Visual programming language environment for different IoT and robotics platforms in computer science education

Gennaro De Luca 1, Zhongtao Li 2, Sami Mian 1, Yinong Chen 1

1School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85287-8809, USA
2School of Information Science and Engineering, University of Jinan, Shandong, People’s Republic of China
E-mail: gdeluca1@asu.edu

Abstract: This study presents the authors’ recent research and application of a new visual programming language and its development environment: VIPL (Visual IoT/Robotics Programming Language Environment) at Arizona State University (ASU). ASU VIPL supports a variety of IoT devices and robots based on an open architecture. Based on computational thinking, VIPL supports the integration of engineering design process, workflow, fundamental programming concepts, control flow, parallel computing, event-driven programming, and service-oriented computing seamlessly into a wide range of curricula, such as introduction to computing, introduction to engineering, service-oriented computing, and software integration. It is actively used at ASU in several sections of FSE 100: Introduction to Engineering and in CSE 446: Software Integration and Engineering, as well as in several other universities worldwide.

1 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 tags to mark the Electronic Product Code (Auto-ID Lab), which is extended to refer to the systems 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 devices that have adequate computing capacity, such as a 32-bit processor. Distributed intelligence is a part of the IoT [3], where local computing resources are used to process the data locally and thus offload the central computing units. 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, which are considered intelligent devices. Among the four billion intelligent devices, one billion devices are intelligent systems that are embedded in smart devices [4].

The development of cloud computing, fog computing, and edge computing pushed the desktop-based computing platform into an Internet-based computing infrastructure. It also seamlessly connects physical devices or things into services.

In addition to IoT and IoIT, a number of related concepts and systems have been proposed to take advantage of the 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 [5–7]. CPSs can be considered the extended and decentralised version of embedded systems. In CPSs, the computation and communication core and the physical elements are tightly coupled and coordinated to fulfil a coherent mission. Research advances in CPSs promise to transform our world with these kinds of systems. They can respond more quickly, for example, autonomous collision avoidance. They are more precise, e.g., robotic surgery and nano-tolerance manufacturing. They work in dangerous or inaccessible environments, e.g., autonomous systems for search and rescue, firefighting, and exploration. They provide large-scale, distributed coordination, e.g., automated traffic control. They are highly efficient, e.g., zero-net-energy buildings, augment human capabilities, and enhance societal wellbeing, for example, assistive technologies and ubiquitous healthcare monitoring and delivery [8].

An autonomous decentralised system (ADS) is a system composed of decentralised 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, where data are not sent to an address, but sent by a content code. The receivers read the content code and decide if or not to use or to discard the data. 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 decentralised systems have a number of applications including industrial production lines, transportation systems, railway signalling and robotics applications [9, 10].

The robot as a Service (RaaS) is a cloud computing unit, particularly a fogging unit that facilitates the seamless integration of robots and embedded devices into Web and cloud computing environments [3, 11]. It follows the service-oriented architecture (SOA), and a RaaS unit includes services for performing functionality, a service directory for discovery, and service clients for the user’s direct access [12, 13]. The current RaaS implementation facilitates SOAP and RESTful communications between RaaS units and the other cloud computing units. Hardware development and interface 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 IoT, IoIT (which have adequate computing capacity to perform complex computations [3]), a CPS (which is a combination of a large computation and communication core and physical elements that can interact with the physical world [4]), and an 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 Scratch and MIT App Inventor [14] uses drag-and-drop style puzzles to construct games or phone applications. University of Virginia and Carnegie Mellon’s Alice is a 3D game and movie development environment [15]. 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. Intel IoT Service Orchestration Layer is Visual/Workflow approach for delivering a robust, extensible, high-quality solution for rapid creating IoT Apps Work seamlessly with Intel IoT boards, such as Galileo and Edison [16]. Microsoft Robotics Developer Studio (MSRDS) and Visual Programming Language Environment (VPLE) are specifically developed for robotics applications [17]. It was 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 an 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 CSE101 in 2006 as freshman level computer science and engineering course. The course grew from 70 students in Fall 2006 to over 350 students in 2011. The course was then 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 majoring at ASU. The course is also adopted in many universities around the world, including China.

