPL

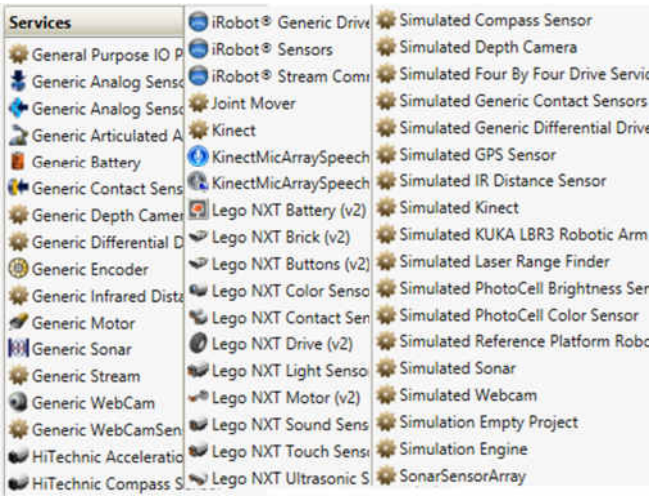
As can be seen, VIPLE has similar programming constructs and can be used for the same kinds of applications using these basic activities.

The usability of a language largely depends on the availability of library functions or called services. Figure 3 (a) shows the ASU VIPLE services, and (b) shows Microsoft

VPL services. Microsoft VPL implements multiple sets of vendor services, including general service, generic robot services, iRobot services, LEGO NXT services, and simulated services. The general services and generic robot services include Log, Text to Speech, SpeechRecognizer, Simple Dialog, Direction Dialog, Timer, etc.



(a) ASU VIPLE general services and robotic services



(b) Microsoft VPL Generic, Vendor, and Simulated services

Figure 3. ASU VIPLE services versus Microsoft VPL services

As shown in Figure 3, three sets of services are implemented in VIPLE. The first set is a list of general services, including input/output services (Simple Dialog, Print Line, Text To Speech, and Random), Event services (Key Press Event, Key Release Event, Custom Event, and Timer), and RESTful services.

The second set is the generic robotic services. VIPLE offers a set of standard communication interfaces, including Wi-Fi, TCP, Bluetooth, USB, localhost, and WebSocket interfaces. The data format between VIPLE and the IoT/Robotic devices is defined as a standard JSON (JavaScript Object Notation) object. Any robot that can be programmed to support one of the communication types and can process the JSON object can communicate with VIPLE and be programmed in VIPLE. As shown in the second part of Figure 3, all VIPLE services that start with "Robot" are generic robotic services. We can use these services to program our simulated robots and custom-built physical robots.

The third set is the vendor-specific services. Some robots, such as LEGO robots and iRobots, do not offer an open communication and programming interfaces. In this case, we can offer built-in services in VIPLE to access these robots without requiring any programming efforts on the device side. Currently, the services for accessing LEGO EV3 robots are implemented, so that VIPLE can read all EV3 sensors and control EV3 drive-motors and arm-motors, as shown in the third part in Figure 3. For those who do not want to build their own robots, they can simply use VIPLE and an EV3. The addition of EV3 services to VIPLE is significant, as it allows the Microsoft VPL developers who used NXT robots now to use the new EV3 robots.

The generic robot services allow the developers to use VIPLE to connect to an open architecture robot. In Microsoft VPL, DSS services developed specifically for MSRDS can be added into the VPL service list. In ASU VIPLE, RESTful services can be accessed in VIPLE diagrams. As RESTful services are widely used in today's Web application development, the access to RESTful services extends the capacity of VIPLE to a wide range of resources. ASU VIPLE does not have simulated services at this time, although the generic robot services can be used to interface with a custom simulation environment.

Many improvements are made in ASU VIPLE. For example, ASU VIPLE uses state.variable consistently. In VPL, local variables (the location of a variable's use has a direct link to the variable) use a variable name only, while global variables use the state.variable syntax. It is not easy for inexperienced programmer to recognize the scope of variables and often causes confusion.

In the next section, we will use educational examples to demonstrate the functions and applications of ASU VIPLE.

IV. EDUCATIONAL EXAMPLES IMPLEMENTED IN VIPLE

VIPLE is a general purpose programming language and is Turing complete in its capacity. It can be used for any kind of computational tasks. In VIPLE, a program is represented as a diagram of workflow, and the components are defined as activities in the diagram.

