Explainable AI in Workflow Development and Verification Using Pi-Calculus

by

Gennaro De Luca

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved October 2019 by the
Graduate Supervisory Committee:

Yinong Chen, Co-Chair
Huan Liu, Co-Chair
Sharon Hsiao
Dijiang Huang

ARIZONA STATE UNIVERSITY

December 2019

ABSTRACT

Computer science education is an increasingly vital area of study with various challenges that increase the difficulty level for new students resulting in higher attrition rates. As part of an effort to resolve this issue, a new visual programming language environment was developed for this research, the Visual IoT and Robotics Programming Language Environment (VIPLE). VIPLE is based on computational thinking and flowchart, which reduces the needs of memorization of detailed syntax in text-based programming languages. VIPLE has been used at Arizona State University (ASU) in multiple years and sections of FSE100 as well as in universities worldwide. Another major issue with teaching large programming classes is the potential lack of qualified teaching assistants to grade and offer insight to a student's programs at a level beyond output analysis.

In this dissertation, I propose a novel framework for performing semantic autograding, which analyzes student programs at a semantic level to help students learn with additional and systematic help. A general autograder is not practical for general programming languages, due to the flexibility of semantics. A practical autograder is possible in VIPLE, because of its simplified syntax and restricted options of semantics. The design of this autograder is based on the concept of theorem provers. To achieve this goal, I employ a modified version of Pi-Calculus to represent VIPLE programs and Hoare Logic to formalize program requirements. By building on the inference rules of Pi-Calculus and Hoare Logic, I am able to construct a theorem prover that can perform automated semantic analysis. Furthermore, building on this theorem prover enables me to develop a self-learning algorithm that can learn the conditions for a program's correctness according to a given solution program.

I dedicate this dissertation to my mother and late father who supported me throughout my journey as a Ph.D. student.

ACKNOWLEDGEMENTS

I would like to thank Dr. Yinong Chen, who has guided me on the path of research, education, and teaching since I started working with him as an undergraduate student. His guidance helped my project grow from the initial ideas I had to a realistic, feasible endeavor. I attribute a large part of the success of VIPLE and my research to his work, and advice. I am very grateful for all his hard work.

I would also like to thank my co-chair, Dr. Huan Liu, who has offered me very helpful advice concerning post-PhD career opportunities. He has helped broaden my horizons and more seriously consider my future career. He has also helped me grow as a professional, especially in terms of communication.

Additionally, I would like to thank both of my committee members, Dr. Sharon Hsiao and Dr. Dijiang Huang, who helped inspire my research, especially when I struggled with burnout.

I would also like to thank my friends and family members who listened to me when I would become stuck in the research process or achieve a goal in my work. Without all of you, this process would have been significantly more difficult.

TABLE OF CONTENTS

Page

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Dissertation Overview 1

1.2 Contributions 8

1.3 Previously Published Work 9

2 VISUAL IOT AND ROBOTICS PROGRAMMING LANGUAGE ENVIRONMENT 11

2.1 VIPLE Introduction 11

2.2 VIPLE versus Microsoft VPL 12

2.3 VIPLE Activity Definitions 14

2.4 VIPLE Robotics Interface 17

3 VISUAL IOT/ROBOTICS PROGRAMMING LANGUAGE IN PI-CALCULUS . 20

3.0 Abstract 20

3.1 Introduction 21

3.2 The Definition of VIPLE 26

3.3 Representing VIPLE in Pi-Calculus 28

3.3.1 Terminology 28

3.3.2 Definitions 29

3.4 VIPLE Model Definition 32
3.5 Applications 43
3.6 Conclusions 44
4 SEMANTIC ANALYSIS OF CONCURRENT COMPUTING IN DECENTRALIZED IOT AND ROBOTICS APPLICATIONS 47
4.0 Abstract 47
4.1 Introduction 48
4.2 The definition of VIPLE in Pi-Calculus 53
4.3 Reasoning About Program Correctness 56
4.3.1 Issue of Parallelism 56
4.3.2 Issue of Halting 57
4.4 Applying Verifier to NP-Complete Problems 59
4.5 Applying Verifier to Finite State Machines 62
4.6 Applying Verifier to Decentralized Applications 63
4.7 Conclusions 66
5 EXPLAINABLE SELF-LEARNING THEOREM PROVER 67
5.1 Theorem Prover Introduction 67
5.2 Ontology 68
5.3 Verifiable Conditions 70
5.3.1 Hoare Triplet Commands 71

CHAPTER Page

5.3.2 Hoare Triplet Conditions 72

5.3.3 Hoare Triplet Example 74

5.4 Knowledge Reasoning 76

5.4.1 Pi-Calculus Inference Rules 76

5.4.2 Hoare Logic Evaluation in Pi-Calculus 79

5.4.3 Knowledge Reasoning Example 82

5.5 Self Learning Hoare Triplets 84

6 CONCLUSIONS AND FUTURE RESEARCH 93

REFERENCES 96

APPENDIX

A PERMISSION TO USE CO-AUTHORED PAPERS 98

LIST OF FIGURES

Figure Page

1.1 Multithreaded Sum 5

1.2 Sequential Sum 5

1.3 Directly Printing Output 5

2.1 Basic Activities in VIPLE and Microsoft VPL 12

2.2 Services in VIPLE 13

2.3 Microsoft VPL Generic, Vendor, and Simulated Services 14

2.4 Merge Activity 16

2.5 Unfolded Merge Activity 16

2.6 VIPLE Counter Activity Block 16

2.7 Join Example 17

2.8 VIPLE Robot/IoT JSON Interface 18

2.9 VIPLE Robot Communication 19

2.10 Robot Connection Interface 19

3.1 Activities: ASU VIPLE 26

3.2 ASU VIPLE Services Categorized by Functionality 27

3.3 A Join Activity 34

3.4 A Merge Activity with Multiple Input Connections 35

3.5 An If Activity 36

3.6(a) A Loop Using the While Activity 39

3.6(b) A Loop Using the If and Merge Activities 39

3.7 A Robot Distance Sensor Activity 42

Figure Page

3.8 Finite State Machine for a Garage Door Opener 43

3.9(a) VIPLE Diagram Implementing a Garage Door Opener (Remote Button Press).... 45

3.9(b) VIPLE Diagram Implementing a Garage Door Opener (Limit Sensor Touched) . 46

4.1 Platforms Supported in VIPLE 51

4.2 Convert VIPLE Program into Pi-Calculus for Verification....................................... 53

4.3 Basic Activities in VIPLE... 54

4.4 Simple Join in Pi-Calculus Example.. 55

4.5 Calculate and Variable in Pi-Calculus 56

4.6 VIPLE Program for Finding Hamiltonian Path 60

4.7 Pi-Calculus Representation of the Hamiltonian Path Finding VIPLE Code 61

4.8 Vending Machine Example.. 63

4.9 General Purpose and Robot/IoT Services 65

4.10 Two Robot/IoT Controllers Used in VIPLE.. 65

5.2 Sum Example 69

5.3 Trivial Assignment.. 75

5.4 Merge Loop Counter.. 85

5.5 Merge Loop Counter Pi-Calculus 86

5.6 Merge Loop Counter Learned Hoare Triplets 86

5.7 While Loop Counter 87

5.8 While Loop Counter Pi-Calculus... 87

5.9 Dog Finite State Machine 88

5.10 Dog Finite State Machine Pi-Calculus.. 88

Figure Page

5.11 Dog Finite State Machine Learned Hoare Triplets 89

5.12 Updated Dog Finite State Machine 89

5.13 Updated Dog Finite State Machine Pi-Calculus 90

5.14 Self-Adjusting Right Wall Following Algorithm 91

5.15 Maze Navigation Pi-Calculus 92

5.16 Maze Navigation Learned Hoare Triplets 92

CHAPTER 1

INTRODUCTION

1.1 Dissertation Overview

Artificial Intelligence (AI) has become a popular area of Computer Science (CS), both within and outside of academia. AI can reference a large variety of different topics, including Machine Learning (ML) to Knowledge Representation and Reasoning (KRR), both of which are incorporated in this dissertation research. One of the major challenges of AI is its inexplicability. Especially in more complex systems with self-learned concepts, explaining the behavior of an AI system is typically infeasible at best. Furthermore, defining the concept of explainability in AI provides its own set of issues. As discussed in the paper by Doshi-Velez and Kim, interpretability (i.e. explainability) of AI is a difficult concept to define and implement, yet has high value in certain use cases. One of these cases is "Scientific Understanding" or the pursuit of knowledge [1]. This dissertation will focus on the pursuit of knowledge for the purpose of software development, particularly, for novice developers. These developers understand the application logic, but are not trained in programming with complex and detailed syntax. These developers include professionals in non-CS disciplines and young students in K-12 classes. We have developed tool called Visual IoT and Robotics Programming Language Environment (VIPLE) that can help the novice developers to develop software without learning complex syntax first. The explainable AI is used in developing the auto-grader of code written in VIPLE.

As software development becomes an increasingly vital facet of society, it is increasingly important to teach students how computers work and how to develop their own software at younger ages. The CS for All initiative requires that CS and programming courses are offered in K-12 classes [2]. In an effort to support this goal, various introductory level computer science courses have been created that teach the fundamentals of computational thinking, rather than only the basics of programming. One such example is CSE101 at Arizona State University (ASU), recently rebranded as FSE100 [3]. This course made use of Microsoft's Robotics Developer Studio (MSRDS) and Visual Programming Language (VPL) in an effort to reduce attrition rates of introductory computer science students as well as to facilitate the learning of computational thinking without forcing the new students to focus on the syntax of a programming language. With Microsoft ending development and support for their VPL in 2014 [4], the courses that relied on this software were forced to find alternatives, ranging from standard text-based programming languages such as Java to specialized languages such as MATLAB. In response to this issue, we developed the Visual IoT and Robotics Programming Language Environment (VIPLE) as a replacement to Microsoft's VPL. While Microsoft's VPL was based on their Concurrency and Coordination Runtime and Decentralized Software Services [4], VIPLE employs modern open interface solutions, including Windows' standard multithreading libraries and REST/SOAP services. VIPLE has been used in several sections of FSE100 at ASU starting in Spring 2016 as well as in universities worldwide.

In order to help students maximize their knowledge of these studied areas, teaching assistants play a vital role in education. By grading student assignments and offering feedback on their performance, including specific issues, next steps, and strong aspects of their work, teaching assistants can offer students additional help that will allow the student to more quickly learn the material. However, the job of teaching assistants is essentially the semantic analysis of student programs combined with feedback. As such, the former job can be replaced with an AI that is able to perform semantic analysis of such programs. As discussed in the paper on interpretable ML, an explainable AI perfectly serves the purpose given here, namely the use of feedback from an AI to assist in the pursuit of knowledge [1].

In order to accomplish this feat and develop such an AI-based auto-grader, several goals have been pursued in this dissertation. The first several goals of this research focus on the advancement of VIPLE functions and the components that compose the semantic analyzer. These goals focus on the construction of a theoretical framework upon which a software application written in VIPLE can be analyzed at the semantic level (i.e. an AI which can perform semantic analysis and grading of VIPLE programs). These goals will culminate in a theoretical framework and an implementation in VIPLE demonstrating the framework.

As a semantic level analyzer, this AI is essentially a novel method of performing autograding. Autograder implementations currently in use, such as the use of a Jupyter Notebook in Vocareum, typically perform analysis of user output, potentially accepting a margin of error in the output [5]. Essentially, such an autograder is performing numerical or string comparisons on the output of the program and not analyzing the behavior of the program. Often, a series of test cases or unit tests are used to generate the output for comparison. As a workflow-based language with limited semantic scope, VIPLE is not subject to all of the restrictions which typically make automated semantic analysis difficult or impossible to perform.

In contrast to these existing autograding frameworks, the semantic analyzer to be developed in this research provides several novel features that contribute to the importance of its research. First, as the name suggests, the semantic analyzer performs semantic level analysis of the code rather than a comparison of the output of test cases. Consider the three programs in Figures 1.1, 1.2, and 1.3. Clearly the operations that these three programs perform are significantly different. The first program spawns 3 threads and partitions the work to produce the sum from 0 to 1500. The second program performs the work sequentially, on a single thread. The third program only prints out the answer. If these programs were given to an output comparison-based autograder, all three would be marked as correct, despite only the first program demonstrating the expected level of content mastery. With the semantic analyzer, these three cases, as well as many others, can be distinguished between and correctly graded.

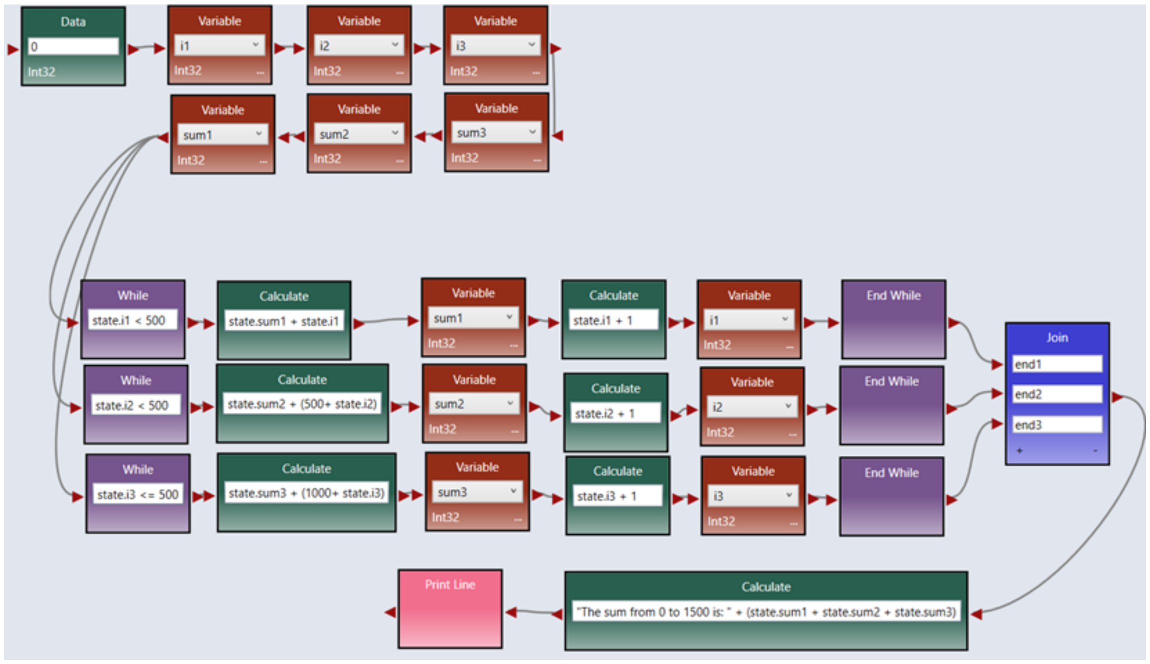


Figure 1.1 Multithreaded Sum

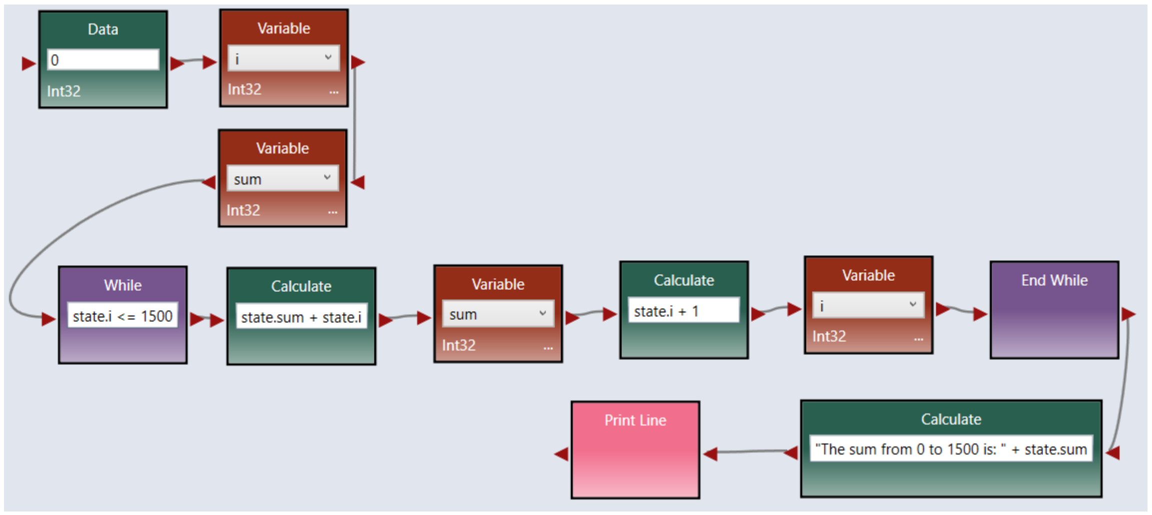


Figure 1.1 Sequential Sum

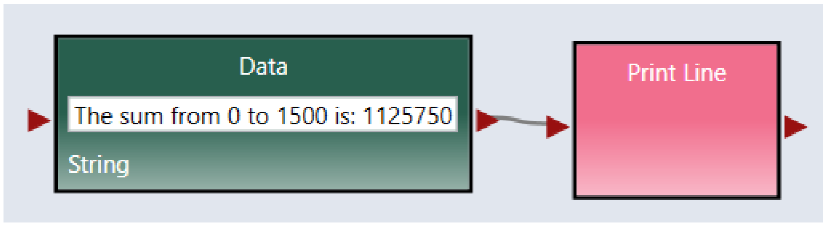


Figure 1.2 Directly Printing Output

The first goal of this research is the advancement of VIPLE in order to support the development of the AI for use in the classroom. In Chapter 2, the original development of VIPLE is discussed. As mentioned above, VIPLE was developed in an effort to enable students to learn computational thinking without focusing on the syntax of a text-based programming language. As such, the details of VIPLE's development reveal the value in employing VIPLE as the platform for this research. The current implementation of VIPLE has included some AI examples for students to learn AI, including image recognition and autonomous driving simulation.

The second goal of this research is the development of a mathematical framework for VIPLE, upon which the research can be developed. Chapter 3 details the use of Pi-Calculus to create this framework. Pi-Calculus is a process calculus that can be used to model languages or networks in terms of processes and their communication. It is based on the Calculus of Communicating Systems and is a simple yet complete foundation for this research. It can be used in two ways. The first is to represent the language using the concept of processes as a primitive, in the same way as a Turing Machine uses the tape as a primitive. The second is to model a general network of processes. This latter method is employed by treating activities in the workflow language as processes. This natural application of Pi-Calculus enables a simpler conversion and the direct application of Pi-Calculus logic to VIPLE.

The third goal of this research is the development of a logic framework upon which the semantic analysis can be performed. Chapter 4 details the two components of this goal. First, the Pi-Calculus representation is updated to fully support and model

VIPLE applications. Using this model, Hoare Logic is employed to perform reasoning about the Pi-Calculus in order to provide a semantic analysis of the VIPLE program.

The fourth goal of this research is the formalization of the semantic analyzer as an AI. Chapter 5 details the advanced aspects of the semantic analyzer, including how analysis is employed to determine correctness of programs. In addition, that chapter provides a detailed overview of how the semantic analyzer can be formally treated as an AI. This formalism is important for a couple of reasons. First, as the semantic analyzer is in actuality an AI, and this formalism allows this research to directly contribute to the area of explainable AI, especially since the ultimate goal remains the pursuit of knowledge for the user. Secondly, this formalism facilitates the development of the final component which is dependent on this AI foundation.