Unfortunately, Microsoft suspended its development and support for MSRDS and VPL in 2014 [18], which leads to our FSE100 course, and many other schools’ courses using VPL, without further support. Particularly, the existing 2012 version of VPL support LEGO NXT, but not the LEGO’s third generation EV3 robot, while the second-generation NEX is off the market.

Through our own development, 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 developed in 2012–2014. VIPLE 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.

VIPLE is a general-purpose programming language and is Turing complete in its capacity. It can be used for any kind of computational tasks. Thus, VIPLE is a perfect language for students to learn their first programming languages. VIPLE uses visual programming style based on drag and drop of existing components and services. This programming style allows students to focus on the logic, instead of syntax.

This paper is an extension to our previous workshop publication on Visual IoT/Robotics Programming Language Environment for Computer Science Education [19]. More than 70% of the contents is now in this paper. The reused parts are mainly the related work and overall system description.

The remaining of the paper is organised as follows. Section 2 introduces the basic IoT standards, protocols and the stack of the technologies related to IoT. Section 3 presents the main functionalities of ASU VIPLE and how it is used for general purpose computing and visually programming IoT and robotics devices. Section 4 presents an array of examples to show VIPLE applications in computer science education. Section 5 describes the interface and connection from VIPLE to IoT devices. Section 6 offers a variety of concepts that are taught with VIPLE and its runtime. Section 7 discusses the use of VIPLE in the classroom. Section 8 concludes the paper.

### 2 VIPLE programming concepts

VIPLE provides a platform upon which a variety of computing and engineering concepts may be taught. In addition to the IoT and Robotics topics mentioned, VIPLE is also used to target a variety of other concepts due to its ease of use and strong parallel and distributed computing (PDC) support.

#### 2.1 Introduction to the VIPLE runtime

One of the major requirements set forth for VIPLE is a strong support for multithreading, event-driven programming, and robotic/IoT device connectivity. To this end, we developed a new runtime to both accurately meet these requirements and to facilitate the expansion of the runtime. The key features of this runtime include all those discussed in Section 3, such as simple multithreaded application development with in-built thread safety and support for any open platform IoT or robotic device.

One of the challenges we faced in developing this runtime was preventing race conditions such as in the maze navigation algorithm we require students to write. This algorithm makes use of two sensors which may receive data at the same time. The branching of code associated with one sensor commands many other requests a right turn. However, the runtime cannot enforce locks that prevent two blocks of code from running at once, even if only from sensors, as that would conflict with the multithreading requirements. Our solution includes the use of locks that are automatically invoked when using global variables in VIPLE. By having students use such a variable in both sensor threads, the runtime can help coordinate the tasks without the user having to understand exactly how the race condition is prevented.

#### 2.2 VIPLE runtime as a service

Developing a custom runtime facilitated the use of the runtime in other applications outside of a desktop application. As this runtime has strong support for teaching PDC topics, we have used it for other related applications. As one example, this runtime allows simple development of a web-based interface for VIPLE. Such an application has several benefits. Students may take applications they developed on the desktop application and run it on the web application. Transferring the application provides an opportunity to teach students the difference between desktop and web-based multithreaded application development. One major difference may include support for high-performance computing on the web application, as the runtime may be run on a server farm, whereas the desktop version is limited to the local computing resources.