4.1 Teaching fundamental programming in VIPLE

ASU VIPLE is similar to Microsoft VPL not only in concepts but also in programming. We will now show examples of basic programming in ASU VIPLE. We will start with the Hello World program. Figure 4 shows the two versions of code using (a) Microsoft VPL and (b) ASU VIPLE. The two diagrams look the same. However, ASU VIPLE has simplified a number of steps to make it even easier for novice programmers. In this simple example, VIPLE automatically changes the type to string after a string is entered, and the default null value step in Microsoft VPL is eliminated.

As a more complex example, we show a counter using a modular design, where a module is implemented as an activity.

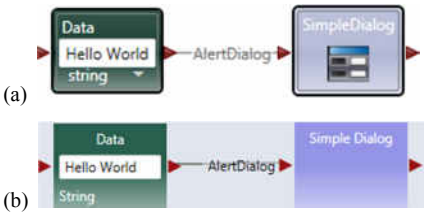
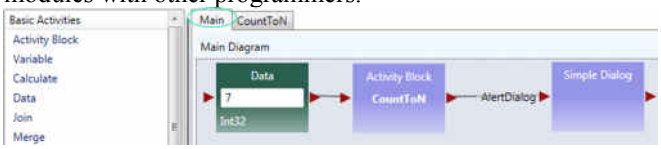
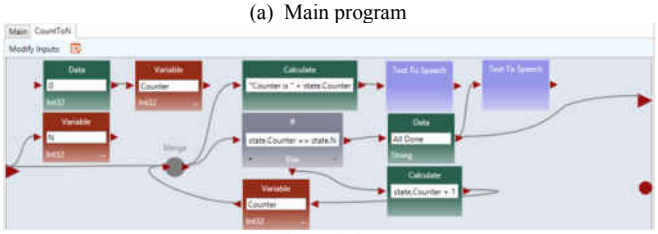


Figure 4. Hello world program in (a) VPL and (b) VIPLE

Figure 5 shows the VIPLE program that implements: (a) a main diagram and (b) a counter activity. The activity takes an input N in the main diagram. In the activity CountToN, it starts from 0 and adds 1 in each iteration. It stops when the counter value is equal to N. The Text to Speech service is used to read out the numbers in the activity, and the Simple Dialog Service is used to print a completion message (modification can allow the display of counted numbers instead). As can be seen in the activity diagram, both data output and notification (event) are supported in ASU VIPLE, as is the ability to define any number of named inputs to an activity. Similar to Microsoft VPL, a custom VIPLE activity can be converted to a VIPLE service, allowing sharing of modules with other programmers.



(a) Main program



(b) Activity

Figure 5. ASU VIPLE program implementing a counter

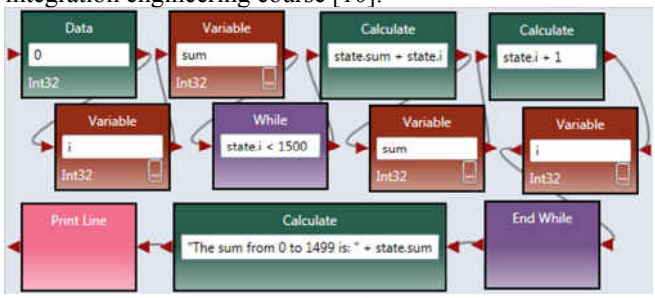
4.2 Teaching parallel computing concepts in VIPLE

VIPLE provides an environment with which instructors can teach students many of the fundamentals of computing and programming. It can also introduce parallel computing concepts, without being required to teach the specifics of synchronization and thread-safety or how to pass data to or from threads. The counter example in Figure 5 allows students to experience in part the difference between working in a multithreaded environment and working in a sequential environment. Namely, in that example, the "all done" text may be spoken before the final number is spoken. Students will learn that not all threads that start first, finish first.