The final goal of this research is the application of ML to create a self-learning AI that can fully replace the teaching assistant's job of software analysis in VIPLE environment. Chapter 6 provides the full details of this AI and the learning process. In the teaching assistant use case, they may be given a solution (or create their own) and develop guidelines for determining correctness of a student program. For example, these guidelines may include correct algorithm structure, correct values at intermediary points in the program, etc. This AI is able to learn from a sample solution the foundational rules that define unique correctness of that program. Applying these rules to the aforementioned semantic analyzer AI enables the AI to become self-explainable without the manual introduction of feedback by the human user. Rather, the AI can both learn rules and determine human explainable reasons for user program mistakes.

1.2 Contributions

This dissertation contributes to the areas of Computer Science Education and Artificial Intelligence. In this section, I will briefly discuss the contributions my dissertation research has made to these disciplines. Specifics of my work will be discussed in full in the remaining chapters.

The first set of contributions my work makes is in the area of Computer Science Education. As mentioned in Section 1.1, there are a variety of existing autograders used in education for grading and offering feedback on student programming assignments. These autograders are limited in their capabilities as they can only perform output comparison. This research, however, focuses on semantic analysis. Essentially, this research provides a methodology to analyze the behavior of student code in a sort of white box analysis, as opposed to the typical black box analysis of standard autograders. Furthermore, this research is performed on the visual programming language VIPLE, as discussed above. The development of VIPLE is also a part of the contributions to the education discipline. Note that VIPLE handles syntactic errors in the same way as standard programming language compilers do, namely when the code is initially compiled or interpreted. This work is focused on the semantic errors of the code.

The Artificial Intelligence research is focused mainly on the area of Knowledge Representation and Reasoning. The first part of this work is in the representation of programming languages and rules for programming language correctness. The former is done by employing and modifying the Pi-Calculus to represent VIPLE programs. The rules are represented by employing Hoare Logic. Knowledge Reasoning is performed by combining the Pi-Calculus and Hoare Logic representations in conjunction with the Pi-

Calculus and Hoare Logic inference and reasoning rules. By combining these representations and rules together, a program can be analyzed for semantic correctness according to a desired set of rules.

In addition to the aforementioned aspect of Artificial Intelligence, one of the final contributions of this work is to the area of Machine Learning. One of the original goals of this research is to develop a system that is usable by instructors regardless of their background knowledge. To this end, requiring an understanding of the Hoare Logic representation would serve as a barrier to entry to this work. As such, inductive machine learning is employed to learn a set of rules according to a set of concepts from a given answer key program. In addition, explainability is introduced into the system to serve two purposes. First, these explanations provide the instructors reasoning about why the given rules were generated. Second, these explanations provide specific feedback to the students if they make an error related to that rule.

1.3 Previously Published Work

The contents of Chapter 3 are the work on Pi-Calculus in VIPLE, "Visual IoT/Robotics Programming Language in Pi-Calculus" [6]. These contents detail the application of Pi-Calculus to formally model VIPLE, which is a vital component in the logical analysis of VIPLE. Chapter 4 contains the work on the initial semantic analyzer, "Semantic Analysis of Concurrent Computing in Decentralized IoT and Robotics Applications" [7]. The contents of this chapter build on the previous chapter in order to provide full detail on the semantic analysis framework and the use of Pi-Calculus to meet the prerequisites of the framework.

CHAPTER 2

VISUAL IOT AND ROBOTICS PROGRAMMING LANGUAGE ENVIRONMENT

2.1 VIPLE Introduction

In 2006, Microsoft released a software suite that had the potential to improve computer science education. This software, the Microsoft Robotics Developer Studio (MSRDS) and Visual Programming Language (VPL), provided an environment to develop workflow applications and interface with various robots, such as the Lego NXT devices [4]. In addition to being a workflow language, Microsoft VPL offers support for parallel computing, a relatively innovative and forward thinking feature set for a software engineering and computer science education application of the time. This software was used to help teach computer science concepts at various high schools and universities, including at Arizona State University (ASU). On September 22, 2014, Microsoft's robotics research division was suspended due to internal restructuring [3]. In order to resolve this, various efforts were made to develop a new VPL to continue the curriculum. Although Microsoft VPL was still available, it lacks support for the robots that are currently on the market, such as Lego EV3 [3]. A solution that arose from this work is a new visual programming language, the Visual IoT and Robotics Programming Language Environment (VIPLE). Unlike Microsoft VPL, VIPLE offers support for open platform robots and Internet of Things (IoT) devices. Further details on what support VIPLE offers and how I developed certain features under the guidance of Dr. Yinong Chen will be discussed in this chapter.

2.2 VIPLE versus Microsoft VPL

When developing VIPLE, Microsoft VPL was used as inspiration. This development process served two goals. First, this enabled the full set of requirements to be met in VIPLE development. Namely, VIPLE must support a similar programming paradigm to Microsoft VPL with similar code blocks and interactions between these blocks. Within VIPLE, these code blocks are called “activities” and “services.” Secondly, this development method resulted in software that could be very easily used by anyone who has previously used Microsoft VPL. With similar interfaces and identical activity behavior, even old Microsoft VPL curriculums could be easily applied to VIPLE in most cases. The major exceptions to this similarity are the set of simulators offered and the robot connectivity options. Figure 2.2 demonstrates the similarities between VIPLE basic activities and Microsoft VPL basic activities. For most activities, there is a direct one-to-one correlation.

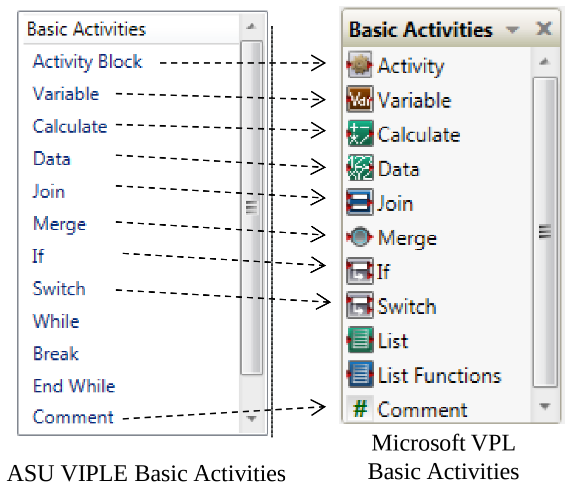


Figure 2.1 Basic Activities in VIPLE and Microsoft VPL

In addition to basic activities, Microsoft VPL offers a large number of services, most notably their vendor specific services. These services can be divided into categories such as general services, simulator services, iRobot services, and Lego NXT services. General services include simple operations such as Text to Speech, Simple Dialog, and Timer. As shown in Figure 2.2, VIPLE currently offers three sets of services, General Purpose Services, Robot/IoT Services, and Lego Services.

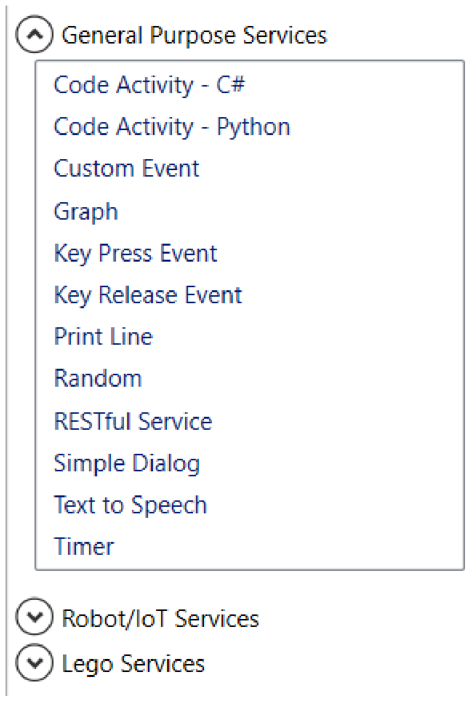
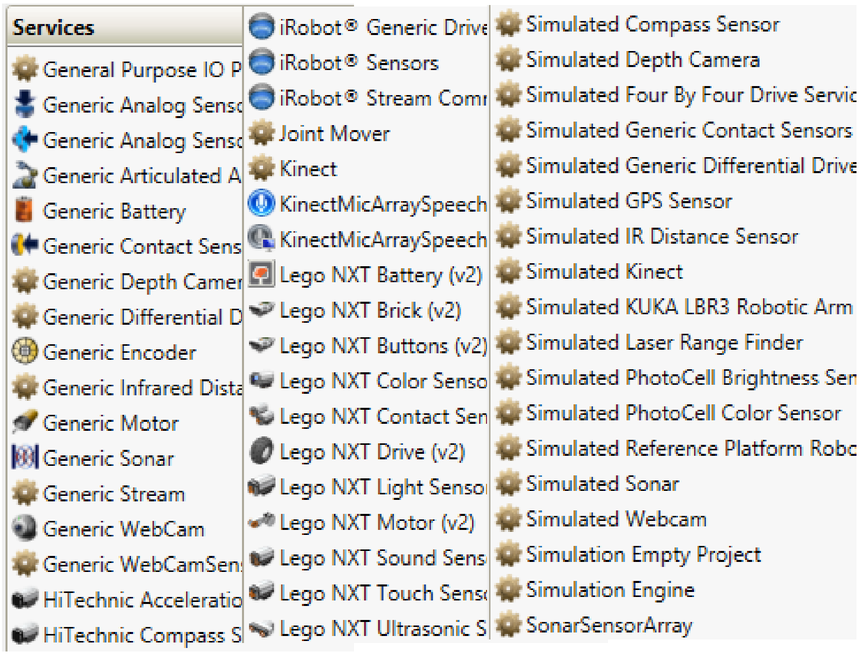


Figure 2.2 Services in VIPLE

The Robot and Lego Services contain all the services required to connect to, communicate with, and control any open hardware robot, simulator, or a Lego EV3 brick. The General Purpose Services includes many services that were originally offered in Microsoft VPL. These services include the Simple Dialog, Text to Speech, and Timer. The set of Microsoft VPL generic, vendor, and simulated services is shown in Figure 2.3 to highlight the similarities with VIPLE's services.



Services
General Purpose IO P
Generic Analog Sensc
Generic Analog Sensc
Generic Articulated A
Generic Battery
Generic Contact Sens
Generic Depth Camer
Generic Differential D
Generic Encoder
Generic Infrared Dista
Generic Motor
Generic Sonar
Generic Stream
Generic WebCam
Generic WebCamSen
HiTechnic Acceleratio
HiTechnic Compass S

iRobot® Generic Drive
iRobot® Sensors
iRobot® Stream Comr
Joint Mover
Kinect
KinectMicArraySpeech
KinectMicArraySpeech
Lego NXT Battery (v2)
Lego NXT Brick (v2)
Lego NXT Buttons (v2)
Lego NXT Color Senso
Lego NXT Contact Sen
Lego NXT Drive (v2)
Lego NXT Light Senso
Lego NXT Motor (v2)
Lego NXT Sound Sens
Lego NXT Touch Sens
Lego NXT Ultrasonic S

Simulated Compass Sensor
Simulated Depth Camera
Simulated Four By Four Drive Servic
Simulated Generic Contact Sensors
Simulated Generic Differential Drive
Simulated GPS Sensor
Simulated IR Distance Sensor
Simulated Kinect
Simulated KUKA LBR3 Robotic Arm
Simulated Laser Range Finder
Simulated PhotoCell Brightness Ser
Simulated PhotoCell Color Sensor
Simulated Reference Platform Robc
Simulated Sonar
Simulated Webcam
Simulation Empty Project
Simulation Engine
SonarSensorArray

Figure 2.3 Microsoft VPL Generic, Vendor, and Simulated Services

2.3 VIPLE Activity Definitions

As shown above in Figure 2.1, VIPLE has a set of basic activities that define standard computation behavior. VIPLE is a multithreaded workflow language. Behavior between threads is handled internally, eliminating the requirement for users to employ synchronization mechanisms such as locks or semaphores. This section will cover the behavior of each of the basic activities, in order to facilitate the explanation of later examples.

The Activity Block is the equivalent of a method and enables definition of a reusable set of activities. Variable enables reading and writing to variables with primitive types. These variables are guaranteed to be atomic. However, race conditions may still occur if variables are written to in concurrent threads. Data provides a simple method for declaring a value of a primitive data type. Calculate, on the other hand, enables the application of a VIPLE expression. These expressions are evaluated using a custom VIPLE text compiler. The set of legal expressions is a subset of C# combined with expressions that were offered in Microsoft VPL, in an effort to support all previous Microsoft VPL expressions. The If, Switch, While, Break, and End While activities all act as standard control flow operations, as in a procedural programming language. These activities all act synchronously, and do not coordinate multiple threads.

In order to perform coordination and communication between multiple threads in VIPLE, the Merge and Join activities are offered. Merge defines an n-to-one connection. Whenever an input thread arrives at the Merge, it is immediately forwarded. The code following the Merge is essentially cloned. Figure 2.4 demonstrates an application of the Merge activity. Figure 2.5 shows the internal behavior of using a Merge activity. By employing this behavior of Merge, looping can be performed without employing the While activity. An example of a counter loop is shown in Figure 2.6. In this example, the code is implemented inside an Activity Block rather than in the main code section. In the Activity Block, there is a single input, denoted by the arrow on the left side. There are also two outputs. The arrow on the right denotes the synchronous output, which enables continuation of program flow directly out of the activity. Note that if an Activity Block exits multiple times, replication is performed in the same way as in the Merge activity example. The circle on the right demonstrates the event output. Employing this output enables the definition of custom events, for creation of custom event-driven programs.

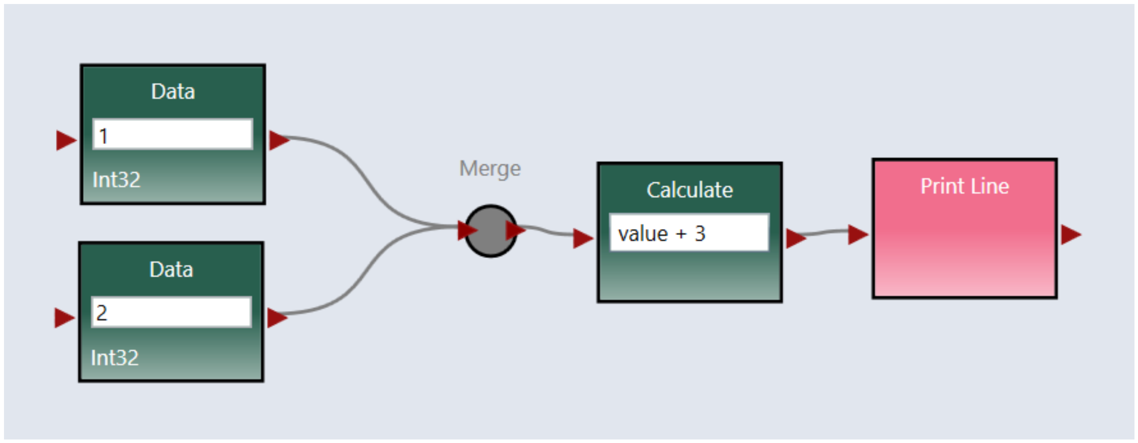


Figure 2.4 Merge Activity

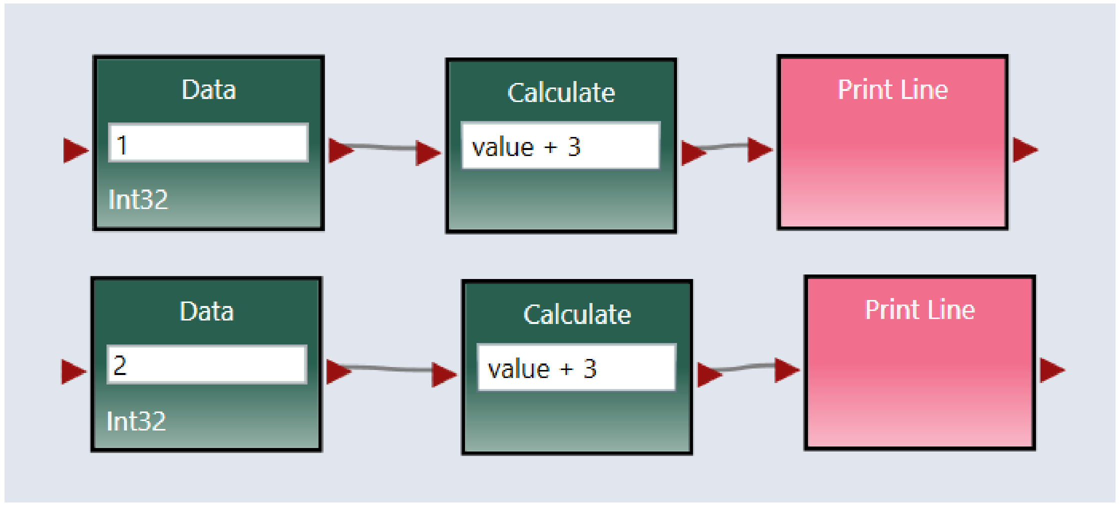


Figure 2.5 Unfolded Merge Activity

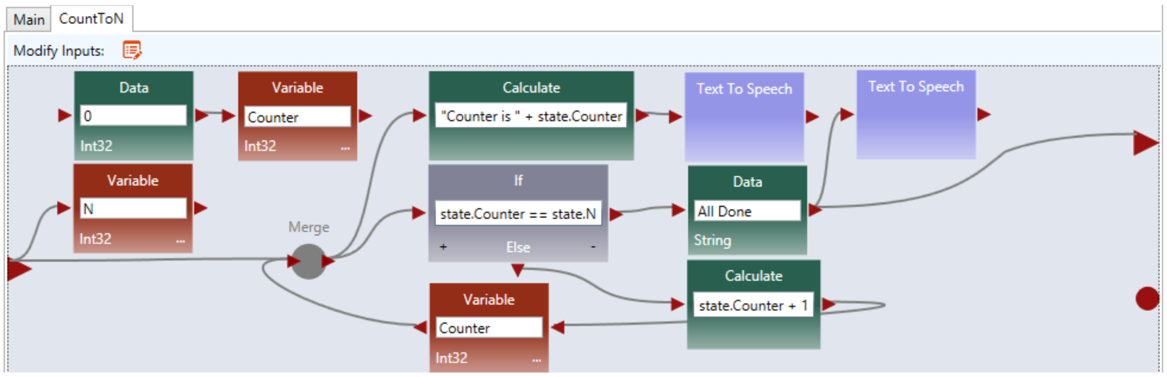


Figure 2.6 VIPLE Counter Activity Block

Join activities act as a coordination mechanism between multiple threads. An example is shown in Figure 2.7. The Join activity awaits all input threads (similar to the

Join method in C#). Once all threads have reached the join, a single thread continues execution after the Join. This thread contains the data from each input thread in the form of named data. In this example, the thread data is named “a” and “b” for use after the Join. In some cases, certain threads may reach the Join activity multiple times before all the threads are reached. Furthermore, the same thread may deliver different data to the Join each time. For these cases, a queue is employed for each thread input to the Join. Whenever every queue in the Join has at least one value in it, execution continues past the join. The data contained in the thread is a combination of the first value from each of the queues. These values are named as denoted in the Join.