#### 2.3 VIPLE optimisations

In an effort to improve the performance of VIPLE, a variety of research was performed to explore the effect of standard and novel optimisation techniques on the VIPLE runtime. These optimisations range from the use of thread pools to the use of covering arrays to provide branch prediction in complex branch activities [20]. Such optimisations are beyond the scope of a VIPLE program since they are targeted at the runtime. As such, students are unable to experiment with these changes. However, in upper division classes, we may explain these various optimisations and have the students explore these effects in their own work. The use of interesting optimisation techniques such as covering arrays has helped to increase student interest and interaction as many of our classes typically focus on antiquated or trivial optimisation techniques.
To provide a solid foundation to VIPLE and also for the upper division students suitable skills for the work they do in VIPLE, VIPLE constructs were defined based on \( \pi \)-calculus [21\textsuperscript{-21}]. We can represent all possible VIPLE activities in \( \pi \)-calculus, and thus prove the correctness of VIPLE program using the available models and tools in the mathematics. The VIPLE constructs provide a variety of benefits in the teaching process and parallel computing concepts that exist in \( \pi \)-calculus and thus can teach these concepts in visual programming. For example, the constructs help to implement and explain map and reduce concept, since a typical map reduce application can be represented in \( \pi \)-calculus using identical constructs. By asking students to write out the mathematical representation of their VIPLE program, they can be shown such similarities. Furthermore, this model provides an opportunity to explore and prove a variety of mathematical and theoretical computing concepts. Fig. 1 lists the basic constructs in \( \pi \)-calculus, their graphic representations, and the VIPLE activities that implement the constructs.

In our theoretical computing courses, students may explore concepts using the Turing Machine model. Introducing a \( \pi \)-calculus model allows students to explore other advanced concepts that would be difficult or impossible with Turing Machines. The main difference between the two models is that \( \pi \)-calculus focuses more on the connections and interfaces between various entities [21]. As such, it is possible in many cases to prove whether their program is structurally equivalent to another program, if their program can be reduced to a simpler form, and then how to create the VIPLE program corresponding to that simpler form [21].

### 3 IoT standards and protocols

IoT is now 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 Fig. 2, the IoT connects to the Internet through the 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 organised into ontology and presented in RDF, RDFS, and OWL for storing, analysing, 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.

On the other hand, IoT and IoIT are 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 [9, 10]. Any other devices that need to delay information will pick up the data by content code and apply the delay information.

The 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 [24]. 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 SOAP and RESTful Web service standards at the device level, similar to 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. RaaS also allows the IoT controller or the robot to be configured as the Wi-Fi router, allowing multiple applications to connect to the device.

In the next section, we will outline the design and implementation of VIPLE and its interface to the Internet and to IoT devices.
4 VIPLE activities and services

The basic building blocks of an ASU VIPLE program (diagram) are listed and explained in Fig. 3. They offer a complete set of activities described in process control theory, such as n-Calculus [20, 21].

One of the main development goals with VIPLE is the usability of the language. To this end, a variety of library functions and services are made available in addition to the standard language functions. These functions and services are split into three sets, as shown in Fig. 4.

The first set of services contains the ‘General Purpose Services’. These services include general input and output services, general event services, and other various services to facilitate application development. The user may also add services to this section, including VIPLE services (created from a Custom Activity) and WSDL services.

The second set of services contains the ‘Robot/IoT Services’. These services allow reading and writing of data on devices which offer an open-standard interface. VIPLE supports a variety of communication protocols, including WebSocket, TCP, UDP, and Bluetooth. To facilitate the use of new devices with VIPLE, a standard interface is defined in JSON that allows specification of the various services offered by the IoT device. The use of this interface allows a VIPLE application to work seamlessly across a variety of devices with very few to no changes in the VIPLE code.

The third set of services includes any vendor-specific services. Robots which lack support for an open communication and programming interface, such as the Lego EV3, Lego NXT, and iRobot, must be supported on a case by case basis. To support such devices, VIPLE offers a set of built-in services that can communicate with the standard firmware of those devices, without requiring additional programming. Currently, the services for Lego EV3 are available in VIPLE. These services allow VIPLE to interface with all EV3 sensors as well as EV3 drive-motors and arm-motors. The full list is shown in the third part of Fig. 4. This support for EV3 devices is significant, as it allows the Microsoft VPL developers who used NXT robots to use the new EV3 robots.

One of the major advantages of using VIPLE over Microsoft VPL is the support for any device with an open platform. To connect to a generic device with Microsoft VPL, DSS services written specifically for MRDS must be added to the VPL service list. In VIPLE, any robot that follows the standard JSON interface will work with the services currently offered. In addition to greater robot support, VIPLE offers access to a variety of web services. As many Web applications make use of RESTful services, VIPLE can access a wide range of resources. Instead of the generic robot services offered by Microsoft VPL, VIPLE offers several simulator unique services that allow significantly quicker prototyping of algorithms for IoT devices and robots.