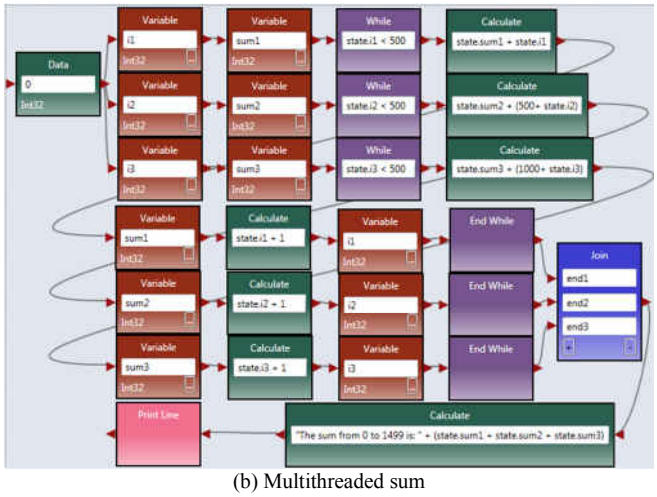
Figure 6 shows two ways to sum all the numbers from 0 to n (n = 1499 in this example). The diagram first initializes all variables to 0. Figure 6 (a) is the first way most students learn to write the algorithm. The program starts at 0, iterates through the sequence from 0 to 1499, and adds each value to the sum variable. At the end, sum will contain the final answer.

In contrast, Figure 6 (b) shows a parallel implementation of the algorithm using three threads. The programmer split the sequence into three pieces (0-499, 500-999, and 1000-1499) and used three sum variables to store the sum of each part of the sequence. In the end, a join is used to wait for all three results, and the sums are then added together for the final answer. This example teaches students the MapReduce concept, where they learn to partition a sequential problem into sub-problems, each of which can be independently solved on its own thread (map). Then, they use a join to wait for all the results to arrive (spin-synchronization). Finally, they add all the sub-sums into the total sum (reduce).

Although Figure 6 (b) is an improvement over the original sequential algorithm, it is not optimal. It provides students an opportunity to critically evaluate how a problem should be split. Currently, numbers are split into adjacent groups, meaning that larger numbers (which take longer to add together) are placed in the same group. An exercise for students may be to improve the multithreaded algorithm to have better splits between groups. For example, a more optimal split would be these three pieces: 0, 3, 6, …, 1497; 1, 4, 7, …, 1498; and 2, 5, 8, …, 1499. This more evenly distributes the workload across the threads, another important concept in multithreading. These more advanced concepts and programming issues are taught in our distributed software development course and Software integration engineering course [10].



(a) Sequential sum



(b) Multithreaded sum

Figure 6. VIPLE program implementing the sum of numbers.

VIPLE's automatic handling of thread-safety and data passing makes it a more valuable tool for introducing freshman students to the concepts of parallel computing. Currently, many students have to learn both how to change their algorithms and how to keep variables thread-safe to start programming with multiple threads. VIPLE reduces this burden, allowing students to more quickly understand the basics of parallel computing. Once students understand these basics, the idea of thread-safe variables should cause less confusion.

4.3 Event-driven programming in VIPLE

Although VIPLE can be used as a general programming language and as a parallel programming language, its strength is in event-driven programming and can respond to a sequence of events. The event-driven applications are best described by finite state machines consisting of states and transitions between the states. The transitions are triggered by events. As an example, Figure 7 shows the finite state machine for a garage door opener. The control system consists of a single button remote controller and a limit sensor. The button on the remote controller allows the door to open, stop, and close. The limit sensor is a build-in sensor in the motor. When the door stops, the limit sensor will generate a notification.

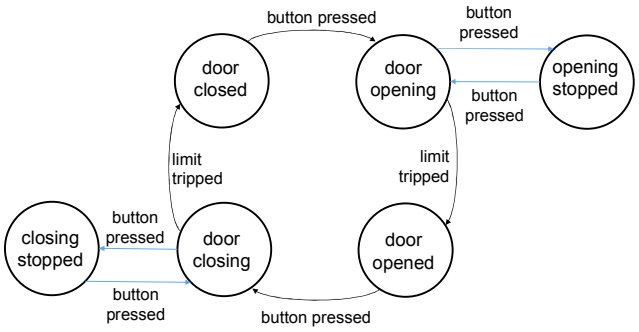
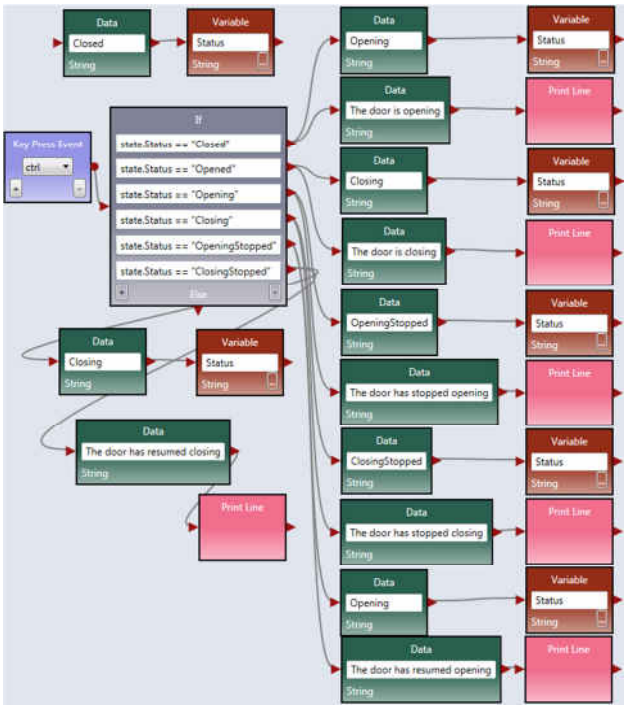