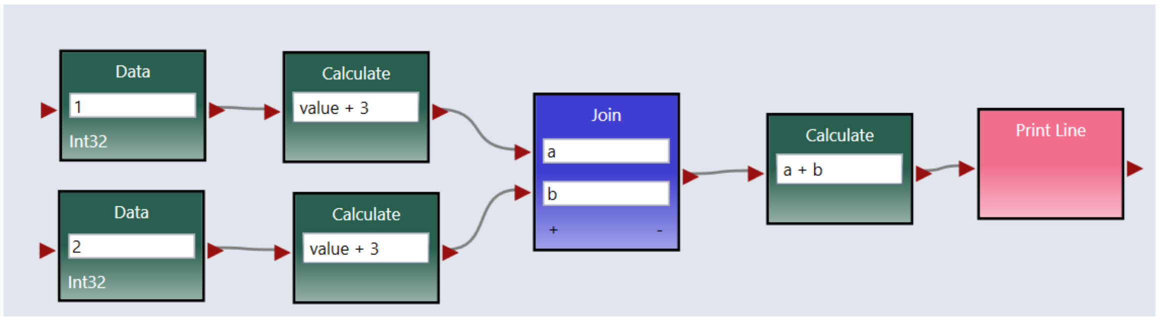
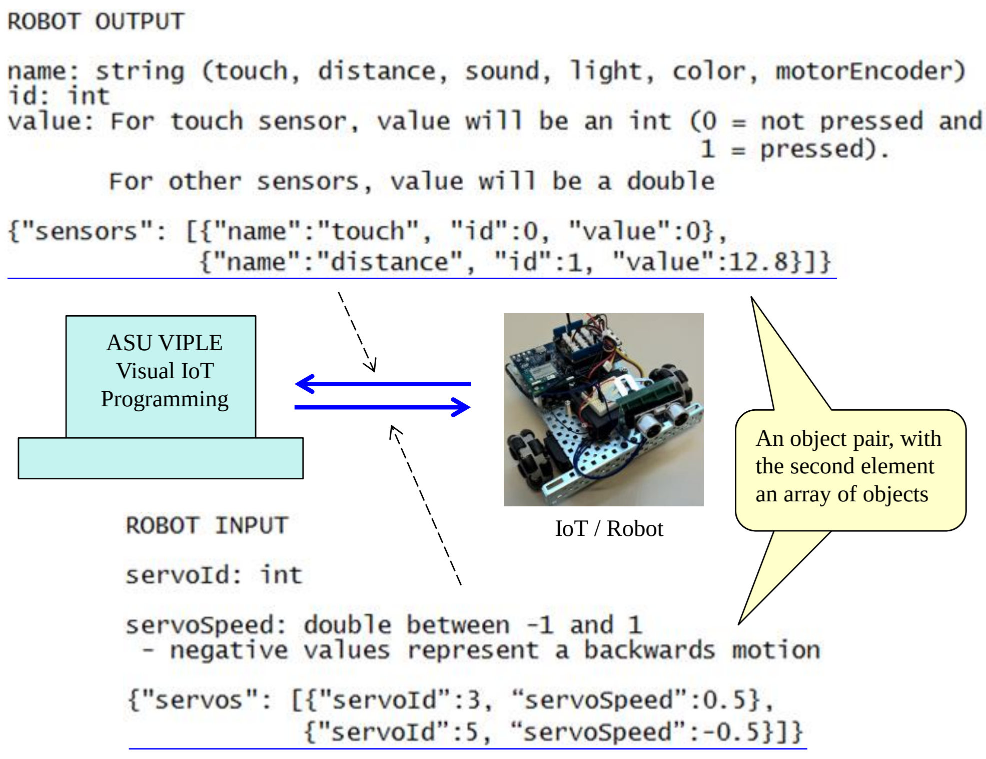


Figure 2.7 Join Example

2.4 VIPLE Robotics Interface

In order to support parallelism, Microsoft VPL employs a special set of parallelism libraries, the Concurrency and Coordination Runtime (CCR) and the Decentralized Software Services (DSS). These libraries support not only parallelism in robotics, but also parallelism in workflow programming and service-oriented applications [4]. However, these libraries lack support for standard solutions such as RESTful services or open platform robots without significant additional work. In contrast, VIPLE employs standard solutions to these problems. For parallelism, it employs standard threading libraries. For service-oriented applications, it employs support for RESTful and SOAP services. For robot and IoT applications, VIPLE supports Wi-Fi and Bluetooth connections that communicate over a standard JSON interface. That interface is shown in Figure 2.8. An interested user could interface with VIPLE using any custom-built robot assuming that these interface requirements were met.



```
ROBOT OUTPUT

name: string (touch, distance, sound, light, color, motorEncoder)
id: int
value: For touch sensor, value will be an int (0 = not pressed and
                                                1 = pressed).
       For other sensors, value will be a double

{"sensors": [{"name":"touch", "id":0, "value":0},
             {"name":"distance", "id":1, "value":12.8}]}
```

```
ROBOT INPUT

servoId: int

servoSpeed: double between -1 and 1
 - negative values represent a backwards motion

{"servos": [{"servoId":3, “servoSpeed”:0.5},
            {"servoId":5, “servoSpeed”:-0.5}]}
```

ASU VIPLE Visual IoT Programming
IoT / Robot
An object pair, with the second element an array of objects

Figure 2.8 VIPLE Robot/IoT JSON Interface

To communicate with a device over this interface, VIPLE offers a set of standardized services. As mentioned above, these are the Robot/IoT services. Figure 2.9 demonstrates an application of these services. In this example, values are being read from the robot’s touch sensor, distance sensor, and color sensor. These three results are all printed out to the output window. Unlike with vendor specific services, these services are generalized, meaning they support any robot with that standard interface. For example, a simulator could be employed to test the code. Then, the exact same code can be used with physical hardware by only changing the connection settings inside the Robot activity. Figure 2.10 shows the interface for changing the Robot activity connection settings.

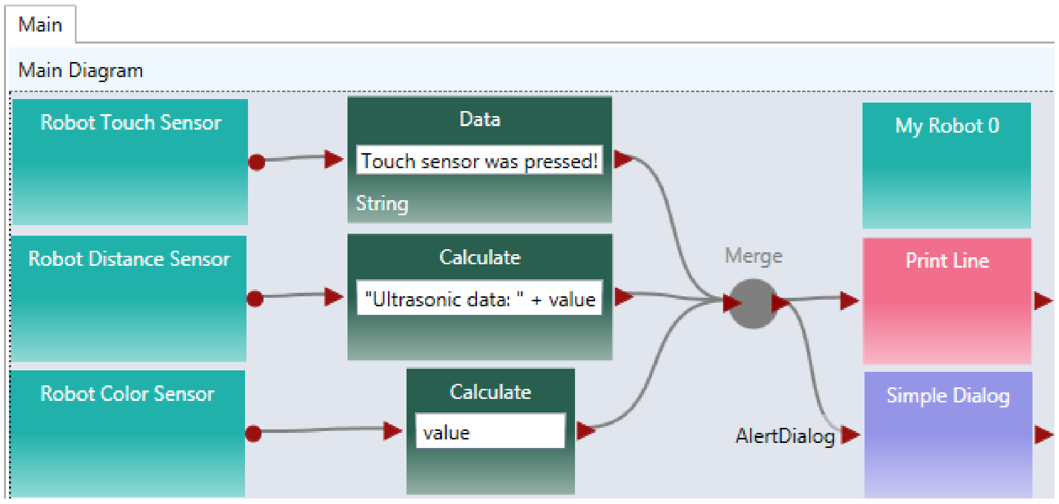
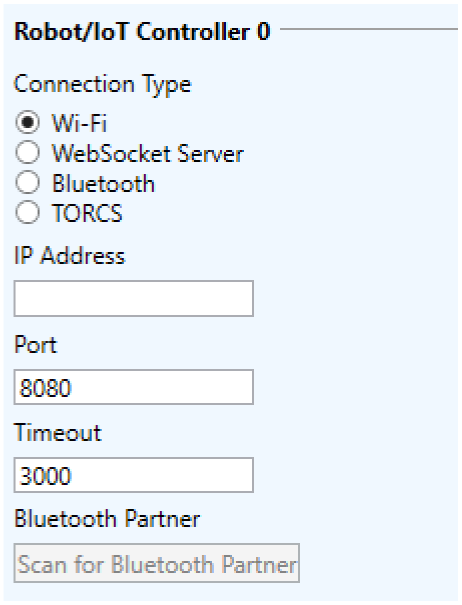


Figure 2.9 VIPLE Robot Communication


Robot/IoT Controller 0

Connection Type

[✓ Wi-Fi]
WebSocket Server
Bluetooth
TORCS

IP Address

Port: 8080

Timeout: 3000

Bluetooth Partner

Figure 2.10 Robot Connection Interface

CHAPTER 3

VISUAL IOT/ROBOTICS PROGRAMMING LANGUAGE IN PI-CALCULUS

3.0 Abstract

As IoT/robotics research and applications expand explosively into many domains in computing, information, and control systems, schools and universities must prepare students to understand and be able to program IoT devices and robots. However, programming IoT and physical devices is hard and depends on good understanding of hardware and low-level programming. Workflow and visual programming environments have been developed to address this issue, including MIT App Inventor, Microsoft Robotics Developer Studio and VPL (Visual Programming Language), and Intel Service Orchestration Layer. We have developed a new visual programming language and its development environment: ASU VIPLE (Visual IoT/Robotics Programming Language Environment). ASU VIPLE is specifically developed to support different device platforms. It 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, and event-driven programming seamlessly into the curriculum. It has been pilot tested at Arizona State University in summers 2015 and 2016, and is now taught in an ASU regular class, Introduction to Engineering. It has been adopted by numerous universities around the world. This paper focuses on the mathematical model of ASU VIPLE. π-calculus is used to describe the basic activities in VIPLE, as well as the workflow defined by the basic activities.

3.1 Introduction

The development of Internet and cloud computing has pushed the desktop-based computing platform into an internet and Web-based computing infrastructure. It has 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 the rapid development of many commercial and mission-critical applications in the new infrastructure, including the integration of physical devices, which gives Internet of Things the unlimited computing capacity.

Internet of Things (IoT) was initially proposed and applied in the Radio-Frequency Identification RFID-tags to mark the Electronic Product Code (Auto-ID Lab) [8]. IoT concept 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 [9]. Internet of Intelligent Things (IoIT) deals with intelligent devices that have adequate computing capacity. Distributed intelligence is a part of the IoIT [10]. According to Intel's report, there are 15 billion devices that are connected to the Internet, of which 4 billon devices include 32-bit processing power, and 1 billon devices are intelligent systems [11].

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 are 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. ADS and the related technologies have a large number of applications in industrial production lines, railway signaling, and robotics [14][15]. ADS concepts are the foundation of later technologies such as cloud computing and Internet of Things.

Robot as a Service (RaaS) is a cloud computing unit that facilitates the seamless integration of robot and embedded devices into Web and cloud computing environments [12][20]. In terms of 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 [13]. 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 implementations. For example, 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 architecture, such as Galileo and Edison, made it easy to program these devices as Web services. From different perspectives, an RaaS unit can be considered a unit of Internet of Things (IoT), Internet of Intelligent Things (IoIT) that have adequate computing capacity to perform complex computations [3], a Cyber-physical system (CPS) that 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).

As IoT/robotics research and applications expand explosively into many domains in computing, information, and control systems, schools and universities must prepare students to understand and to be able to program IoT devices and robots. However, programming IoT and physical devices is hard and depends on good understanding of hardware and low-level programming. Workflow and visual programming languages have been developed to address this issue.

MIT App Inventor [16] uses drag-and-drop style puzzles to construct phone applications on the Android platform. Carnegie Mellon's Alice is a 3D game and movie development environment [17] on desktop computers. 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 Service Orchestration Layer is a workflow language that allows quick development of IoT applications on Intel's IoT platforms, such as Edison and Galileo.

Microsoft Robotics Developer Studio (MSRDS) and Visual Programming Language (VPL) are specifically developed for robotics applications [18], and they are milestones in software engineering, robotics, and computer science education from many aspects. MSRDS 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, the author participated in the earlier discussion of service-oriented robotics developer environment at Microsoft. MSRDS VPL was immediately adopted at Arizona State University (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 Ira A. Fulton Schools of Engineering (FSE) at ASU in 2011 and was renamed FSE100, which is offered to thousands of freshman engineering students now [20].

Unfortunately, Microsoft stopped its development and support to MSRDS and VPL in 2014 [4], which led to our FSE100 course, and many other schools' courses using VPL, without further support. Particularly, the latest version of VPL (MSRDS R4) does not support LEGO's third generation 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 the 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 [3].

ASU VIPLE is based on our previous eRobotics development environments [19]. It is designed to support as many features and functionalities that MSRDS VPL supports as possible, in order to better serve the MSRDS VPL community in education and research. To serve this purpose, VIPLE also uses a similar user interface, so that the MSRDS VPL development community can use VIPLE with little learning curve. VIPLE does not replace MSRDS VPL. Instead, it extends MSRDS VPL in its capacity in multiple aspects. It can connect to different physical robots, including EV3 and any robots based on the open architecture processors. ASU VIPLE has been pilot tested at

ASU in summer 2015 and in spring 2016 as well as at several other universities. ASU VIPLE software and documents are free and can be downloaded at:

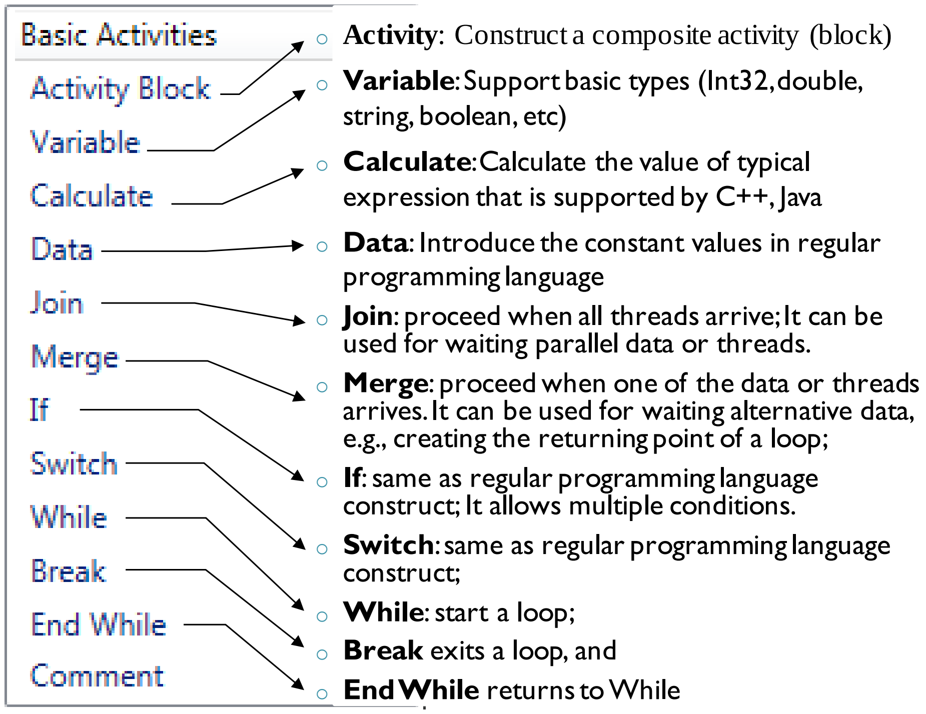
http:// neptune.fulton.ad.asu.edu/WSRepository/VIPLE/

The main goal of this paper is to provide a model to represent VIPLE using π-calculus. As VIPLE is a workflow language, this paper will expound and improve a pre-existing model for workflow languages. In their paper on workflow patterns, Puhlmann and Weske present a model for representing general workflow languages [23]. They begin with a model for basic control flow patterns and proceed to cover several advanced branching and synchronization patterns [23]. This paper will reduce the size of the model to succinctly express VIPLE's various patterns. Of additional note is their use of τ throughout their model to represent actions performed by each activity that cannot be represented by π-calculus [23]d. As VIPLE is a well-defined (i.e. the behavior is well-defined) workflow language, these actions can be simplified, and in some cases removed, to more specifically detail the behavior of each activity.

The rest of the paper is organized as follows. Section II presents ASU VIPLE, the Visual IoT/Robotics Programming Language Environment, which makes IoT/Robotics device programming as easy as drawing and connecting functional boxes. Section III provides an introduction to VIPLE and π-calculus terminology and definitions to facilitate the creation of the model. Section IV presents the model for VIPLE in π-calculus. Section V provides examples of applications facilitated by using VIPLE. Section VI concludes the paper.

3.2 The Definition of VIPLE

The basic building blocks of an ASU VIPLE program (diagram) is listed and explained in Figure 3.1.



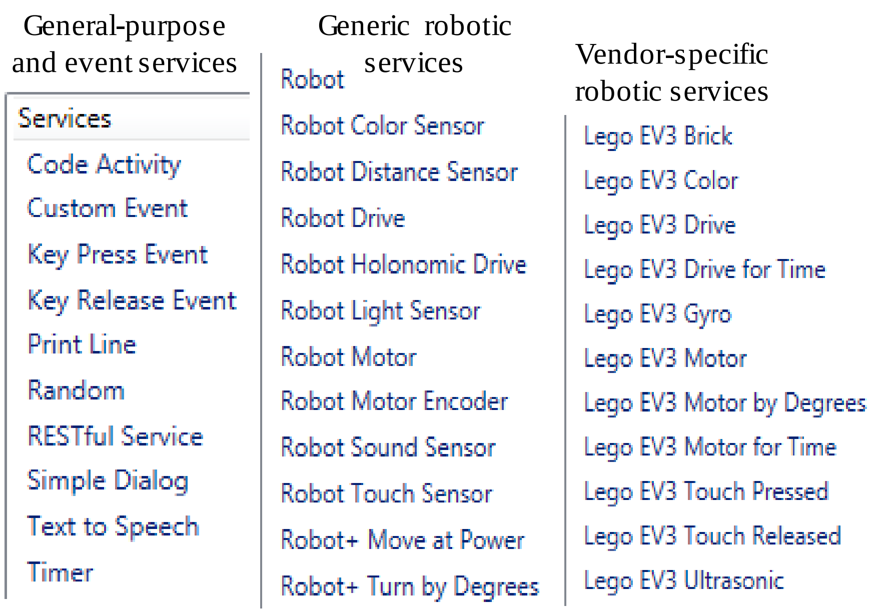
- **Activity**: Construct a composite activity (block)
- **Variable**: Support basic types (Int32, double, string, boolean, etc)
- **Calculate**: Calculate the value of typical expression that is supported by C++, Java
- **Data**: Introduce the constant values in regular programming language
- **Join**: proceed when all threads arrive; It can be used for waiting parallel data or threads.
- **Merge**: proceed when one of the data or threads arrives. It can be used for waiting alternative data, e.g., creating the returning point of a loop;
- **If**: same as regular programming language construct; It allows multiple conditions.
- **Switch**: same as regular programming language construct;
- **While**: start a loop;
- **Break** exits a loop, and
- **End While** returns to While

Figure 3.1 Activities: ASU VIPLE

The usability of a language largely depends on the availability of library functions or called services. Figure 3.2 shows the three sets of ASU VIPLE services.