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

5 IoT and robotics platforms

A number of robot units have been implemented based on different hardware platforms, as shown in Fig. 5.

5.1 Simulators

The robot units can be physical simulated. Both a desktop simulator and Web simulators are developed. The desktop simulator is implemented using the Unity 3D game engine, which runs on the same computer as the VIPLE program. The Web simulators are implemented in HTML/JavaScript, which runs in a Web browser on any computer or mobile device. One Web simulator is a 2D version, while the other is a 3D version, depending on the user’s preference. Fig. 6 shows the three simulators. The simulators consist of two parts: the robot and the environment. We can
choose the robot to be a humanoid robot or a vehicle, and the environment is a maze that can be reconfigured dynamically while the robot is moving. We can create one robot or multiple robots in the environment. The 3D simulators can be changed dynamically while the program is running and the robot is traversing the maze.

VIPLE programs can start with no simulator. In this case, either no device is needed, or the physical devices will be used. VIPLE programs can also start with one of the simulators to visualise the movement of the robot in the environment.

5.2 Physical robots

Different hardware platforms have been developed in addition to the EV3. A specific set of services are developed to work with Lego EV3 as our open source middleware cannot be installed on EV3. For all the other open architecture platforms, a standard interface in JSON format has been defined and a middleware has been developed, which can be executed on all these platforms. A school can build the robots for the course based on the open source middleware or purchase the prebuilt robots from a vendor that builds such robots. For example, the pcDuino robots based on ARM architecture and built to work with ASU VIPLE can be purchased from Link Sprite at http://store.cutedigi.com/pcduino-asu-viple-robot/.

The boards we use for building the robot units are IoT boards, with both processor and Arduino connectors, which makes the construction very easy. Fig. 7 shows the hardware design and connection based on Intel’s Edison board. No additional hardware or logic is needed, simply connect the sensors and motors to the board. We can have the Linux operating system and the driver library installed on the Edison board by following Intel’s Edison’s software installation guide at https://software.intel.com/en-us/articles/flash-tool-lite-user-manual.

It is possible to build the robot without using additional circuits or boards, as shown in Fig. 7. However, Edison boards do not provide ground pins and require welding multiple lines together. An alternative is to use a breadboard to extend the ground pins, as shown in Fig. 8.

5.3 Communication between VIPLE and robots

The communication between VIPLE and all the platforms are defined by a standard protocol shown in Fig. 10. Since VIPLE is designed to work with a variety of IoT devices, a number of challenges have been identified with the use of IoT platforms. Some of these problems were easy to correct using VIPLE-based changes or additions, while other problems require user input and tuning in order to circumvent.

One of the biggest problems with IoT devices is establishing and maintaining a reliable wireless connection. For our platforms, this is done using either a Bluetooth connection or a Wi-Fi connection. On
most embedded devices, it is a lengthy process to find and connect to a network, especially after multiple reboots or power failures. With our work, we have created solutions to help curb these connection issues. VIPEL provides support for the Bluetooth stack, making it easy for a user to connect to Bluetooth devices using the host computer’s standard Bluetooth interface. With respect to Wi-Fi, the runtime software on IoT microcontrollers is setup to automatically connect to a known Wi-Fi network and to keep that connection open at all times. In order to facilitate connection to the robot via the host computer, a script was also created to automatically detect and display the platform’s connection information on an LCD screen. By using compressed JSON packages to communicate between VIPEL and IoT devices, a low bandwidth connection is fine for basic tasks, such as a robot autonomously navigating a maze.