Figure 7. Finite state machine for a garage door opener with a single button remote controller

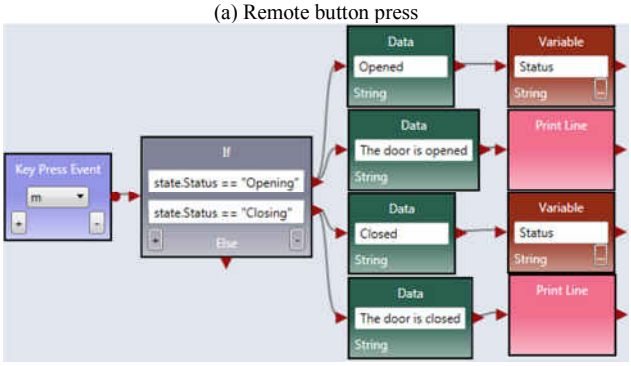
The VIPLE program that implements the finite state machine is given in Figure 8. The remote controller button is simulated by the keyboard Ctrl key. The six states are coded in the If-activity with six conditions. When the Ctrl key is pressed, the If-activity checks the current state, transits into the next state, and prints the result to the console using the Print Line service.

The limit sensor in the motor can be simulated as follows. Pressing key m will tell the system that the limit sensor has been activated. The state will change accordingly, and the result will be printed.

This program also provides an example of the parallel computing support present in VIPLE. Each activity node that has no incoming line (and is not an event) will spawn its own thread that will process the nodes on that line. Drawing multiple lines from the same outgoing pin will spawn additional threads so that each line is handled by its own thread. Finally, when an event is triggered, a new thread will be created to handle that event.



(a) Remote button press



(b) Limit sensor touched

Figure 8. VIPLE diagram implementing garage door opener

VIPLE implements its own parallel computing and service runtimes, instead of using Microsoft's Concurrency and Coordination Runtime/Decentralized Software Services. This custom implementation allows novice users to gain hands on experience with parallel computing and its complexities with a smaller emphasis on the exact syntactic details of multithreading. In addition, this runtime takes full advantage of computer hardware, allowing computation to be shared among any number of CPU cores (depending on layout of the user's VIPLE program). This use of hardware allows new users to experience firsthand the power of using parallel computing instead of sequential computing.

V. INTERFACING WITH IOT DEVICES

This section presents VIPLE's standard interfaces to IoT devices and an example of connecting VIPLE to the devices.

5.1 VIPLE's interface definition

ASU VIPLE supports an open interface to other robotic platforms. Any robot that follows the same interface and can interpret the commands from an ASU VIPLE program can work with ASU VIPLE. An ASU VIPLE program communicates with the robot using the following JSON object shown in Figure 9, which defines the input to the robot from the ASU VIPLE program and the output from the robot to the ASU VIPLE program.