The first set contains the 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, 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 3.2, 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.



General-purpose and event services	Generic robotic services	Vendor-specific robotic services
Services	Robot	Lego EV3 Brick
Code Activity	Robot Color Sensor	Lego EV3 Color
Custom Event	Robot Distance Sensor	Lego EV3 Drive
Key Press Event	Robot Drive	Lego EV3 Drive for Time
Key Release Event	Robot Holonomic Drive	Lego EV3 Gyro
Print Line	Robot Light Sensor	Lego EV3 Motor
Random	Robot Motor	Lego EV3 Motor by Degrees
RESTful Service	Robot Motor Encoder	Lego EV3 Motor for Time
Simple Dialog	Robot Sound Sensor	Lego EV3 Touch Pressed
Text to Speech	Robot Touch Sensor	Lego EV3 Touch Released
Timer	Robot+ Move at Power	Lego EV3 Ultrasonic
	Robot+ Turn by Degrees	

Figure 3.2 ASU VIPLE Services Categorized by Functionality

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 column in Figure 3.2. 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.

3.3 Representing VIPLE in Pi-Calculus

In order to define a mathematical model to represent the VIPLE workflow language, several standards for how VIPLE functions must be defined.

3.3.1 Terminology

First, we introduce the terminology for defining the needed standards.

In VIPLE, an activity represents a single unit of computation. An activity may perform a task such as input, output, calculate, save value, or load value. However, an activity may also act as a control flow, either by analyzing the input, or simply performing some sort of synchronization or merging. Activities can be connected by drawing lines from the output of one activity to the input of another. These lines will hereafter be called connections.

When discussing a single activity, the input pin is the triangle pin on the left-hand side of the activity, the output pin is the triangle pin on the right-hand side, and the event pin is the circle pin on the right-hand side. Therefore, an input connection is a connection drawn to the input pin of the activity, and an output connection is a connection drawn from the output or event pin of the activity.

The model defined below will consider two different groups of activities. The first group will be called the standard activities and the second group will be the nonstandard activities. Standard activities permit either 0 or 1 input connections and n output connections ($n \geq 0$). Standard activities have 1 input pin, 1 output pin, and 0 event pins. Nonstandard activities are those activities that do not meet these standard activity criteria. As such, nonstandard activities may have a variable number of input, output, and event pins as well as support for a variable number of input or output connections. Nonstandard activities will be separately defined in their own sections in the model.

3.3.2 Definitions

As defined by Milner [22], the π-calculus "can play two distinct roles. First, it models networking…in which messages are sent from site to site. …Second, the π-calculus can be seen as a basic model of computation. The π-calculus rests upon the primitive notion of interaction" [22]. As such, the model presented for VIPLE will be based on these roles. First, the model will describe the networking between activities (i.e.

how messages are synchronized and passed between various activities). Second, the model will only describe the interaction between activities. The model will not include a description of what each activity performs internally. However, as mentioned earlier, the use of τ allows a representation of a black box action that fits within the definitions of π-calculus.

To improve the clarity of the model, below is a brief introduction to the π-calculus, based on Milner's definition.

Message passing will be standardized in this model using the concept of action prefixes. Action prefixes represent the sending or receiving of some message and are defined as follows [22]:

$x(\mathrm{y})$	receive y along x
$\bar{x}\langle y\rangle$	send y along x.

The formal definition of π-calculus is below.

Definition 9.1 The π-calculus [22]: The set Pπ of π-calculus process expressions is defined by the following syntax:

$$P ::= \sum_{i \in I} \pi_i . P_i \mid P1|P2 \mid new\ a\ P \mid !P$$

where I is any finite indexing set. The processes $\sum_{i\in I} \pi_i . P_i$ are called summations or sums.

Additional important operators are |, which executes processes in parallel, new a P which defines a private channel a belonging to process P, and !P, the replication operator, which allows the explicit definition of parametric processes.

There are two general versions of the π-calculus, namely the monadic and polyadic versions. In the monadic version, a message consists of exactly one name. In the polyadic version, multiple names can be sent in a message along a channel with no interference from other processes [22].

Below are two definitions used in proofs within the model, for later reference.

Definition 9.7 Structural congruence [22]: Two process expressions P and Q in the π-calculus are structurally congruent, written $P \equiv Q$, if we can transform one into the other by using the following equations (in either direction):

(1) Change of bound names (alpha-conversion)

(2) Reordering of terms in a summation

(3) $P|0 \equiv P,\ P|Q \equiv Q|P,\ \ P|(Q|R) \equiv (P|Q)|R$

(4) $new\ x\ (P|Q) \equiv P|new\ x\ Q\ if\ x \notin fn(P),\ new\ x\ 0 \equiv 0,\ new\ xy\ P \equiv new\ yx\ P$

(5) $!P \equiv P|!P$

Definition 4.5 Process congruence [22]: Let ≅ be an equivalence relation over P, i.e. it is reflexive ($P \cong P$), symmetric (if $P \cong Q$ then $Q \cong P$) and transitive (if $P \cong Q$ and $Q \cong R$ then $P \cong R$). Then ≅ is said to be a process congruence if it is preserved by all elementary contexts; that is, if $P \cong Q$ then

$\alpha.P + M \cong \alpha.Q + M$

$new\ a\ P \cong new\ a\ Q$

$P|R \cong Q|R$

$R|P \cong R|Q$

3.4 VIPLE Model Definition

This section defines the behavior of workflow activities that are applicable in VIPLE diagrams.

Sequential Process

$S ::= new\ a(A|B).\mathbf{0}$

$A ::= \ \tau_A.\bar{a}\langle x\rangle.\mathbf{0}$

$B ::= a(x).\tau_B.\bar{b}\langle y\rangle.\mathbf{0}$

This definition describes the behavior of two standard activities where A is the first activity and B is the second. A performs its action, then outputs a value onto the private channel a. B receives that value from the channel a, performs its own action, and then outputs its own value on any number of channels. Note that the channel b is used to represent a non-negative number of channels. B will output the same value y along all of these channels.

In the definition of S, the keyword new is used to show that the channel a is private within the system. The value sent along the channel will be sent to its destination and nowhere else (i.e. no other activities can listen to the channel other than the ones defined in S). The channel b is not designated as new since such a definition would be redundant. For example, by reapplying the sequential process definition, b would be designated as new in the second iteration. Also note the use of **0** (nil) at the end of each process, symbolizing that the process dies after performing its task. In the remainder of the model definition, the **0** will be removed. However, all processes implicitly end with **0**.

Split (Parallel Processes)

$n \geq 2$

$S ::= new\ a1a2 \ldots an(A|B1|B2| \ldots |Bn)$

$A ::= \tau_{\mathrm{A}}.\overline{a1}\langle x\rangle.\overline{a2}\langle x\rangle. \ldots .\overline{an}\langle x\rangle$

$Bn ::= an(x).\tau_{\mathrm{Bn}}.\overline{bn}\langle y\rangle$

This part of the model covers the case where one activity spawns more than 1 activity (in parallel). In such a case, A’s output value is passed along all n channels to the receiving activities (denoted by B1, B2, …, Bn). Note that although passing values along the channels occurs sequentially, the receiving processes are not waited on. Thus, the receiving processes are able to run concurrently. Since passing data along a channel in VIPLE takes a negligible amount of time (compared to an activity’s run time), the receiving processes essentially receive their input and run at the same time.

As in the sequential process model, note that all activities in this system are standard. In addition, note that multiple different model definitions can be combined. For example, a sequential process can be added immediately following B1 while another parallel split can be added immediately following B2. In this way, various instances of the model may run in parallel. In the example, the sequential processing will occur concurrently with the parallel split. If another sequential process is added after B3, that process will be performed in parallel with B1’s sequential process and B2’s parallel split.

Join

$n \geq 1$

$S ::= new\ a1a2 \ldots an(A1|A2| \ldots |An|B)$

$An ::= \tau_{\mathrm{An}}.\overline{an}\langle x\rangle$

$B ::= a1(x1).a2(x2). \ldots .an(xn).\overline{b}\langle y\rangle$

Join is the first example of a nonstandard activity in this model. A join activity, as shown in Figure 3.3, has n input pins, n ≥ 1. Each pin still only accepts a single input connection. The job of a join, as formalized in the model, is to synchronize multiple incoming connections. A join will not allow values to be passed further (thereby preventing the execution of the proceeding activities) until values are received from all incoming connections (i.e. until all preceding activities complete their tasks). Also note that although the preceding activities perform computation (τ_{An}), there is no computation inherent to the join activity. The join's only job is to perform synchronization, which is fully represented in the model for B.

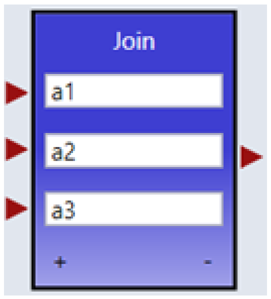


Figure 3.3 A Join Activity

Also of note in this model is that a single value (y) is output from the join once the synchronization has completed, even though n distinct values are being output (x1, x2, …, xn). A join may also be modelled using the polyadic version of π-calculus to support n outputs. However, in its implementation in VIPLE, the join activity outputs a single object containing all the values. To remain true to this implementation, the monadic version of π-calculus is used in defining join.

Merge

$n \geq 1$

$S ::= (A1|A2|\ldots|An|B)$

$An ::= \tau_{\mathrm{An}}.\bar{a}\langle x\rangle$

$B ::= !\,a(x).\bar{b}\langle y\rangle$

Merge is another nonstandard activity. A merge activity, as shown in Figure 3.4, has a single input pin and a single output pin. However, unlike standard activities, the merge activity will accept n input connections (n ≥ 0). The merge activity automatically forwards any incoming values to the output connection(s). The merge activity performs no calculations, and does not perform any synchronization, unlike the join activity. The main purposes of the merge activity are to provide the abilities to create loops and reuse code.

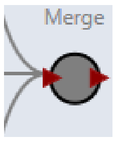


Figure 3.4 A Merge Activity with Multiple Input Connections

Of note in the above model is the use of the replication operator and the lack of a private input channel. As a merge activity supports multiple input connections to a single input pin, the input channel can be used by any activity. To preserve thread safety, the replication operator is used. In this way, any activities following the merge activity (i.e. output connections) will be replicated (a new instance will be created) for each merge input connection. This replication can be likened to the replication of a method on a stack when the method is called concurrently from multiple different threads.

Theorem 1 A merge with a single input is congruent to **0** (the nil process).

Proof

Consider a merge with a single input and n outputs (for $n \in \mathbb{N}\ where\ \mathbb{N} = \{0, 1, 2, \dots\}$). For all values of n, the merge activity receives only 1 input, thereby removing the need for replication by the replication operator. In addition, merge performs no operations, so the output value is equal to the input value.

When n = 0, a merge activity is clearly equivalent to **0** as it has no output connections, so the process ends immediately upon reaching the merge. In the cases where n > 0, the value from the input connection is passed directly to all output connections. As such, connecting the output of the previous activity directly to the inputs of the following activities will result in a congruent process.

Choice (Generalized)

$n \geq 2$

$S ::= new\ a1a2 \ldots an(A|B1|B2| \ldots |Bn)$

$A ::= \overline{a1}\langle x\rangle + \overline{a2}\langle x\rangle + \cdots + \overline{an}\langle x\rangle$

$Bn ::= an(x).\tau_{\mathrm{Bn}}.\overline{bn}\langle y\rangle$

The choice activities are nonstandard activities since they may have n output pins (n ≥ 1). Other than this difference, the choice activities follow the same rules as the standard activities (1 input pin, 1 input connection, m output connections per output pin (m ≥ 0)). An example of one of the choice activities, the if activity, is shown in Figure 3.5.

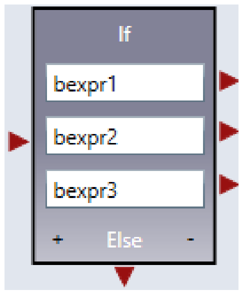


Figure 3.5 An If Activity

Of immediate notice in this model is that the choice activities share the same process diagram as the split operation. The main difference between the choice activities and the split operation is that only a single output (e.g. a2) is followed.

In this part of the model, the goal is to represent choice activities in general. To meet this goal, a summation is used. The model can be further developed to concisely define a single choice activity.

Choice (If)

The if activity (Figure 3.5) will follow the output pin corresponding to the first boolean expression (bexprn) which evaluates to true. If none of the n-1 conditions evaluate to true, the output pin corresponding to else will be followed. This behavior can be modelled using the if/then/else construct. The syntax for this construct is:

if bexp **then** P **else** Q

where P and Q are processes and bexp is a boolean expression [21].

Following this construction, the model for an if activity becomes:

$n \geq 2$

$S ::= new\ a1a2 \ldots an(A|B1|B2| \ldots |Bn)$

$A ::= \boldsymbol{if}\ bexpr1\ \boldsymbol{then}\ \overline{a1}\langle x\rangle\ \boldsymbol{else}\ \boldsymbol{if}\ bexpr2\ \boldsymbol{then}\ \overline{a2}\langle x\rangle$

$\boldsymbol{else}\ \boldsymbol{if} \ldots \boldsymbol{else}\ \boldsymbol{if}\ bexpr(n-1)\boldsymbol{then}\ \overline{a(n-1)}\langle x\rangle$

$\boldsymbol{else}\ \overline{an}\langle x\rangle$

$Bn ::= an(x).\tau_{\mathrm{Bn}}.\overline{bn}\langle y\rangle$

where bexprn represents the nth boolean expression in the if activity.

Choice (Switch)

The switch activity can be modelled in an identical fashion to the if activity with the exception that the cases of a switch activity can be any expression, and those values are compared to the input value to create a boolean expression. An important difference between if and switch is that switch requires an input value to compare with its expressions.

Supposing that the switch activity (process A) has an input channel c from process C, the switch activity can be modelled as follows:

$n \geq 2$

$S ::= new\ ca1a2 \ldots an(C|A|B1|B2| \ldots |Bn)$

$C ::= \bar{c}\langle z\rangle$

$A ::= c(z).\boldsymbol{if}\ expr1 = z\ \boldsymbol{then}\ \overline{a1}\langle x\rangle\ \boldsymbol{else\ if}\ expr2 = z$

$\boldsymbol{then}\ \overline{a2}\langle x\rangle\ \boldsymbol{else\ if} \ldots \boldsymbol{else\ if}\ expr(n-1) = z$

$\boldsymbol{then}\ \overline{a(n-1)}\langle x\rangle\ \boldsymbol{else}\ \overline{an}\langle x\rangle$

$$Bn ::= an(x).\tau_{\mathrm{Bn}}.\overline{bn}\langle y\rangle$$

where exprn represents the nth expression in the switch activity.

Loop

As mentioned above, the merge activity can be used to design loops within VIPLE. However, VIPLE also provides simplified methods of defining loops, such as the while/end while activities. The goal of this section is to prove that the model defining loops built from merge can be transformed into the model defining while loops, which would prove that they are structurally congruent.

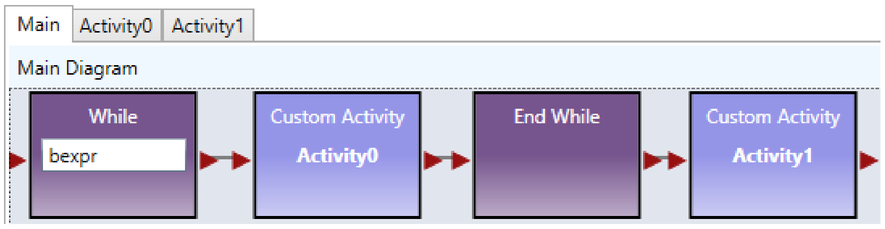


Figure 3.6(a) A Loop Using the While Activity

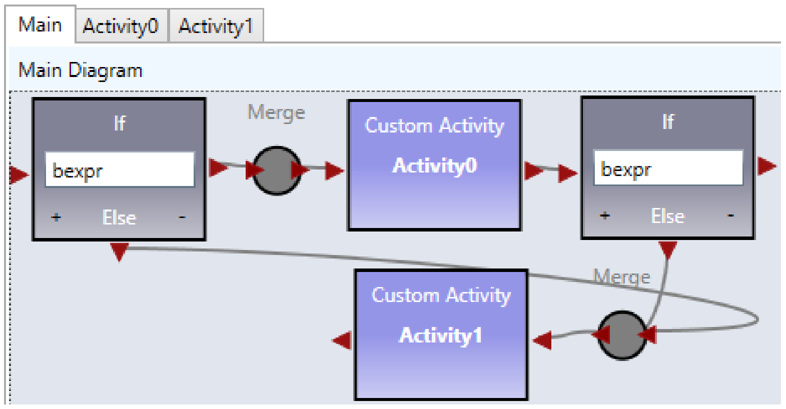


Figure 3.6(b) A Loop Using the If and Merge Activities

The model for a while loop such as the one shown in Figure 3.6(a) is:

$S ::= A|B|C$

$A ::= d(y).\boldsymbol{if}\ bexpr\ \boldsymbol{then}\ \bar{a}\langle x\rangle\ \boldsymbol{else}\ \bar{c}\langle y\rangle$

$B ::= a(x).\tau_B.\bar{b}\langle y\rangle$

$C ::= b(y).\bar{d}\langle y\rangle$

where channel c is the output channel(s) from C to the activity following the while loop and bexpr is the boolean expression in the while activity. Note that this model assumes some initial input to A, which will be **0** (nil) if the while activity has no input connections.

The model for an if/merge loop such as the one shown in Figure 3.6(b) is:

$S ::= A|B|C|D$

$A ::= d(x).\boldsymbol{if}\ bexpr\ \boldsymbol{then}\ \bar{a}\langle x\rangle\ \boldsymbol{else}\ \bar{e}\langle y\rangle$

$B ::= a(x).\tau_B.\bar{b}\langle y\rangle$

$C ::= b(y).\bar{c}\langle y\rangle$

$D ::= c(y).\boldsymbol{if}\ bexpr\ \boldsymbol{then}\ \bar{a}\langle x\rangle\ \boldsymbol{else}\ \bar{e}\langle y\rangle$

where channel e is the output channel(s) from D (and A) to the activity following the while loop and bexpr is the boolean expression in the if activity (note that the boolean expression is assumed to be the same in both if activities). As above, process A is assumed to receive some input, which will be **0** (nil) if A has no input connections.

An important note about this model is that identical output channels are used from both A and D. Since standard activities do not support multiple inputs, a merge activity can be placed before the output channels. Since that merge will only receive a single input (only the first time bexpr is false will the merge be used), that merge is equivalent to **0** according to Theorem 1. As such, inserting the merge will create a structurally congruent process.