Another problem with these platforms, in general, is the large variance in hardware inconsistencies, both with IoT devices/ sensors and actuators. For example, many of the cheap ultrasonic sensors originally used to test VIPEL’s robotics packages included a large amount of noise in their data, and some even returned false data completely (likely due to faulty components). Servo motors purchased in large quantities can vary in terms of speed and digital control; sometimes you will have two servos connected to the same control signal spanning at different speeds. However, using VIPEL and some small software changes, students are able to overcome most of these problems. For sensors with low amounts of noise, students widen the parameters for decision making based on sensor input, allowing for a larger margin of error. For servos and other actuators, students have managed to compensate for inconsistencies by varying the power and intensity of the signals for individual outputs. Calibration modules have also been developed in VIPEL to allow for students to tune and modify individual actuators in order to produce similar input/ output behaviours (usually maximum speed).

VIPEL was initially designed to work with Intel-based IoT devices: the Galileo, Edison, Curie, and Minnow boards. However, support has increased to a number of other popular boards, including the Arduino branded boards (Uno, Mega, Due, etc.) as well several ATmega boards. In addition to these boards, VIPEL is compatible with any boards that can connect to VIPEL over one of the aforementioned interfaces (e.g. Wi-Fi) and can process the required JSON objects.

6 Educational examples implemented in VIPEL

VIPEL can be used as a general-purpose programming language and is Turing complete in its capacity. It can be used for any kind of computational tasks. Thus, VIPEL is a perfect language for students to learn their first programming languages.

In VIPEL, a program is represented as a diagram of workflow, and the components are defined as activities in the diagram.

6.1 Teaching fundamental programming in VIPEL

We now start to show examples of basic programming techniques in ASU VIPEL. Fig. 11 shows a simple ‘Hello World’ type of program.

In this simple example, VIPEL automatically changes the type to string after a string is entered, and the default null value step in Microsoft VPL is eliminated. The program illustrates three types of output methods.

As a more complex example, we show a counter using a modular design, where a module is implemented as an activity. An activity can take input parameters and generate output as the return value.

Fig. 12 shows the VIPEL program that implements: Fig. 12a a main diagram and Fig. 12b 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 VIPEL, as is the ability to define any number of named inputs to an activity. Similar to Microsoft VPL, a custom VIPEL activity can be converted to a VIPEL service, allowing sharing of modules with other programmers.

6.2 Teaching parallel computing concepts in VIPEL

VIPEL 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 synchronisation and thread-safety or how to pass data to or from threads. The counter example in Fig. 12 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.

Fig. 13 shows two ways to sum all the numbers from 0 to $n$ ($n = 1499$ in this example). The diagram first initialises all variables to 0. Fig. 13a shows 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. In the end, sum will contain the final answer.

In contrast, Fig. 13b 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 Fig. 13b shows 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 [12].

As an exercise, students are asked to modify the examples to add all the numbers from 0 to 1500 (instead of 1499) and time the operation in milliseconds. As shown in the answer key in Figs. 14a–c, there is a significant speedup switching from a sequential sum to a multithreaded sum. Furthermore, performing the optimisation technique mentioned above also provides a speedup, albeit not as significant. Using these basic examples, students can be asked to explore various other optimisation
techniques and be taught concepts such as the point at which thread overhead exceeds the advantages offered by multithreading. To further improve the performance, a multithreading optimiser is also implemented, and the result is given in Fig. 14c.

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.

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 the layout of the user’s VIPLE program). This use of hardware allows new users to experience first-hand the power of using parallel computing instead of sequential computing.

6.3 Event-driven programming in VIPLE

In addition to the features illustrated in the previous section that VIPLE can be used as a general programming language and as a parallel programming language, VIPLE’s strength is in event-driven programming and can respond to a sequence of events in IoT and robotics application. The event-driven applications are best described by finite state machines consisting of a finite number of states and transitions between the states. The transitions are triggered by events. As an example, Fig. 15 shows a typical traffic light system at a four-way intersection.

Fig. 16 shows the finite state machine that models the behaviours of the system. The control system consists of a number of timers that regulate the length of each state and two emergency signals E Pass and N Pass, for switching the lights to allow the traffic in east-west (E-W) traffic and north-south (N-S) traffic to pass, respectively. The emergency signals can be triggered by police and ambulance vehicles. More complex traffic light system can be implemented by adding more states and more inputs.

The finite state machine can be represented in a state transition table as shown in Table 1.