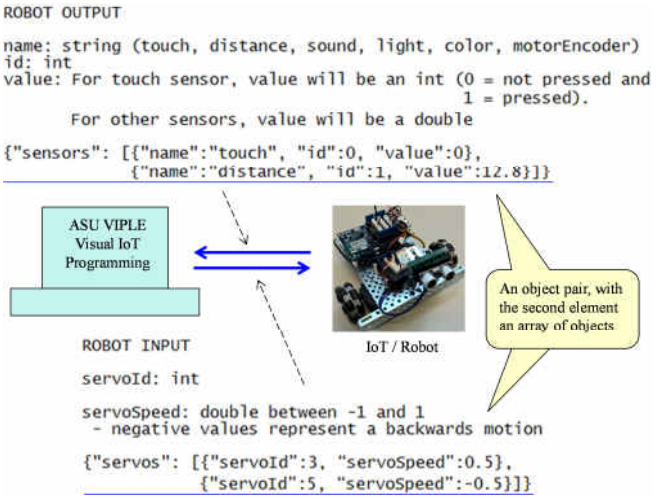


Figure 9. VIPLE code testing generic sensors

The ASU VIPLE environment encodes the control information into this object. The robot needs to interpret the script and perform the actions defined. On the other hand, the robot encodes the feedback in the same JSON format and sends the object back to the ASU VIPLE program. Then, the ASU VIPLE program will extract and use the information to generate the next actions.

5.2 Connecting VIPLE to devices

Sponsored by the Intel IoT Group, a number of robots based on the Intel architecture, including Intel's Galileo, Bay-Trail, and Edison, have been developed. ASU VIPLE can connect to these robots via Wi-Fi, TCP, Bluetooth, USB, localhost, and WebSocket interfaces, send commands to them, and control them to perform different tasks.

ASU VIPLE implemented two types of robots: generic robots and EV3 robots. A generic robot is a robot that can communicate with the computer running ASU VIPLE and can process the JSON packet. As LEGO EV3 is not an open platform, we cannot deploy code to EV3 to interpret the JSON object. Instead, we follow the EV3 open specification and use specific code in VIPLE to call EV3 APIs to implement the communication and command interpretation.

Figure 10 shows the VIPLE code for testing generic sensors connected to a generic robot.

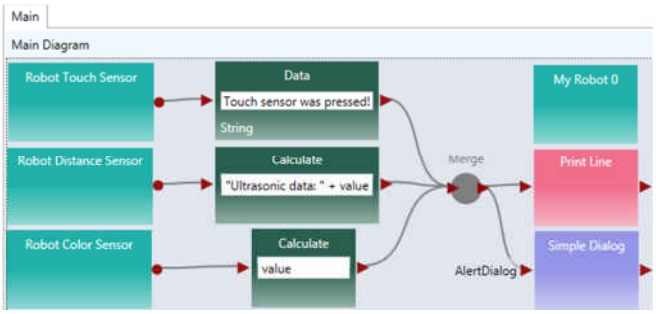


Figure 10. VIPLE code testing generic sensors

In order for the main robot, the sensors, and the motors to communicate with ASU VIPLE properly, we need to configure the partnership between the main robot and its devices, the IP address, and ports. Figure 11 shows the configuration of the three devices: main robot, drive (motors) and the distance sensor. Notice that the numbers may differ for different robot configuration.

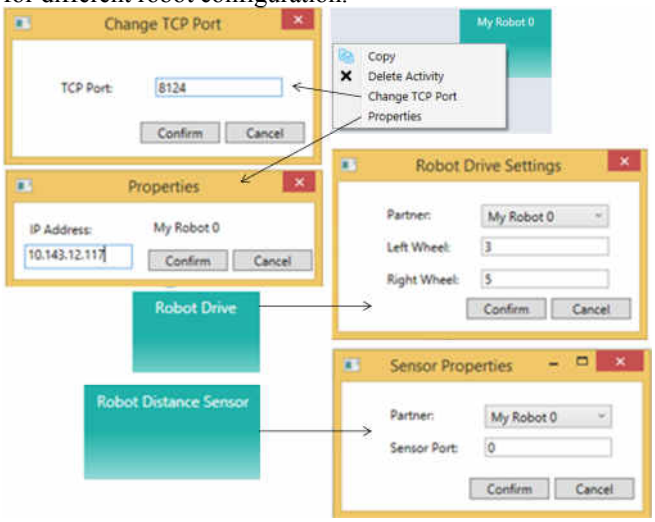


Figure 11. Configuration of IoT device ports

5.2 Maze navigation using IoT device

As a more complex example, we show the implementation of a robot navigating through a maze autonomously. Figure 12 shows the finite state machine specifying a heuristic algorithm for navigating a maze. Two variables are used in the finite state machine. The variable "Status" can take six possible string-type values: Forward, TurningRight, TurnedRight, TurningLeft (i.e. Spin180), TurnedLeft, and Resume180. The int-type variable "RightDistance" is used to store the distance to the obstacle after the robot turned right.