As the if activities (processes A and D) do not modify their input values, have the same boolean expressions, and output to the same channels in the same cases, processes A and D are congruent processes. Thus, instead of inputting a value to process A along channel d, channel c will be used to pass the value directly to process D. Since the processes are congruent, this new model is structurally congruent to the previous if/merge model. As process A no longer receives input along channel d, it can no longer perform any operations and is removed from the model. The model then becomes:

$S ::= B|C|D$

$B ::= a(x).\tau_B.\bar{b}\langle y\rangle$

$C ::= b(y).\bar{c}\langle y\rangle$

$D ::= c(y).\boldsymbol{if}\ bexpr\ \boldsymbol{then}\ \bar{a}\langle x\rangle\ \boldsymbol{else}\ \bar{e}\langle y\rangle$

With alpha-conversions (rename c to e, d to c, e to d, and D to A, the model becomes:

$S ::= A|B|C$

$A ::= d(y).\boldsymbol{if}\ bexpr\ \boldsymbol{then}\ \bar{a}\langle x\rangle\ \boldsymbol{else}\ \bar{c}\langle y\rangle$

$B ::= a(x).\tau_B.\bar{b}\langle y\rangle$

$C ::= b(y).\bar{d}\langle y\rangle$

Therefore, according to the definition of structural congruence, the if/merge loop model is structurally congruent to the while loop model. Therefore, the while activity construction can be reduced to using merge and if activities, and does not require a formal definition within the model.

Event Handling

$n \geq 1$

$S ::= A|B1|B2|\ldots|Bn|C$

$A ::= c(x).\overline{a1}\langle x\rangle.\overline{a2}\langle x\rangle.\ldots.\overline{an}\langle x\rangle$

$Bn ::= !\,an(x).\tau_B.\bar{b}\langle y\rangle$

To explain the general model offered, a specific example of event handling will be constructed. One example of an event in VIPLE is the robot distance sensor activity (a nonstandard activity with 0 input ports, 0 output ports, 1 event port, and n output connections for n ≥ 1). A distance sensor activity is shown in Figure 3.7. Without any distance input on a specified port, that event will never trigger, and the proceeding code will never execute. As such, an additional component is required for all event handling activities, namely the event generator. In this example, there may be an IoT board with an attached ultrasonic sensor which sends data to VIPLE every 200ms on port c. This board can be modelled as such:

$n \geq 1$

$C ::= C1|C2| \ldots |Cn$

$C1 ::= \tau_{sleep}.receive\_distance(d).\bar{c}\langle d \rangle$

In this implementation, the IoT board has n concurrently running sensors, where C1 is the process controlling the distance sensor and communicating on port c. The process C1 waits until the polling time has passed (e.g. it calls sleep for the specified amount of time) and then it receives the distance and sends it to VIPLE over port c. Returning to the above model for general event handling, VIPLE receives the distance value and passes that value to all output connections. As an event can be triggered multiple times, the replication operator is used so that each distance value is forwarded to its own replicated set of processes.

Figure 3.7 A Robot Distance Sensor Activity

This model can be extended to all event activities in VIPLE. Each event activity awaits an event to occur from an external source. That event may occur at a timed interval as above, or when an event occurs in the physical world (e.g. a button is pressed).

3.5 Applications

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 workflow diagram, and the components are defined as activities or Web services in the diagram.

However, VIPLE's strength is in event-driven programming that 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 3.8 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 controls the door open, stop, and close functions. The limit sensor is a built-in sensor in the motor. When the door reaches the limit, the sensor will generate a notification and stop the motor.

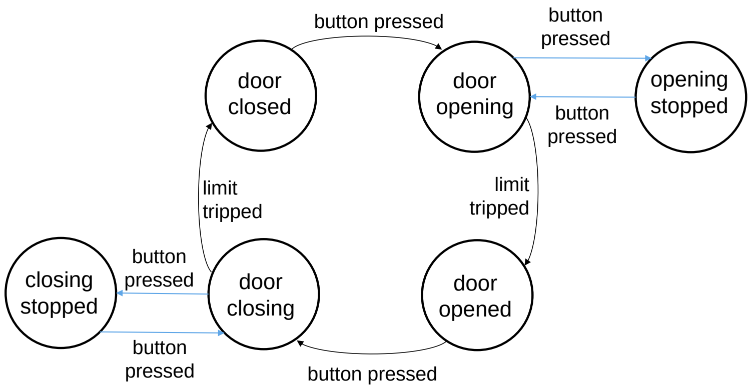


Figure 3.8 Finite State Machine for a Garage Door Opener

The VIPLE program that implements the finite state machine is given in Figure 3.9. The remote controller button is simulated by the keyboard "Ctrl" key. As shown in Figure 3.9(a), 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 in the console using the Print Line service.

The limit sensor built into the motor is simulated by the m-key. As shown in Figure 3.9(b), when the m-key is pressed while the door is opening or closing, the state will change and the result will be displayed.

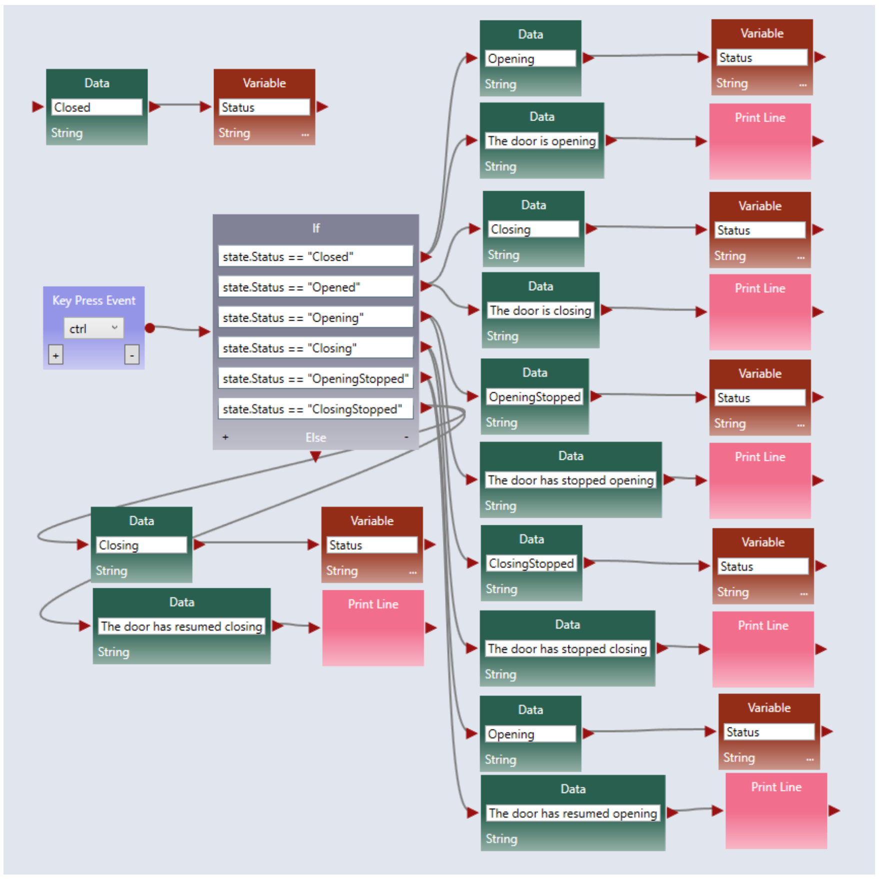
ASU VIPLE is based on the concept of computational thinking, which hides the technical and programming details from the user, and thus can be used for teaching high school students and college freshmen as their first programming language. It has been pilot-tested at Arizona State University to high school students in summer camps in 2015 and 2016. It is used in an ASU regular class, FSE100 (Introduction to Engineering) and several other universities worldwide. As VIPLE supports open architecture and open source development [3], VIPLE has been adopted by numerous universities around the world, and different VIPLE robots have been developed by universities and companies to support course development. A typical VIPLE robot costs less than $200, while a LEGO EV3 robot costs around $400.

Two robotics simulation environments have been developed: A Unity-based simulator and a Web-based simulator. These simulators allow VIPLE application development without physical robots.

3.6 Conclusions

This paper presented ASU VIPLE (Visual IoT/Robotics Programming Language Environment) and its mathematical model in π-calculus. ASU VIPLE allows novice programmers to program complex IoT devices and robots, and learn programming concepts. It supports LEGO EV3 and all IoT devices based on an open architecture. ASU VIPLE is free and well documented at:

http:// neptune.fulton.ad.asu.edu/WSRepository/VIPLE/



Data
Closed
String
Variable
Status
String
Key Press Event
ctrl
If
state.Status == "Closed"
state.Status == "Opened"
state.Status == "Opening"
state.Status == "Closing"
state.Status == "OpeningStopped"
state.Status == "ClosingStopped"
Else
Opening
The door is opening
Print Line
Closing
The door is closing
OpeningStopped
The door has stopped opening
ClosingStopped
The door has stopped closing
The door has resumed opening
The door has resumed closing

Figure 3.9(a) VIPLE Diagram Implementing a Garage Door Opener (Remote Button Press)

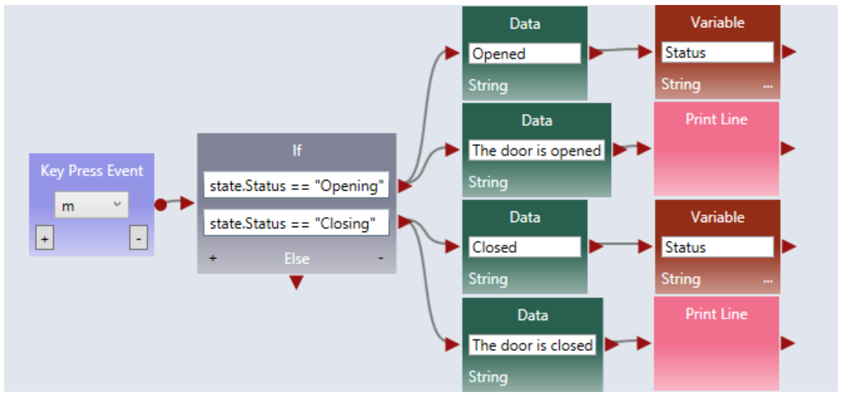


Figure 3.9(b) VIPLE Diagram Implementing a Garage Door Opener (Limit Sensor Touched)

CHAPTER 4

SEMANTIC ANALYSIS OF CONCURRENT COMPUTING IN DECENTRALIZED IOT AND ROBOTICS APPLICATIONS

4.0 Abstract

As IoT and robotics applications continue to become increasingly complex and decentralized, there is an increase in the difficulty of verifying requirements such as guarantees of reliability, efficiency, and correctness of the entire system. Modern IoT and robotics research and applications employ a variety of technologies to accommodate the distribution of real world IoT devices. Many of these technologies support the orchestration of these devices with concurrent computation and workflow-based processing, e.g., Intel Service Orchestration Layer. We developed a new workflow-based programming language VIPLE (Visual IoT/Robotics Programming Language Environment), which supports the orchestration of decentralized IoT and robotics devices through concurrent computation. VIPLE has been used by many universities for teaching robotics programming, event-driven computing, service-oriented computing, and parallel computing concepts. Historically, simple single threaded applications could be verified by employing formal systems such as Hoare logic. It is difficult to verify parallel and distributed computing. In our recent research, we developed a novel system to verify the semantics and other requirements of decentralized applications written in VIPLE. This system is developed using a combination of the techniques offered in Hoare logic and Rely-Guarantee logic, and it is applied using the automatically generated Pi-Calculus

representation of VIPLE applications. This paper focuses on the foundations of this system, its ability to handle concurrent computing and decentralized applications, as well as its ability to support automated verification and semantic analysis.

4.1 Introduction

As software development continues to evolve, the complexities of application development also grow. IoT and robotics applications also become increasingly complex and decentralized, bringing an assortment of related problems. These problems include communication between devices, coordination among devices, and verification of factors such as reliability, efficiency, and correctness of the entire system.

Internet of Things (IoT) was initially proposed and applied in the Radio-Frequency Identification RFID-tags for creating the Electronic Product Code (Auto-ID Lab) [8]. IoT concepts have been extended to refer to any system where physical objects are seamlessly integrated into the internet, and where the physical objects become active participants in industrial and business processes [9]. IoT can be a simple sensor connected to Internet. It can be a full computing unit. Internet of Intelligent Things (IoIT) deals with intelligent devices that have adequate computing capacity, such as including a 32-bit processor. Distributed intelligence is a part of the IoIT [10]. According to a 2016 ZDNet report, there are 8 billion intelligent devices that are connected to the Internet and are applied in various information systems [11].

An autonomous decentralized system (ADS) consists of decentralized modules or subsystems that are designed to operate independently but are capable of collaborating with each other to meet the overall goal of the system. ADS subsystems can cooperate in a loosely coupled manner and the data are shared through a content-oriented protocol. Traditional communication protocols route data by address. A content-oriented protocol route data by a content code, which indicates the purpose or domain of application of the data. This design paradigm enables the system to continue to function even when some subsystems fail. It also enables maintenance and repair to be conducted while the overall system is in operation. ADS and the related technologies have a large number of applications in manufacture production lines, transportation, and robotics [14][15]. ADS concepts are the foundation of the later technologies, such as service-oriented computing, cloud computing, enterprise service bus, and Internet of Things.

Robot as a Service (RaaS), proposed in our previous studies, is a cloud computing unit that allows the seamless integration of embedded devices, such as sensory, processing, and actuator devices used in robots, into Web and cloud computing environments [12][20]. In terms of service-oriented architecture (SOA), a RaaS unit includes services and activities for performing functionality, a service directory for discovery, and service clients for the user's direct access [13]. The current RaaS implementation facilitates SOAP and RESTful communications between RaaS units and the other Web and cloud computing units. The data transferred between the units is represented in JSON format. Hardware supports and standards are available to support RaaS implementations. For example, DPWS (Devices Profile for Web Services) defines standards and interfaces for enabling secure embedded service messaging, discovery, description, and eventing on resource-constrained devices between Web services and devices. The recent embedded SoC (System on a Chip), such as Galileo, Edison, Raspberry Pi, and EPS 8226, as well as the modern embedded system programming languages, such as Python and JavaScript, made it easy for programming these devices as Web services.

As IoT and robotics research and applications expand explosively into many domains in computing, information, machine learning, and control systems, schools and universities must prepare students to understand and to be able to program IoT devices and robots. However, programming IoT and physical devices is hard and depends on good understanding of hardware and low-level programming. Workflow and visual programming languages have been developed to address this issue.

To serve this purpose, we develop a graphic IoT/robotics programming environment at Arizona State University (ASU). The environment is called VIPLE, standing for Visual IoT/Robotics Programming Language Environment [3].

ASU VIPLE is built on our previous RaaS concepts and development environments [12]. It is designed for supporting different robotics platforms and simulation environments. It can connect to different physical robots through Wi-Fi and Bluetooth, including EV3, Wonder Workshop, and any robots based on the open architecture processors. Figure 4.1 illustrates the platforms and applications currently supported in VIPLE.

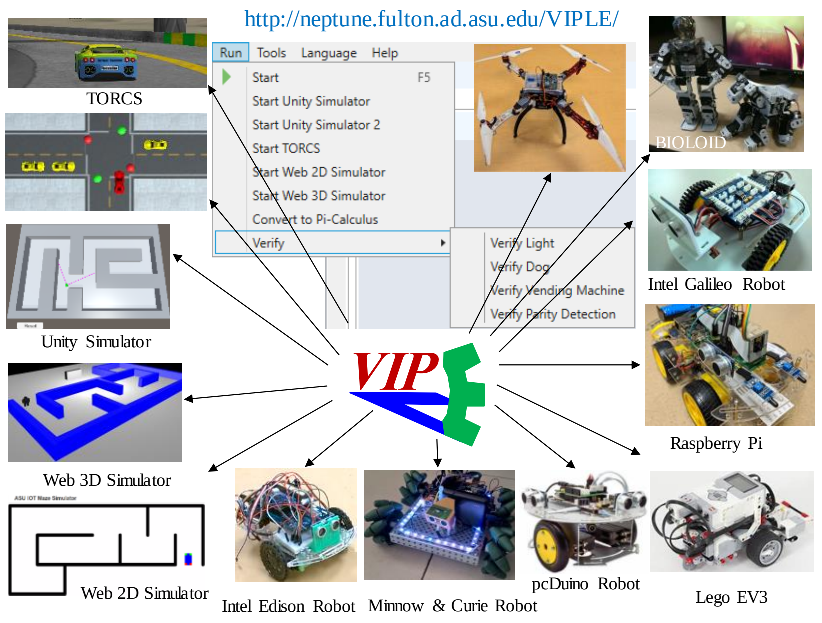


Figure 4.1 Platforms Supported in VIPLE

On the left side of Figure 4.1 are the simulation environments supported. It includes the TORCS racecar and autonomous driving simulator, a traffic simulator implemented in Unity, a maze navigation simulator in Unity, and two maze navigation simulators in HTML5. The rest of the platforms in Figure 4.1 are hardware systems. It includes a drone developed for flight data collection and flight-pattern machine learning, humanoid/dog robots, Lego EV3, and custom-built vehicles implemented using different processors. These platforms allow different universities to use simulators only, to purchase, or to build their hardware to support the course. The custom-built vehicles are based on open architecture and can also be used in other courses.

ASU VIPLE has been pilot tested at Arizona State University since summer 2015 as well as at several other universities. ASU VIPLE software and documents are free and can be downloaded at:

http:// neptune.fulton.ad.asu.edu/WSRepository/VIPLE/

The main goal of this paper is to provide an updated model to represent VIPLE using Pi-Calculus. The first version of the Pi-Calculus representation of VIPLE was immediately based on work by Puhlmann and Weske and their model for representing general workflow languages [23]. We have advanced that model by utilizing a variety of Pi-Calculus assets discussed in Milner’s book on Pi-Calculus [22]. Furthermore, we needed a mathematical foundation for performing automated semantic verification of VIPLE applications. Although Pi-Calculus is immediately suited to describing process communication and coordination, it lacks an easily accessible formulation of internal actions. Our previous attempts made use of τ throughout our model to represent actions performed by each activity that cannot be represented by Pi-Calculus [6]. As VIPLE is a well-defined workflow language, these actions can be simplified and updated to more specifically detail the behavior of each activity.

In addition, the work allows the instructors grade the students work automatically. The instructor can convent the VIPLE programs written by students into Pi-Calculus and perform semantic analysis and equivalence check with the model solution. Figure 4.2 shows the built-in menu for performing such verification. A few built-in examples have been giving to demonstrate the use of generating Pi-Calculus expressions and verifying the correctness of the Pi-Calculus expressions.