A traffic simulator has been implemented in Unity game software that visualises the traffic lights, with cars moving through the traffic lights. One car is programmable in VIPLE, and other cars are random are generated and they mode automatically in the simulator. Fig. 17.

### Table 1 State transition table

<table>
<thead>
<tr>
<th>Current state</th>
<th>Description</th>
<th>Event input</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>green red</td>
<td>timer 1</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>yellow red</td>
<td>EW pass</td>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
<td>red red</td>
<td>time 2</td>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
<td>red green</td>
<td>timer 4</td>
<td>S5</td>
</tr>
<tr>
<td>S5</td>
<td>red yellow</td>
<td>timer 5</td>
<td>S6</td>
</tr>
<tr>
<td>S6</td>
<td>red red</td>
<td>timer 6</td>
<td>S1</td>
</tr>
</tbody>
</table>
shows a screenshot of the simulator, where the N-S direction is in red, while the E-W direction is in green.

The VIPLE program that implements the finite state machine is given in Fig. 18. A variable ‘light’ is used for storing the states, which is initialised state ‘S1’. On the top-left, the six states of the finite state machines are coded in the If-activity with six conditions. The timers are initialised with different time lengths. After a timer expires, the state is changed to the next state and the state is printed on the console window using the Print Line service. The emergency signals are simulated by two keypress events on the bottom-right of Fig. 18. When the key ‘e’ is pressed, the If-activity checks the current state, transitions into the next state based on the finite state machine to make E-W sides of the traffic light green. Similarly, when the key ‘n’ is pressed, the If-activity checks the current state and transitions into the next state based on the finite state machine to make N-S sides of the traffic light green.

7 Interfacing with IoT devices

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

7.1 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 Fig. 10 in the previous section, which defines the input to the robot from the ASU VIPLE program and the output from the robot to the ASU VIPLE program.

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.

The VIPLE does not include the logic to control the 12 traffic lights, which can be simulated using Print Line service. We are developing the traffic light simulator that can graphically demonstrate the lights changing colours.

7.2 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 vendor-specific robots, such as LEGO 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 the specific code in VIPLE to call EV3 APIs to implement the communication and command interpretation.

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. Fig. 19 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.

7.3 Maze navigation using IoT device

As a more complex example, we show the implementation of a robot navigating through a maze autonomously. Fig. 20 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.

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.

For better support different needs of different classes, multiple simulators have been developed. Among them, Unity-implemented maze simulator allows the VIPLE program to visualise the execution of robots in the maze. Fig. 21 shows the simulated robot in a 3D maze.

Of course, the main reward and excitement for freshman students taking this course are to use VIPLE to program physical IoT and robotic devices. Fig. 5 presented all the types of physical robots developed at AU to support our classes. Fig. 22 (first part) and Fig. 23 (second part) show the main diagram of the ASU VIPLE code that uses ARM-based Raspberry Pi, robot and use generic sensors and 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° 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 in Fig. 23 processes four cases of the If-activity, respectively. In the main diagram, a Print Line service is used for printing the distance sensor values for debugging purpose. A merge activity is used share the print service from multiple sources.

There are six activities, as can be seen in Fig. 22, which are implemented to support the main diagram: Forward, Right90, Stop, Left180, Right180, and Init. 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.

Fig. 24 shows the code of three of these activities: Forward, Left180, and Init. The code of the Right90, stop, and Right180 are similar to the given activities in Fig. 24, but with different values. A Print Line service is used in Forward to print Status for debugging purpose.

7.4 Multi-robot systems with VIPLE

One of the new improvements that separates VIPLE from its predecessor is the ability to connect to multiple platforms at once. This includes both IoT devices and entire robot platforms. This capability allows VIPLE to connect to a distributed network of sensors, which allows the user to simulate a distributed network of systems. A prime example is using a group of robots to autonomously transverse and map a maze. VIPLE acts as a central planner for the robot ‘swarm,’ where sensor data on the maze is sent back and analysed from each robot. By utilising several robot platforms as one, just as a distributed computing system, the maze transversal is completed in a shorter amount of time. This also allows the central planner to use a more accurate path planning algorithm for the maze due to the increased amount of information. Users can also use VIPLE to simulate a decentralised system of computers, by merely acting as a node in the network of connected robot platforms. Users can use this network to test decentralised path planning algorithms on the robots.