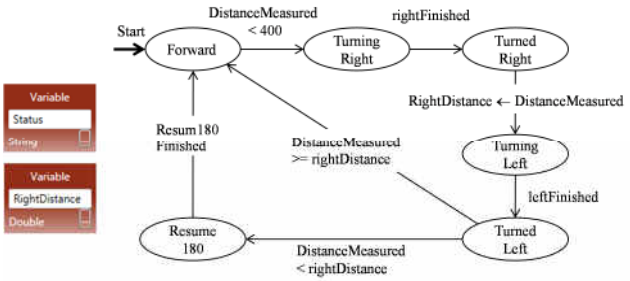


Figure 12. Finite state machine for navigating a maze

The finite state machine implements a heuristic algorithm that can be elaborated in the following steps.

1. The robot starts to move forward;
2. If the distance measured by the range sensor is less than 400 millimeters, it turns (90 degrees) right;
3. After the event "rightFinished" occurs, it saves the distance measured to the variable RightDistance;
4. The robot then spins 180 degrees left to measure the distance on the other side;
5. After the event "leftFinished" occurs, it compares the distance measured to the value saved in the variable RightDistance;
6. If the current distance is greater, it transits to the state "Forward" to move forward;
7. Otherwise, it resumes (spins 180 degrees) in the other direction;
8. Then, it transits to the state "Forward" to move forward.

The algorithm is said to be heuristic, because it cannot find a solution for all mazes. However, it has a good chance to find a solution in most mazes, given the information collected by a single range sensor.

A simulator has been developed to work with VIPLE to visualize the execution of robots in the maze. Figure 13 shows the simulated robot in a 3D maze.

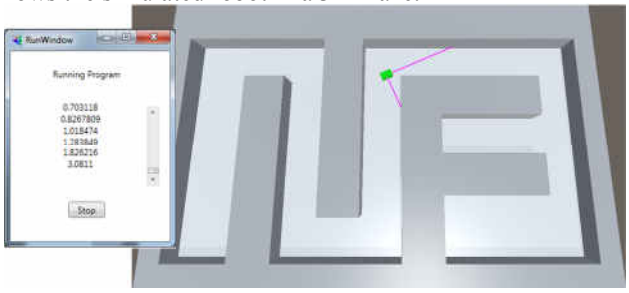


Figure 13. VIPLE simulation environment

Of course, the main reward and excitement for freshman students taking this course is to use VIPLE to program physical IoT and robotic devices. Figure 14 shows the first part of the main diagram of ASU VIPLE code that reads an Intel Edison-based robot's sensor and control its motors.

The algorithm starts with the robot moving forward. When it approaches a wall in the front, it measures the distance to the right and saves the distance into a variable. Then, the robot spins 180 degree to measure the other side's distance. It compares the two distances and moves to the direction with more space. In this part of the diagram, an If-activity is used to compare the current status and the distance value from the sensor, which generates four different cases. The second part of the main diagram is shown in Figure 15, which processes four cases of the If-activity, respectively.

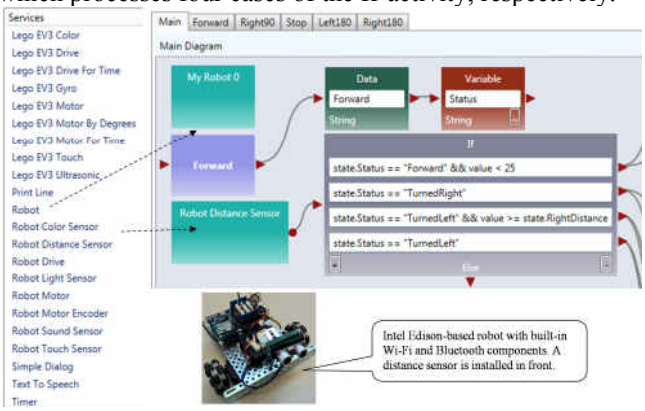


Figure 14. The first part of the main diagram implementing the two-distance maze algorithm