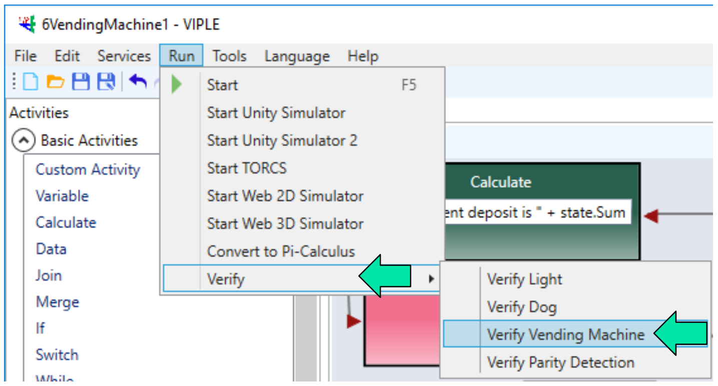


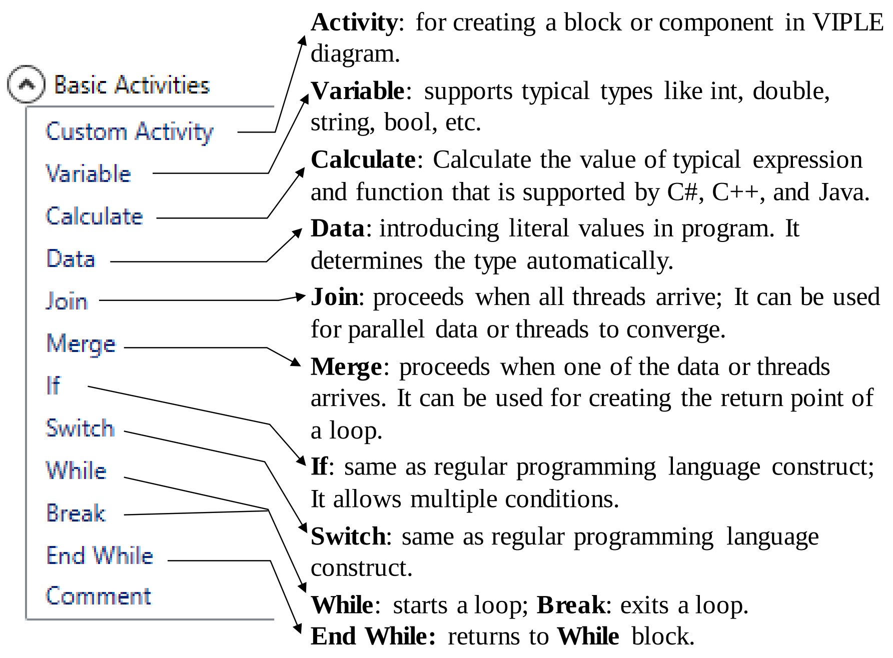
Figure 4.2 Convert VIPLE Program into Pi-Calculus for Verification

The rest of the paper is organized as follows. Section II presents ASU VIPLE, the Visual IoT/Robotics Programming Language Environment, as well as its updated Pi-Calculus representation. Section III discusses our theoretical foundation for verification and program correctness. Section IV presents an example of using our verifier to check the correctness of an NP-hard problem. Section V discusses the application of this verifier to robotic applications that can be characterized by a finite state machine. Section VI discusses the potential for verification of applications that use decentralized IoT devices. Section VII concludes the paper.

4.2 The definition of VIPLE in Pi-Calculus

The basic building blocks of an ASU VIPLE program are listed and explained in Figure 4.3. These activities include the basic constructs of a general-purpose programming language, and thus VIPLE can implement functionalities that a general-purpose programming language.

Each of these basic activities in VIPLE can be considered to be a fundamental process represented in the Pi-Calculus. Many of the definitions are unchanged, according to [6]. However, there are several important new additions. Below is the modified definition of Join, demonstrating the specialized value passing and calculations.



Activity: for creating a block or component in VIPLE diagram.

Variable: supports typical types like int, double, string, bool, etc.

Calculate: Calculate the value of typical expression and function that is supported by C#, C++, and Java.

Data: introducing literal values in program. It determines the type automatically.

Join: proceeds when all threads arrive; It can be used for parallel data or threads to converge.

Merge: proceeds when one of the data or threads arrives. It can be used for creating the return point of a loop.

If: same as regular programming language construct; It allows multiple conditions.

Switch: same as regular programming language construct.

While: starts a loop; **Break**: exits a loop.

End While: returns to **While** block.

Figure 4.3 Basic Activities in VIPLE

In the definitions, we consider the “A” processes to be inputs to the defined process and “B” is the process being defined. A τ action is used to represent that the previous process(es) perform some operation, but that operation is inconsequential to the definition. In the Join definition, we assume that the Join variable names are equivalent to the input channel names. An example of Join is shown in Figure 4.4.

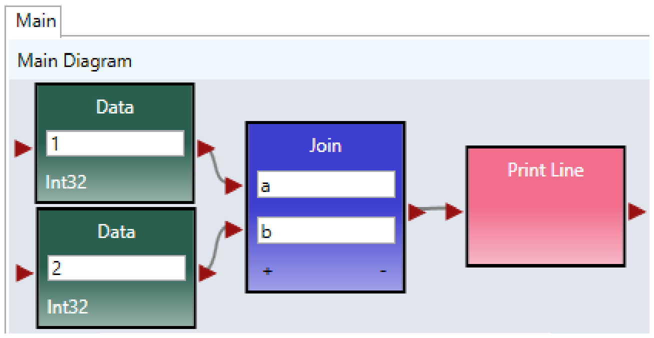
Join

$n \geq 1$

$S ::= new\ a1a2 \ldots an(A1|A2|\ldots|An|B)$

$An ::= \ \tau_{\mathrm{An}}.\overline{an}\langle x\rangle$

$B ::= a1(x1).a2(x2).\ldots.an(xn).y = \{(a1,x1),(a2,x2),\ldots,(an,xn)\}.\bar{b}\langle y\rangle$



x = 1.a0<x>
x = 2.a1<x>
a0(x0).a1(x1).x = {(a, x0), (b, x1)}.a2<x>
a2(x).τ

Figure 4.4 Simple Join in Pi-Calculus Example

The If, Switch, and Merge activities remain unchanged. In addition, VIPLE contains a variety of services beyond the Basic Activities shown in Figure 4.3. As these advanced services typically have no impact on the data (e.g. Print Line outputs the value, but does not change it), we continue to use τ actions to represent those services. However, we have added functionality to the Pi-Calculus definitions of the other Basic Activities. To simplify the definitions, we are only including the modified portions. Figure 4.4 includes an example of Data and Figure 4.5 includes an example with Calculate and Variable.

Variable

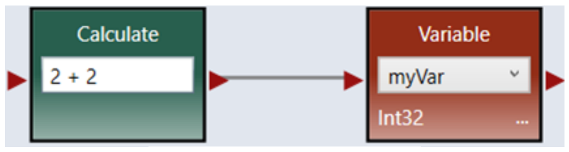
$B ::= a(x).var_{name} = x.\bar{b}\langle x\rangle$

Calculate

$B ::= a(x).y = Compute(expression).\bar{b}\langle y\rangle$

Data

$B ::= a(x).y = value.\bar{b}\langle y\rangle$



x = Compute(2 + 2).a0<x>
a0(x).myVar = x

Figure 4.5 Calculate and Variable in Pi-Calculus

4.3 Reasoning About Program Correctness

In order to perform verification on the VIPLE programs converted to Pi-Calculus, we employed a combination of several formal reasoning systems. Starting with a simple single-threaded application, we are able to perform automated reasoning using Hoare Logic. In order to verify a program, we define a set of Hoare triplets, P, C, Q, where P is the precondition, Q is the post-condition, and C is a command [25]. Using this formalism, we can trace the application's execution path and determine if the post-condition is met after the command is executed, assuming the precondition was true. This approach has several issues, the issue of halting and the issue of parallelism.

4.3.1 Issue of Parallelism

For multithreaded applications, such as decentralized IoT applications, the control flow cannot be easily followed without executing the application. To resolve the issue of parallelism, we employed a combination of Rely-Guarantee Logic and Separation Logic. Essentially, this combination of logics allows analysis of a program by assuming that the relevant pieces of memory are guaranteed to be the same regardless of order of execution. In our implementation, we made use of our modified Pi-Calculus representation to analyse all potential memory accesses to determine the maximum possible guarantees. We add on to the Hoare triple an internal pair of values to each string of processes, R and G where R is the set of memory values that the thread expects other threads will not modify and G is the set of memory values that the thread will not modify. Thus, we can analyse the Pi-Calculus representation of each thread in conjunction with the R and G sets to perform program verification.

4.3.2 Issue of Halting

In order to perform full formal verification, an application must be able to both guarantee correctness assuming halting and guarantee halting. The former condition is handled through our use of the aforementioned formal logics. In order to handle halting, we must consider the use of loops within an application. Since a Pi-Calculus representation can be represented as a directed graph [22], finding a loop in the program becomes a problem of finding a cycle in the Pi-Calculus representation. If no loop is found, the application is guaranteed to halt. However, determining if a loop will halt is the Halting Problem and is not always guaranteed to be possible [26].

Despite an inability to always reason about loops, we considered the subset of programs where reasoning is possible. To do so, we first analyse the loop to determine any potential exit conditions. We formulate a loop invariant so that if both the loop exit condition and the loop invariant are true, the loop is guaranteed to exit. Then, we use the loop invariant to prove termination. To prove termination, we first define a value e such that e is a natural number upon entering the loop and after every iteration. Furthermore, we prove that e is strictly decreasing. For example, e may decrease as a value increases in case of a condition such as: value <= n. Thus, e will be inversely proportional to value and will be indicative of whether the loop is continually nearing completion.

Once e is defined, there are two main components to our proof. The first component is to prove that e is always a natural number, and the second is that e is strictly decreasing. In the previous example, we would automatically define e to be an expression such as the absolute value of (n – value). As value increases, e decreases. The absolute value guarantees non-negative values, and e is guaranteed to be a natural number if n and value are integers. Floating point numbers are discussed below. With these preconditions, the proof that the loop terminates is as follows:

Proof: Consider a natural number e with values corresponding to closeness of a given loop invariant to true, with 0 representing equivalence of the loop invariant and true and larger numbers representing a greater distance between the loop invariant and true. Assume that e is strictly decreasing. Then, by the Principle of Well-Ordering, e is guaranteed to eventually equal 0. By extension, the loop invariant is guaranteed to equal true, and the loop is guaranteed to exit. Thus, a program using loops where the value e can be defined as above are guaranteed to halt.

One issue with the definition of e as a natural number is the case of floating-point values. However, since computer precision is finite and limited by the number of bits being used (e.g. 64-bit precision for a double), the floating-point value can be guaranteed to approach 0 in a finite number of steps due to the finite number of possible values that can be stored in a computing system.

4.4 Applying Verifier to NP-Complete Problems

One of the benefits of an automated verification system is the ability to quickly verify complex problems. In Figure 4.6 is the VIPLE code that finds a Hamiltonian Path in a given graph, which is represented in a binary format. The Hamiltonian Path finding problem is NP complex in computation time. However, it can be verified in polynomial time, despite the complexity of the application. Thus, the application of this verifier is not limited to simple applications with short Pi-Calculus representations. Figure 4.7 shows our Pi-Calculus representation for the Hamiltonian Path finding VIPLE code.

As the commands performed in the Hamiltonian Path algorithm are not dependent on the input, the logic definition discussed in an earlier section will remain relatively consistent across various inputs. This ease of modifying the corresponding logic files facilitates advanced testing, such as boundary testing. Such techniques may be especially valuable in the classroom to help students learn.

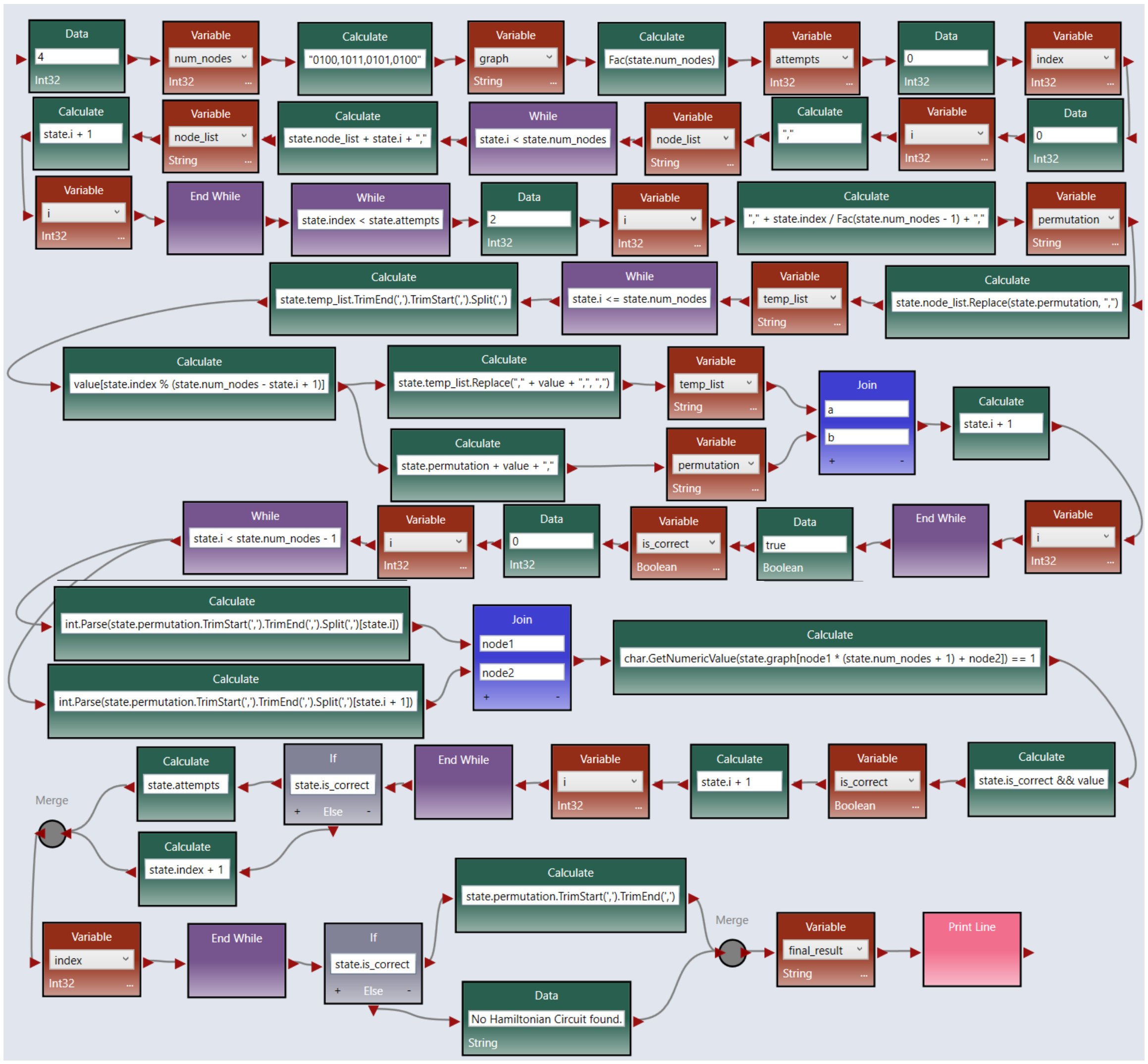
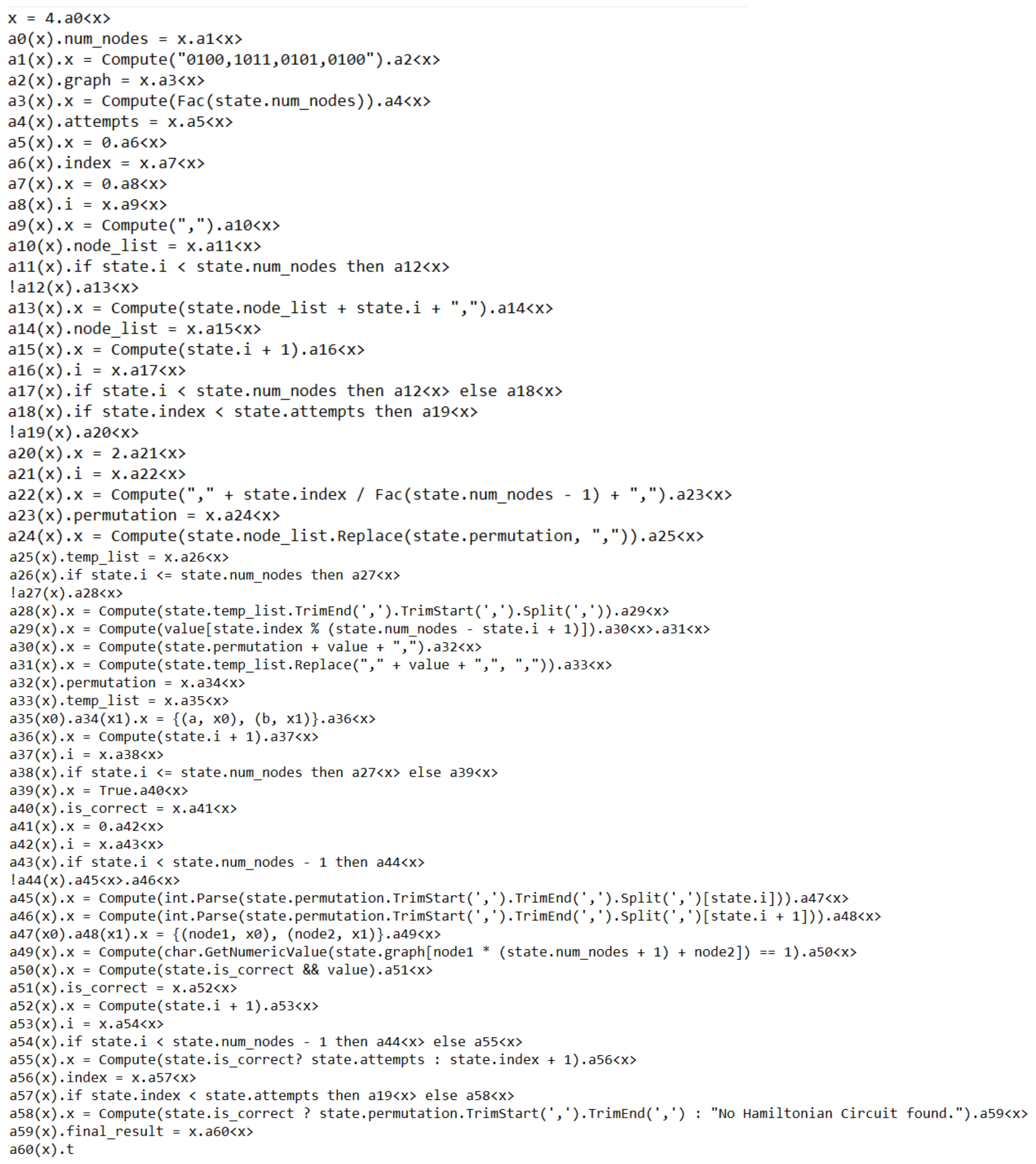