The use of VIPLE is ideal for distributed computing education. The VIPLE environment takes care of all the communication protocols and setup usually needed in a decentralised computing environment and instead uses the environment’s runtime as the main connection hub. Students can simulate different types of the multi-system environment by either using the built-in simulator with several robot instances or by connecting to a host of different IoT devices, once again each acting as a standalone system.

8 VIPLE in the classroom

In order to validate the effectiveness of VIPLE as an educational tool, several classes and camps have started to utilise VIPLE as the main programming environment, along with using curriculum developed by the authors specifically for VIPLE. These classroom trials have resulted in important feedback from users, which has led to numerous improvements in the VIPLE software and updates to the accompanying curriculum.

8.1 VIPLE in the intro to programming camps

The first use of the VIPLE program was in a selection of summer robotics camps held at ASU, led by the authors of this paper. Students aged 14–18 were given a basic overview and introduction to programming using VIPLE and Lego EV3 robots. Lectures on a variety of computer science topics were accompanied by exercises designed to teach these students the fundamentals of computational thinking, with a focus on devising algorithms and translating their thoughts and English sentences into a format the machine can understand. Further exercises proceeded to use this knowledge to write an algorithm to guide a robot through a random maze autonomously, as mentioned earlier in this paper. While our previous courses that used text-based programming languages focused a large amount of effort on syntax, VIPLE eliminates this necessity and allows us to focus on teaching computational thinking.

8.2 VIPLE in the introduction to engineering courses

After VIPLE was successfully tested and used in a handful of these summer camps, it was decided to integrate the use of VIPLE in several university level programming courses, including the Introduction to Engineering course at ASU. In this course, VIPLE was used in the same way as the summer camps, as an introduction to computational thinking. However, the lecture materials contained more advanced concepts, and the exercises in VIPLE focused on a wider assortment of programming techniques, including finite state machine design and parallel programming (especially used in interfacing with robots with multiple sensors). Thanks to the feedback from these courses, many significant changes were made to VIPLE, including the rewrite of the VIPLE runtime.

8.3 VIPLE in advanced software courses

In addition to introductory courses, VIPLE has also been used in several upper division courses, such as the Software Integration and Engineering course at ASU. In these courses, VIPLE is used to teach a variety of higher level concepts, especially PDC topics. As discussed in the \(\pi\)-calculus section, VIPLE is introduced in our section on Hadoop, showing similarities and allowing students to gain additional experience. VIPLE is also used in the earlier sections on multithreading and workflow programming. Its main use in the course is on developing multithreaded robotics and IoT applications, especially those that make use of more than one robot.

8.4 VIPLE abroad

Due to the success of VIPLE as a programming environment for introductory courses, the software has started to be used in other universities, mainly in China and Vietnam. In these locations, VIPLE is used primarily as an introductory programming language, with a special focus on robotics and IoT devices. The authors have held several workshops, both at ASU and abroad, in order to train new faculty on the use of VIPLE and its curriculum, e.g. through HEEAP [25]. This widespread use of VIPLE to teach both introductory and advanced programming topics has provided a great deal of valuable feedback and has resulted in several updates to the VIPLE software, as well as added support for new languages.

9 Conclusions

This paper presented a visual programming language and its development environment (ASU VIPLE) for programming IoT and robotics devices. Simulation environment, as well as physical devices, are tested and put in actual classrooms. 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. It added many more functions so that the platform can be used in more advanced classes, including service-oriented computing, parallel and distributed computing, and workflow-based software integration. ASU VIPLE is free and well documented at http://neptune.fulton.ad.asu.edu/VIPLE/. ASU VIPLE has been tested at ASU FSE100 class and several other universities since 2015.

10 Acknowledgments

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, and John Robertson 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 first version of the VIPLE Unity simulator.

11 References

[16] Intel IoT Service Orchestration Layer. Available at https://github.com/01org/intel-iot-services-orchestration-layer