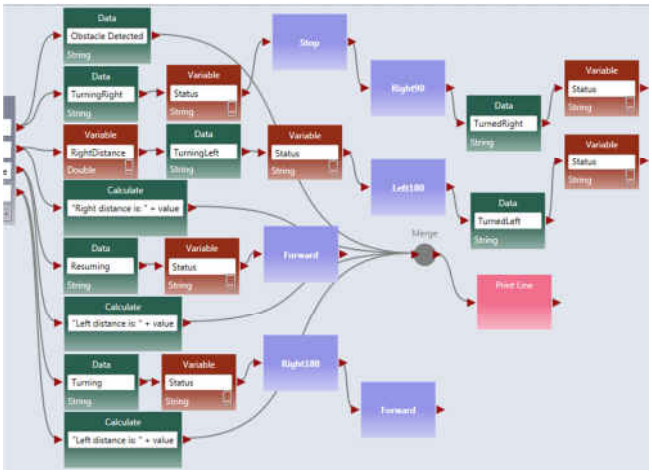


Figure 15. The second part processing the four cases

There are five activities that are implemented to support the main diagram: Forward, Right90, Stop, Left180, and Right180. Note that this example uses sequential programming after each activity (e.g. Forward) ends, instead of events. If desired, a user could create an event inside each of these activities and continue execution from the event thread instead.

A benefit of using the event method is that additional code can be handled sequentially after an activity exits normally, while the event code executes only when the activity creates the event (which can be at any user defined point). This functionality is made possible by VIPLE's support for parallel computing.

Figure 16 shows the code of three of these activities: Forward, Right90, and Stop. The code of the Right180 and Left180 are similar to Right90, but with different values. A Print Line service is used for distance values for debugging purpose.

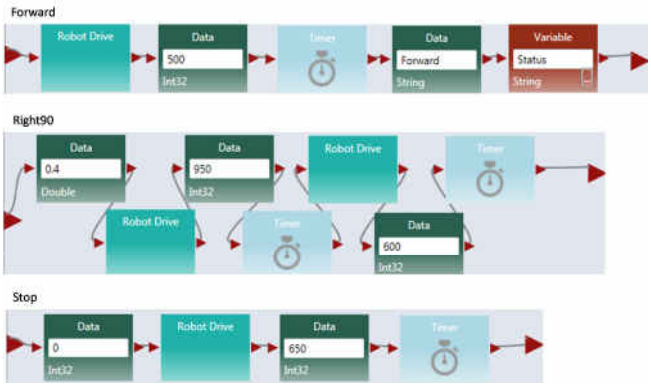


Figure 16. Codes for Forward, Right90, and Stop

VI. CONCLUSIONS

This paper presented a new visual programming language and its development environment (ASU VIPLE) for programming IoT devices and robots. ASU VIPLE extends the discontinued Microsoft VPL to help the community with their VPL projects. ASU VIPLE supports LEGO EV3 and all IoT devices based on an open architecture. ASU VIPLE is free and well documented at:

http://venus.eas.asu.edu/WSRepository/VIPLE/

ASU VIPLE has been pilot-tested at Arizona State University and several other universities.

ACKNOWLEDGMENT

The authors wish to acknowledge those who contributed and helped in conceptualization and development of ASU VIPLE. Particularly, Dr. Yann-Hang Lee co-advised some of the student teams working on the related projects. Garrett Drown developed the first version of the eRobotics visual programming tool. Garret Walliman implemented the second version of the visual programming language. Calvin Cheng helped in the early development of VIPLE and contributed the EV3 APIs for VIPLE. Tara De Vries contributed to the service integration of VIPLE development. Megan Plachecki, John Robertson, and Sami Mian contributed to the JSON interface design, the implementation of the middleware on the Edison robot, and the robot hardware design. Matthew De Rosa developed the VIPLE simulator.

REFERENCES

[1] IoT in Wikipedia, http://en.wikipedia.org/wiki/Internet_of_Things

[2] Stephan Haller, http://services.future-internet.eu/images/1/16/A4_Things_Haller.pdf, May 2009.

[3] Yinong Chen, Hualiang Hu, "Internet of Intelligent Things and Robot as a Service", Simulation Modelling Practice and Theory, Volume 34, May 2013, Pages 159–171.

[4] Ton Steenman, GM, Intel Intelligent Systems Group: "Accelerating the Transition to Intelligent Systems", Intel Embedded Research and Education Summit, February 2012, http://embedded.communities.intel.com/servlet/JiveServlet/downloadBody/7148-102-1-2394/Accelerating-the-Transition-to-Intelligent-Systems.pdf.