Data 4 Int32
Variable num_nodes Int32
Calculate "0100,1011,0101,0100"
Variable graph String
Calculate Fac(state.num_nodes)
Variable attempts Int32
Data 0 Int32
Variable index Int32
Data 0 Int32
Variable i Int32
Calculate ","
Variable node_list String
While state.i < state.num_nodes
Calculate state.node_list + state.i + ","
Variable node_list String
Calculate state.i + 1
Variable i Int32
End While
While state.index < state.attempts
Data 2 Int32
Variable i Int32
Calculate "," + state.index / Fac(state.num_nodes - 1) + ","
Variable permutation String
Calculate state.node_list.Replace(state.permutation, ",")
Variable temp_list String
While state.i <= state.num_nodes
Calculate state.temp_list.TrimEnd(',').TrimStart(',').Split(',')
Calculate value[state.index % (state.num_nodes - state.i + 1)]
Calculate state.temp_list.Replace("," + value + ",", ",")
Variable temp_list String
Calculate state.permutation + value + ","
Variable permutation String
Join a b
Calculate state.i + 1
Variable i Int32
End While
Data true Boolean
Variable is_correct Boolean
Data 0 Int32
Variable i Int32
While state.i < state.num_nodes - 1
Calculate int.Parse(state.permutation.TrimStart(',').TrimEnd(',').Split(',')[state.i])
Calculate int.Parse(state.permutation.TrimStart(',').TrimEnd(',').Split(',')[state.i + 1])
Join node1 node2
Calculate char.GetNumericValue(state.graph[node1 * (state.num_nodes + 1) + node2]) == 1
Calculate state.is_correct && value
Variable is_correct Boolean
Calculate state.i + 1
Variable i Int32
End While
If state.is_correct Else
Calculate state.attempts
Calculate state.index + 1
Merge
Variable index Int32
End While
If state.is_correct Else
Calculate state.permutation.TrimStart(',').TrimEnd(',')
Data No Hamiltonian Circuit found. String
Merge
Variable final_result String
Print Line

Figure 4.6 VIPLE Program for Finding Hamiltonian Path

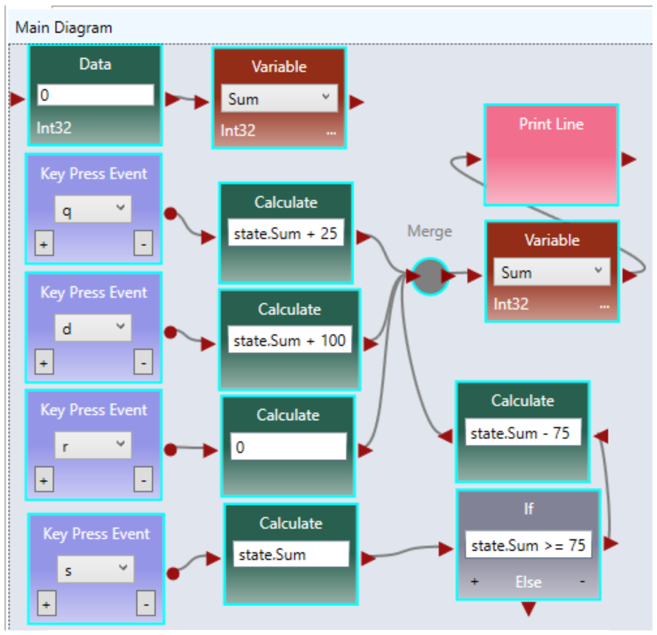


```
x = 4.a0<x>
a0(x).num_nodes = x.a1<x>
a1(x).x = Compute("0100,1011,0101,0100").a2<x>
a2(x).graph = x.a3<x>
a3(x).x = Compute(Fac(state.num_nodes)).a4<x>
a4(x).attempts = x.a5<x>
a5(x).x = 0.a6<x>
a6(x).index = x.a7<x>
a7(x).x = 0.a8<x>
a8(x).i = x.a9<x>
a9(x).x = Compute(",").a10<x>
a10(x).node_list = x.a11<x>
a11(x).if state.i < state.num_nodes then a12<x>
!a12(x).a13<x>
a13(x).x = Compute(state.node_list + state.i + ",").a14<x>
a14(x).node_list = x.a15<x>
a15(x).x = Compute(state.i + 1).a16<x>
a16(x).i = x.a17<x>
a17(x).if state.i < state.num_nodes then a12<x> else a18<x>
a18(x).if state.index < state.attempts then a19<x>
!a19(x).a20<x>
a20(x).x = 2.a21<x>
a21(x).i = x.a22<x>
a22(x).x = Compute("," + state.index / Fac(state.num_nodes - 1) + ",").a23<x>
a23(x).permutation = x.a24<x>
a24(x).x = Compute(state.node_list.Replace(state.permutation, ",")).a25<x>
a25(x).temp_list = x.a26<x>
a26(x).if state.i <= state.num_nodes then a27<x>
!a27(x).a28<x>
a28(x).x = Compute(state.temp_list.TrimEnd(',').TrimStart(',').Split(',')).a29<x>
a29(x).x = Compute(value[state.index % (state.num_nodes - state.i + 1)]).a30<x>.a31<x>
a30(x).x = Compute(state.permutation + value + ",").a32<x>
a31(x).x = Compute(state.temp_list.Replace("," + value + ",", ",")).a33<x>
a32(x).permutation = x.a34<x>
a33(x).temp_list = x.a35<x>
a35(x0).a34(x1).x = {(a, x0), (b, x1)}.a36<x>
a36(x).x = Compute(state.i + 1).a37<x>
a37(x).i = x.a38<x>
a38(x).if state.i <= state.num_nodes then a27<x> else a39<x>
a39(x).x = True.a40<x>
a40(x).is_correct = x.a41<x>
a41(x).x = 0.a42<x>
a42(x).i = x.a43<x>
a43(x).if state.i < state.num_nodes - 1 then a44<x>
!a44(x).a45<x>.a46<x>
a45(x).x = Compute(int.Parse(state.permutation.TrimStart(',').TrimEnd(',').Split(',')[state.i])).a47<x>
a46(x).x = Compute(int.Parse(state.permutation.TrimStart(',').TrimEnd(',').Split(',')[state.i + 1])).a48<x>
a47(x0).a48(x1).x = {(node1, x0), (node2, x1)}.a49<x>
a49(x).x = Compute(char.GetNumericValue(state.graph[node1 * (state.num_nodes + 1) + node2]) == 1).a50<x>
a50(x).x = Compute(state.is_correct && value).a51<x>
a51(x).is_correct = x.a52<x>
a52(x).x = Compute(state.i + 1).a53<x>
a53(x).i = x.a54<x>
a54(x).if state.i < state.num_nodes - 1 then a44<x> else a55<x>
a55(x).x = Compute(state.is_correct? state.attempts : state.index + 1).a56<x>
a56(x).index = x.a57<x>
a57(x).if state.index < state.attempts then a19<x> else a58<x>
a58(x).x = Compute(state.is_correct ? state.permutation.TrimStart(',').TrimEnd(',') : "No Hamiltonian Circuit found.").a59<x>
a59(x).final_result = x.a60<x>
a60(x).t
```

Figure 4.7 Pi-Calculus Representation of the Hamiltonian Path Finding VIPLE Code

4.5 Applying Verifier to Finite State Machines

One example of an IoT application commonly developed in VIPLE is a maze navigation algorithm. One method of implementing maze navigation is through the right-wall following algorithm, which is essentially a finite state machine. To prepare students for writing this algorithm, they may create other finite state machines with user input rather than automated robot input. Due to the Pi-Calculus foundation we used, we are able to analyze applications that use user input over a given input space. Figure 4.8 demonstrates a simple vending machine example with internal storage. In this example, key press events are used, which can be represented as standard IoT events [6]. Applying the verifier will test a collection of test cases as defined in the logic file in the P, C, Q format discussed earlier. For example, if a student forgets to include the If check after requesting a soda, the verifier will warn the student about that issue.



!KeyPress(Q).a0<x>
x = 0.a1<x>
!KeyPress(D).a2<x>
!KeyPress(R).a3<x>
!KeyPress(S).a4<x>
a0(x).x = Compute(state.Sum + 25).a5<x>
a1(x).Sum = x
a2(x).x = Compute(state.Sum + 100).a5<x>
a3(x).x = Compute(0).a5<x>
a4(x).if state.Sum >= 75 then a6<x>
!a5(x).a7<x>
a6(x).x = Compute(state.Sum - 75).a5<x>
a7(x).Sum = x.a8<x>
a8(x).τ

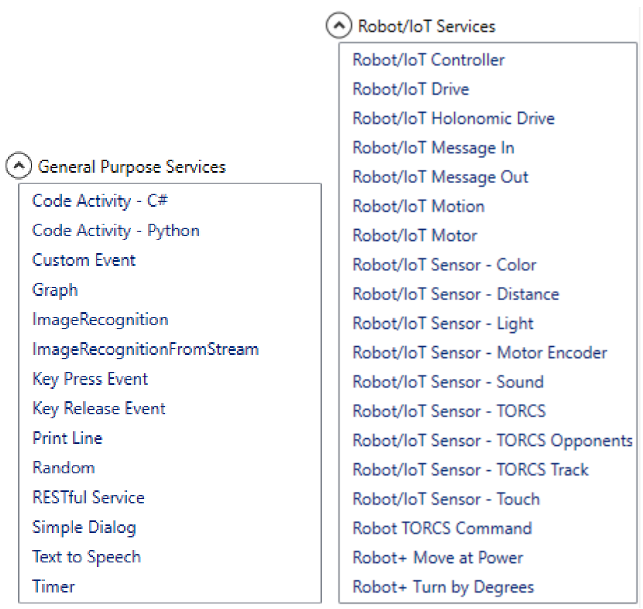
Figure 4.8 Vending Machine Example

4.6 Applying Verifier to Decentralized Applications

One of the benefits of VIPLE outside of the classroom is its ability to facilitate communication and processing among decentralized devices. In typical distributed applications, one of the major development issues is quality assurance and testing. As described above, our verifier facilitates such tests in a variety of applications including complex problems, finite state machines, and event-oriented applications.

By making use of multiple IoT controllers in a VIPLE application, programming decentralized devices is greatly facilitated. Events can then be attached to each device to perform data processing unique to each device, or in conjunction with other devices. Regardless of processing performed, the underlying Pi-Calculus representation of these events and services is consistent with the above examples. Thus, the same techniques can be used to verify complex distributed applications.

Figure 4.9 shows the built-in services that perform general purpose computing and facilitate the connection and communication between VIPLE diagram and physical robot/IoT devices, including sensors, motors, and complex devices. The complex services include Robot/IoT Controller, Robot/IoT Holonomic Drive, Robot/IoT Motion, and all the TORCS sensors. In the current implementation, these services are modelled as black boxes and their functions are assumed to be correct.


General Purpose Services

- Code Activity - C#
- Code Activity - Python
- Custom Event
- Graph
- ImageRecognition
- ImageRecognitionFromStream
- Key Press Event
- Key Release Event
- Print Line
- Random
- RESTful Service
- Simple Dialog
- Text to Speech
- Timer

Robot/IoT Services

- Robot/IoT Controller
- Robot/IoT Drive
- Robot/IoT Holonomic Drive
- Robot/IoT Message In
- Robot/IoT Message Out
- Robot/IoT Motion
- Robot/IoT Motor
- Robot/IoT Sensor - Color
- Robot/IoT Sensor - Distance
- Robot/IoT Sensor - Light
- Robot/IoT Sensor - Motor Encoder
- Robot/IoT Sensor - Sound
- Robot/IoT Sensor - TORCS
- Robot/IoT Sensor - TORCS Opponents
- Robot/IoT Sensor - TORCS Track
- Robot/IoT Sensor - Touch
- Robot TORCS Command
- Robot+ Move at Power
- Robot+ Turn by Degrees

Figure 4.9 General Purpose and Robot/IoT Services

As an example, Figure 4.10 shows our latest application of creating a robotic guide dog to help the vision-impaired person [27]. The robot dog is an autonomous entity and the wheelchair that carries the vision-impaired person is another autonomous entity. These two entities are coordinated through VIPLE code, where two Robot/IoT Controllers are used.

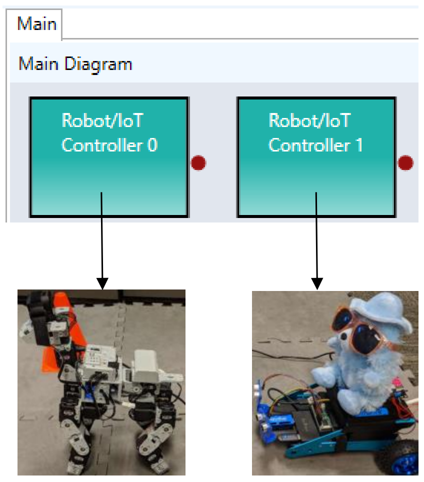


Figure 4.10 Two Robot/IoT Controllers Used in VIPLE

The robot dog has many complex movements. In addition to forward, backward, left-turn and right-turn, it can perform many actions, such as shaking hand, rolling, and moving head. It is programmed with Robot/IoT Motion service. The wheelchair hosts an Intel Up Squared board and an Intel Movidius Neural Compute Stick for processing sensor data camera image data. The robot dog helps the vision-impaired to avoid obstacle, cars on the road, and to recognize the traffic signals.

4.7 Conclusions

This paper presented ASU VIPLE (Visual IoT/Robotics Programming Language Environment), its mathematical model in π-calculus, and our novel automated semantic verifier. This verifier supported many different applications, with limits only existing with certain loop invariants, many of which can be transformed into valid conditions. The verifier can support education, prototyping, or decentralized IoT application development. ASU VIPLE is an active research and education platform. Software and hardware components have been constantly added into the platform for broader and deeper applications around IoT and robotics applications. ASU VIPLE is free and well documented at:

http:// neptune.fulton.ad.asu.edu/WSRepository/VIPLE/

CHAPTER 5

EXPLAINABLE SELF-LEARNING THEOREM PROVER

5.1 Theorem Prover Introduction

The papers published and employed for Chapters 3 and 4 present a strong mathematical background for VIPLE. This background enables the direct conversion of a VIPLE program into a Pi-Calculus representation. Furthermore, the modified Pi-Calculus enables the direct translation of not only VIPLE program communication and coordination, but also the VIPLE activity behavior. By employing the Hoare Logic approach discussed in Chapter 4 in conjunction with this modified Pi-Calculus representation, reasoning can be performed on any VIPLE program. This chapter will go into greater detail on how this reasoning is performed and formal definitions of the program behavior and analysis. Building on this reasoner, AI techniques are employed to develop a Theorem Prover that is also self-learning and an explainable AI.

With VIPLE, its modified Pi-Calculus representation, and the Hoare Logic foundation, the explainable self-learning theorem prover can be formally defined. The approach I have taken is to create a theorem prover that is able to reason about a given set of rules to determine whether a certain condition is met. Furthermore, the use of a theorem prover facilitates the introduction of explainability. As the inference rules are well defined, the AI's behavior is also well defined and clearly explainable.

There are three core components that enable the creation of this theorem prover [28]. They are:

1. An ontology that contains all the information about the program being analyzed;
2. A set of conditions to be analyzed for veracity (i.e. “sentences”); and
3. A formal set of rules that enables inference and reasoning about the ontology in order to verify the conditions.

This chapter will demonstrate each of these three components to illustrate the formal definition of the theorem prover.

5.2 Ontology

To define the ontology used in this theorem prover, I will first return to the formal definition provided by Milner of Pi-Calculus:

“The set P^{π} of π-calculus process expressions is defined by the following syntax:

$$P ::= \sum_{i \in I} \pi_i . P_i \mid P1 | P2 \mid new\ a\ P \mid !P$$

where *I* is any finite indexing set. The processes $\sum_{i \in I} \pi_i . P_i$ are called summations or sums” [22].

As discussed in Chapter 3, this definition is applied to VIPLE in order to provide a formal definition of any VIPLE program. The action prefix π is also formally defined by Milner as:

“$\pi ::= x(y)$ receive y along x

$x\langle y \rangle$ send y along x

τ unobservable action” [22].

By the modifications presented in Chapter 4, the unobservable action τ is replaced by observable actions within VIPLE. These actions do not directly interact with the Pi-

Calculus communication or coordination. Rather, these actions may produce values that can be used to determine branching that is already defined within the Pi-Calculus expressions. Alternatively, these actions may be employed to interact with the Variable Table to send or receive values along existing Pi-Calculus channels. The Variable Table is a two-dimensional data storage mechanism that tracks all defined VIPLE variables within the given program. The values can only be modified or read by employing one of the new observable actions. In this way, symbolic execution can be applied to perform reasoning about certain parts of the program by only analyzing the Pi-Calculus expressions.

By employing these Pi-Calculus expressions and the Variable Table, a full ontology can be automatically generated for any given VIPLE program.

Example 5.1: Consider the example program shown in Figure 5.2.

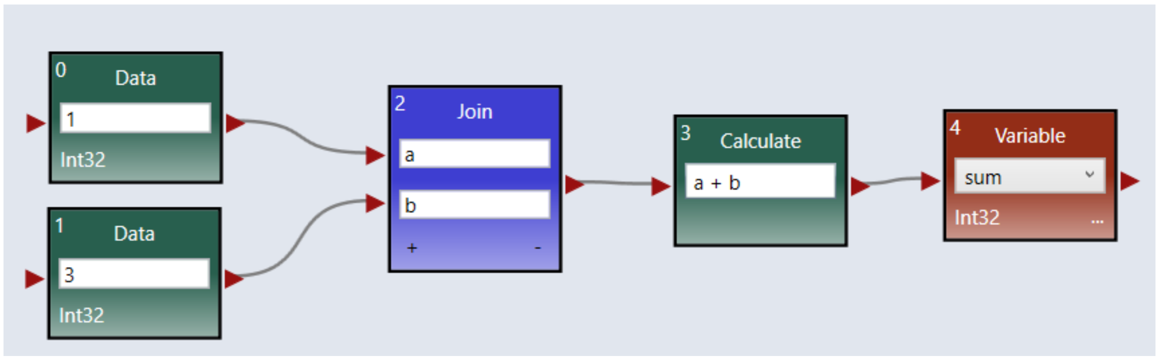


Figure 5.2 Sum Example

Applying the Pi-Calculus expression rules defined in Chapter 4, the code will be translated into the following set of Pi-Calculus expressions:

$$x_0 = 1.\overline{a_0}\langle x_0\rangle$$

$$x_1 = 3.\overline{a_1}\langle x_1\rangle$$

$$a_0(x_2).a_1(x_3).x_4 = \{(a, x_2), (b, x_1)\}.\overline{a_2}\langle x_4\rangle$$

$$a_2(x_5).x_6 = Compute(x_5[a] + x_5[b]).\overline{a_3}\langle x_6\rangle$$

$$a_3(x_7).VariableTable[sum] = x_7$$

This example will be returned to in the following sections of this chapter, to illustrate the complete lifecycle of the theorem prover.

5.3 Verifiable Conditions

The second prerequisite of the theorem prover is the definition of a set of verifiable conditions that define the correctness of a given VIPLE program. These verifiable conditions are defined by employing Hoare Logic, which was briefly discussed in Chapter 4. These conditions are specified by first defining a set of Hoare triplets, P, Q, R, where P is the precondition, R is the postcondition, and Q is a command [29]. Hoare specifies the application of these three components in verifying aspects of a program in his paper. He said, “If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion” [29]. This section will examine each of these three parts of the Hoare triplet and their applications. It will culminate with a continuation of Example 5.1 with the definition of Hoare triplets for verification of program correctness.