[5] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic, "Cyber Physical Systems: The Next Computing Revolution", 47th Design Automation Conference (DAC 2010), CPS Demystified Session, Anaheim, CA, June 17, 2010.

[6] Wikipia, Cyber-physical system, http://en.wikipedia.org/ wiki/ Cyber-physical_system

[7] Jian Huang, Farokh Bastani, I-Ling Yen, and Wenke Zhang, "A Framework for Efficient Service Composition in Cyber-Physical Systems", In Proceedings of 5th IEEE International Symposium on Service Oriented System Engineering (SOSE), Nanjing, June 2010.

[8] NSF solicitation, "Cyber-Physical Systems", 2008, www.nsf.gov/pubs/2008/nsf08611/nsf08611.pdf

[9] Yinong Chen, Zhihui Du, and Marcos Garcia-Acosta, M., "Robot as a Service in Cloud Computing", In Proceedings of the Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE), Nanjing, June 4-5, 2010, pp.151-158.

[10] Yinong Chen, W.T. Tsai, Service-Oriented Computing and Web Software Integration, 5th Edition, Kendall Hunt Publishing, 2015.

[11] Yoshiaki Kakuda, "Assurance networks: concepts, technologies, and case studies," Proc. Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing (UIC-ATC 2010), Xi'an, China, October 2010, pp.311-315.

[12] Kinji Mori, "Autonomous Decentralized System and Its Strategic Approach for Research and Development", Invited Paper, Special Issue on Autonomous Decentralized Systems Theories and Application Deployments, IEICE Transactions on Information and Systems, Vol.E-91-D, No.9, pp.2227-2232, Sept. 2008.

[13] Autonomous Decentralized Systems (ADS) in Wikipedia: https://en.wikipedia.org/wiki/Autonomous_decentralized_system

[14] M. Malek: Responsive Systems: A Marriage Between Real Time and Fault Tolerance, Keynote Address, Proceedings of the Fifth International Conference on Fault-Tolerant Computing Systems, Nuernberg, Germany, Springer-Verlag, Informatik-Fachberichte 283, 1-17, September 1991.

[15] Chris Gill, Jeanna M. Gossett, David Corman, Joseph P. Loyall, Richard E. Schantz, Michael Atighetchi, and Douglas C. Schmidt, Integrated Adaptive QoS Management in Middleware: An Empirical Case Study, submitted to the 24th IEEE International Conference on Distributed Computing Systems (ICDCS), May 23-26, 2004, Tokyo,.

[16] DOC, Distributed Object Computing Group for Distributed Real-time and Embedded (DRE) Systems, http://www.dre.vanderbilt.edu/

[17] Hiroshi Nakagawa, Kazuyuki Nakamaru, Tomoyuki Ohta, Yoshiaki Kakuda, "A hierarchical routing scheme with location information on autonomous clustering for mobile ad hoc networks", ISADS 2009: 269-274.

[18] Yinong Chen, W.T. Tsai, "Towards dependable service-orientated computing systems", Simulation Modelling Practice and Theory, Volume 17, Issue 8, September 2009, Pages 1361-1366.

[19] Ioannis A. Moschakis, Helen D. Karatza, "Evaluation of gang scheduling performance and cost in a cloud computing system, the Journal of Supercomputing 59(2): 975-992 (2012).

[20] Zafeirios C. Papazachos, Helen D. Karatza, "Performance evaluation of bag of gangs scheduling in a heterogeneous distributed system", Journal of Systems and Software 83(8): 1346-1354 (2010).

[21] Georgios L. Stavrinides, Helen D. Karatza: Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques. Simulation Modelling Practice and Theory, 19(1) 2011, pp. 540-552.

[22] App Inventor, http://ai2.appinventor.mit.edu/

[23] Alice, http://www.alice.org/

[24] Microsoft Robotics Developer Studio, https://msdn.microsoft.com/en-us/library/bb648760.aspx

[25] MRDS in Wikipedia https://en.wikipedia.org/wiki/Microsoft_Robotics_Developer_Studio

[26] ASU eRobotics programming environment, http://venus.eas.asu.edu/WSRepository/erobotic/

[27] DPWS in Wikipedia, https://en.wikipedia.org/wiki/Devices_Profile_for_Web_Services