5.3.1 Hoare Triplet Commands

The first element of the Hoare triplet that I will examine is the command, Q, also known as the “program” in Hoare’s paper [29]. A single Hoare triplet may not be able to correctly define all important behaviors of a given program, but only part of a program. As such, the command is dynamically defined during verification of the program. There are two mechanisms by which this dynamic definition is performed. First, in the case of event-driven VIPLE programs, events can be specified as part of the Hoare triplet in place of the command. As these VIPLE event activities are translated into Pi-Calculus expressions, the utilization of an event in a Hoare triplet is equivalent to invoking a new Pi-Calculus term. Consider for example a program that employs the key press event of ‘A’. Then, it may have a Pi-Calculus term that appears as follows:

$$!\,KeyPress_A(x_0).\overline{a_1}\langle x_0\rangle$$

This term awaits the receipt of a value along the channel corresponding to A’s key press event, namely the event $\mathrm{KeyPress}_\mathrm{A}$. Of important note is the application of the replication operator, as was discussed in Chapter 3. Events of any variety may be repeatedly triggered. As such, the formal Pi-Calculus definition reflects this behavior through the replication operator. To test the code gated behind such an operator, a single Pi-Calculus expression can be employed to open the channel by sending the expected value to that specific channel. Specifics of channel communication and coordination will be covered later in this chapter. Below is the Pi-Calculus expression which performs this task.

$$x_0 = KeyPressed_A.\overline{KeyPress_A}\langle x_0\rangle$$

The second mechanism of dynamic command generation is employed in cases both with and without events. This mechanism is the continual verification of the VIPLE program's Pi-Calculus expressions until either (1) the postcondition is met or (2) the program terminates. This behavior is also formalized in the later section on Pi-Calculus channel communication and coordination. By employing these dynamic command generation mechanisms, multiple Hoare triplets can be analysed sequentially, enabling a more thorough analysis of a given VIPLE program.

5.3.2 Hoare Triplet Conditions

I will treat the other two parts of a Hoare Triplet, the precondition P and postcondition R, together in this section. Both of these parts are conditions and share syntax. As Hoare mentions in his paper, the main method of evaluating a program is an assignment condition, $x \coloneqq f$, where x is a variable identifier and f is an expression from the programming language without side effects [29]. Using this condition, Hoare offers the Axiom of Assignment which forms the foundation for evaluation of multiple conditions. Namely, $\vdash P_0\{x \coloneqq f\}P$ where P_0 is obtained by substituting f in place of x in P. Building on this rule, more specific rules can be developed, including rules of consequence, composition, and iteration [29]. In terms of this work, the application of these rules and axioms enables the analysis of a complete program and its behavior by analyzing each step of the program. Furthermore, since f can be any expression without side effects, Hoare Logic enables and facilitates the analysis of VIPLE programs as no VIPLE expressions have side effects. Thus, the precondition P and postcondition R can be constructed by employing expressions in VIPLE syntax and assignment operators (which are represented in VIPLE by the use of the Variable activity).

Following from these guidelines, the purpose of the postcondition is clearly to specify the expected state of the program at any given point. For example, a single, trivial Hoare triplet such as $true \left\{x := \frac{f}{2} * 2\right\} x := f$ can be used to evaluate whether the given program (the part in curly braces) culminates in x being f. Furthermore, true can be used as a precondition to signify that this postcondition has no prerequisites and can be evaluated starting at the beginning of the program. However, the precondition serves two vital roles in cases with multiple Hoare triplets. First, the precondition enables precise definition of the required state of the program before evaluating the postcondition. For example, perhaps a program is testing the following Hoare triplet: $true \left\{x := \frac{2}{f} * k\right\} x := 27$. In this case, a precondition could be vital to ensure that attempts to evaluate the expression avoid invalid states. For example, the precondition $f \neq 0$ prevents division by 0. Most importantly, a continuation of postcondition to precondition enables the application of the inference rule from Hoare's paper, the Rule of Consequence [29]. This inference rule is vital in cases with multiple Hoare triplets. The Rule of Consequence states:

$$If \ \vdash P\{Q\}R \ and \ \vdash R \supset S \ then \vdash P\{Q\}S.$$

The application of the Rule of Consequence implies that the entire program is correct assuming that each sub-condition is verified and that the sub-conditions are connected in this manner. Essentially, this rule enables the analysis of the program step by step while still guaranteeing that the entire program is correct.

5.3.3 Hoare Triplet Example

This section will return to Example 5.1 in order to demonstrate how to manually construct the Hoare triplets for the given program. This process will follow the rules described above. First, as mentioned above, the commands are dynamically constructed during verification. The only case where manual definition of commands is required is in event driven programs, when a certain postcondition requires an event to occur. However, in this example there are no events. If both Data at the start of the program were gated by a Key Press Event activity, a key press event would need to be included in the first command, in order to start verification of the program. In such a case, a lack of the event in the command would result in no program execution and immediate postcondition failure (assuming non-tautological postconditions).

The number of Hoare triplets (conditions in terms of a theorem prover) is proportional to the depth of analysis of a given program. In Example 5.1, a weak verification could be done with a single Hoare triplet, such as $true\{\ \}state.sum == 4$ (note that the conditions in a Hoare triplet are defined in the target language's (i.e. VIPLE's) syntax, not in the Pi-Calculus syntax [29]). This condition is weak, as it can pass any program that culminates in sum being 4, including the trivial program shown in Figure 5.3.

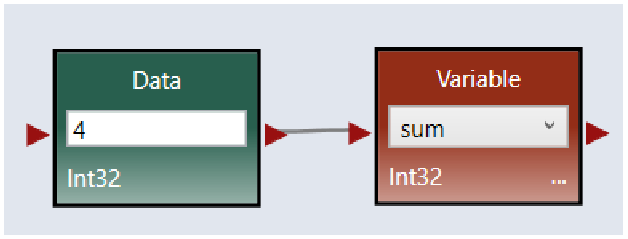


Figure 5.3 Trivial Assignment

Adding more conditions can enable more precise analysis of the programs. For example, perhaps the use of a Join is enforced with those specific values (1 and 3). In this slightly more detailed approach, multiple Hoare triplets would be required. One such example is shown below.

$$true\{\,\}value["a"] == 1 \;\&\&\; value["b"] == 3$$

$$value["a"] == 1 \;\&\&\; value["b"] == 3\{\,\}state.sum == 4$$

In this example, the values in the Join are first verified, which increases the strength of the set of conditions. As multiple conditions are used in this example, each postcondition and following precondition matches up for the sake of inference, as discussed above. Furthermore, the fully detailed VIPLE syntax is employed, such as the use of value[“a”]. Creating these detailed conditions (instead of only the join variable name) offers the user more flexibility. Rather than being forced to use Join, any mechanism they employ that generates the dictionary is acceptable. The condition developer may choose to enforce specific syntax (i.e. force the use of Join) or to accept any syntax, assuming correct semantics. Typically, there are very few options for enforcing syntax. This lack of options is a result of the system’s focus on semantic verification and VIPLE’s lack of focus on syntax, especially in its expression set. How these Hoare triplets are used to verify the program through the Pi-Calculus will be detailed in the following section.

5.4 Knowledge Reasoning

The final prerequisite of the theorem prover is a formal set of rules that enables inference and reasoning about the ontology in order to verify the conditions. This set of inference rules will enable the theorem prover to reason about the given ontology (the Pi-Calculus representation) to verify each of the sentences (the Hoare Logic triplets). There are two components required for the theorem prover to be able to analyze the Pi-Calculus expressions and verify the Hoare Logic triplets. First, it must be able to reason about the Pi-Calculus expressions' communication and coordination. Second, it must be able to evaluate the Hoare Logic triplets using the Pi-Calculus expressions and Variable Table.

5.4.1 Pi-Calculus Inference Rules

In order to accurately represent VIPLE programs, I have employed a specially modified variant of Pi-Calculus unique to VIPLE. In order to reason about the program, the inference rules need to be examined for correctness despite those modifications. The key aspect of analyzing the Pi-Calculus expression lies in the analysis of how these expressions communicate and coordinate with each other. These behaviors are contained within the term "reaction" as defined by Milner [22]. There are several vital reaction rules that are defined by Milner for Pi-Calculus. Those rules are listed below [22].

$$TAU: \tau . P + M \rightarrow P$$

$$REACT: (x(y).P + M)|(\bar{x}\langle z\rangle .Q + N) \rightarrow \{^z/_y\}P|Q$$

$$PAR: \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$RES: \frac{P \rightarrow P'}{new\ x\ P \rightarrow new\ x\ P'}$$

These rules succinctly define several important reaction concepts that are vital to the analysis of the Pi-Calculus expressions. First, the REACT rule defines how processes can communicate and coordinate. Namely, if process P is preceded by the receiving of y along x and process Q is preceded by the sending of z along x, the communication will immediately occur, sending z to P (denoted by the replacement of y by z in P). Concurrently, Q continues execution. Using this rule, complex sets of Pi-Calculus expressions can communicate with each other. Furthermore, a simple lemma that can be obtained from the REACT and PAR rules enables sequential coordination. The PAR rule states that one process can independently execute regardless of other processes executing in parallel. In other words, parallel processes can execute concurrently. Consider the below Pi-Calculus expression:

$$(P.x(y).\mathbf{0} + M)|(\bar{x}\langle z\rangle .Q + N)$$

From the PAR rule, P can execute in parallel of the second process. After P completes execution, the next process in the sequence is executed, namely x(y). From the REACT rule, this expression becomes:

$$(\mathbf{0})|(\bar{x}\langle z\rangle .Q + N)$$

The nil activity further reduces as follows:

$$(\bar{x}\langle z\rangle .Q + N)$$

As such, P and Q were forced to execute sequentially, despite their expressions being in parallel. In this manner, all VIPLE activities are considered to exist in parallel. This parallelism enables the direct representation of activities as distinct processes, rather than considering all sequential processes as part of one process. The distinction of processes is vital to the reasoning of programs, such as in the dynamic generation of Hoare Logic commands. Furthermore, despite all activities being in parallel, these rules enable both coordination and communication between the activities.

The TAU rule must be slightly changed to correctly apply to the updated Pi-Calculus representation used in VIPLE. In that new representation, tau actions are replaced by representations of the activities' actual behavior. However, since these actions do not directly modify or communicate with the Pi-Calculus channels, they do not affect the Pi-Calculus process execution or inference rules. By replacing tau with any VIPLE activity action, the inference rule remains valid. Consider for example the Calculate action. The rule can be modified as follows while remaining valid:

$$CALCULATE: Compute(expression).P + M \rightarrow P$$

Identical rules can be constructed for variable assignment and data values.

As mentioned above, the Variable Table is completely independent from the Pi-Calculus channels, so interfacing with the table is also an acceptable action that does not affect the reaction rules for Pi-Calculus expressions. However, an issue arises in the case of Join. Join employs the sending and receiving of a vector value (the dictionary), which is not directly supported in the monadic Pi-Calculus. As such, the polyadic Pi-Calculus can be employed in such cases. The main difference between the two is the ability to send multiple values along a single channel. This guarantees that all values are received by a single receiving process and that all values are received at once. In order to support the polyadic Pi-Calculus, the reaction rule must be updated to support vectors, as explained by Milner [22]. The updated REACT rule is below (note that the vectors $\vec{z}$ and $\vec{y}$ must have the same length:

$$REACT\colon (x(\vec{y}).P + M)|(\bar{x}\langle\vec{z}\rangle.Q + N) \rightarrow \left\{{}^{\vec{z}}/_{\vec{y}}\right\} P|Q$$

The combination of these rules enables the complete evaluation of any set of VIPLE specific Pi-Calculus expressions. The next section will detail how the Hoare Logic triplets can be evaluated, despite being written in VIPLE syntax and not as Pi-Calculus expressions.

5.4.2 Hoare Logic Evaluation in Pi-Calculus

By employing those reaction rules, the VIPLE program's Pi-Calculus representation can be analyzed for specific communication or coordination behaviors. However, the conditions to be analyzed are written as Hoare Logic triplets, and therefore use the VIPLE syntax. I considered two solutions for resolving this issue. First, the Hoare triplets could be written in the modified Pi-Calculus syntax. However, Pi-Calculus expressions are restrictive in terms of specifying the behavior of a VIPLE program. By introducing new Pi-Calculus expressions, the behavior of the system changes, since communication will be sent to these new terms. One of the rules of Hoare triplets is that expressions must not have side effects, but Pi-Calculus expressions would break that rule. Furthermore, I wanted a solution that could ultimately be feasibly employed by instructors lacking the background Pi-Calculus understanding. VIPLE syntax, on the other hand, is much simpler and lacks side effects. The solution I chose to pursue is an evaluation of the Hoare triplet conditions by analyzing the effects of the Pi-Calculus expressions.

There are three components required to analyze Hoare triplets according to the effects of the Pi-Calculus expressions. They are (1) the translation of state variables to the Variable Table, (2) the analysis of values being sent along channels, and (3) the evaluation of VIPLE expressions. Furthermore, the analysis of Hoare Logic triplets is dependent on the concept of dynamic command definition, as mentioned above. In order to dynamically define commands, Pi-Calculus expressions were defined distinctly, with each activity being its own expression. In this way, the command can be constructed with a single activity at a time. One of the main issues with this approach is the problem of parallelism. To resolve this issue, I employed a modified backtracking approach. Essentially, a tree is constructed based on the Pi-Calculus expressions for the VIPLE program. A breadth-first analysis is conducted. All Pi-Calculus nodes from that level of the tree are marked as the Hoare triplet command. Then, the postcondition is checked according to the rules which will be described below. Note that the precondition is always checked before constructing the command. Once a specific path through the tree is found that satisfies the postcondition, all the nodes in that path are marked, and the other nodes are released. In this way, nodes from other paths (i.e. threads) that do not contribute to the postcondition are not consumed by the analysis of that Hoare triplet. With this command construction and postcondition analysis algorithm, solutions to the aforementioned issues can be defined.

The first issue (translation of state variables to the Variable Table) is easily resolved by employing the Variable Table, which is already independent of the Pi-Calculus expressions. Whenever the postcondition needs to be evaluated, references to state variables (e.g. state.variableName) are replaced by references to the Variable Table (e.g. VariableTable[“variableName”]).

The second issue (analysis of values sent along channels) can be resolved during the application of the backtracking algorithm. Each time a node on the tree is evaluated, its children are designated according to the Pi-Calculus communication that occurred following the reaction rules detailed above. Each child will receive a single input along a single channel from its parent. Since the polyadic Pi-Calculus is employed, even Join parents are guaranteed to send a single input (a vector) along a single channel to each of its children. Thus, after a command is evaluated, the values sent along these channels can be immediately analyzed for the Hoare Logic triplet postconditions. References in these postconditions to “value” will be replaced by a reference to these values sent down the tree.

The final issue (evaluation of VIPLE expressions) can be resolved now that the previous two issues have been resolved. Since these expressions are guaranteed to not have side effects, as discussed earlier, the only difficulty in their analysis is the retrieval of variables (resolved using the Variable Table) and the retrieval of values sent along channels (resolved during the backtracking algorithm). After these two adjustments are applied, the expressions can be immediately resolve by employing the VIPLE expression evaluation algorithm. This algorithm is essentially a stateless (i.e. no side effects) interpretation engine that inputs the given syntax and offers the resulting value.

Employing this approach enables the complete evaluation of any of the Hoare Logic triplet conditions according to the effects of the Pi-Calculus expressions. These rules will be applied in the following section to the original example.

5.4.3 Knowledge Reasoning Example

In order to perform the knowledge reasoning for Example 5.1, both the Pi-Calculus representation and Hoare Logic triplets are required. That information has been included here for the reader's convenience.

Pi-Calculus expressions:

$$x_0 = 1.\overline{a_0}\langle x_0\rangle$$

$$x_1 = 3.\overline{a_1}\langle x_1\rangle$$

$$a_0(x_2).a_1(x_3).x_4 = \{(a, x_2), (b, x_1)\}.\overline{a_2}\langle x_4\rangle$$

$$a_2(x_5).x_6 = Compute(x_5[a] + x_5[b]).\overline{a_3}\langle x_6\rangle$$

$$a_3(x_7).VariableTable[sum] = x_7$$

Hoare Logic triplets:

$$true\{\,\}value["a"] == 1 \;\&\&\; value["b"] == 3$$

$$value["a"] == 1 \;\&\&\; value["b"] == 3\{\,\}state.sum == 4$$

The first step of the theorem proving is the analyze the first precondition of the first Hoare triplet to verify veracity. The precondition is "true" which is vacuously true. Next, the command building is started. A tree is constructed to store the different Pi-Calculus expressions. The root node is considered the head, and does not include any expressions. The head is only a technical requirement, as otherwise the tree may not have a single root (as in this example). No events are included in the Hoare triplet, so no attention needs to be payed to event handling. The root node has two children, namely each of the Pi-Calculus expressions that is not gated by an input channel (i.e. the first two expressions). Neither of these nodes writes to the Variable Table. They both write a numerical value to their respective channel. The first postcondition is analyzed and immediately fails, because the channels do not contain the correct data type (a dictionary containing the values “a” and “b” is expected, but a scalar value is in each of the channels).

These two expressions join together with a single child, and send their values to the same node. Since a Join was employed, both of these nodes are now considered part of one path for the sake of the backtracking algorithm. This new node outputs a dictionary, as the postcondition expected. Thus, the first postcondition passes, and the above notes are marked. No backtracking was needed for this search. The next precondition is then checked. Since it is identical to the previous postcondition, it also passes.

Employing similar reasoning, the next two expressions are evaluated, the postcondition is evaluated after each one, and the VIPLE expression evaluation algorithm is employed in conjunction with the Variable Table. After the final Pi-Calculus expression is evaluated, the postcondition will be true, so all Hoare Logic triplets have been verified. Thus, this program is marked as correct. Note that since this analysis is performed according to the reaction rules and other inference rules discussed above, the reason for errors can be clearly stated to the use. The use of a theorem prover enables simple explainability as was mentioned in Chapter 1. The final section in this chapter will discuss that application of these rules and concepts to enable the self-learning of the Hoare Logic triplets.

5.5 Self Learning Hoare Triplets

With the above inference and reaction rules, the theorem prover can be modified to not only prove the conditions, but also generate conditions. Returning to the idea of the dynamically defined command algorithm, the Pi-Calculus expressions can be viewed as a tree. In order to maximize the correctness of the Hoare Logic triplets, each distinct path through the tree can be considered as vital to the program's correctness. Furthermore, overfitting should be avoided. Thus, a general middle ground approach is taken, where the defining factors of a program's behavior are the Variable Table and Join nodes.

To construct the Hoare triplets, the algorithm starts by searching through the tree in an identical fashion to the theorem prover. Each time one of those aforementioned defining conditions is met, a postcondition is defined (and the following precondition). Events are also handled in an identical fashion, where each event is considered a unique branch of the tree. As events enable different ordering of the code, the results of an event path may change depending on ordering of the events. To resolve this issue, the number of possible paths is permuted in every possible way. The Pi-Calculus expression tree is analyzed in every ordering of every permutation. The ordering that produces a set containing every other uniquely met state is considered the testing order. In other words, that ordering defines the Hoare triplets that will be learned.

Below are several examples demonstrating the automated conversion of a VIPLE program to Pi-Calculus expressions and the self-learning of the Hoare Logic triplets. Note that in each case, the learning algorithm provides reasoning for each of the rules that it learns. In this way, not only the theorem prover but also the self-learning algorithm are both explainable AI. Figures 5.4, 5.5, and 5.6 are an example of a Merge-based looping counter. In this example, a counter variable is initialized to 0, incremented, and printed out. The Merge activity in this example acts as a label would in a standard programming language, enabling control to return to that point. By employing this Merge followed by an If, a custom looping structure has been created without using the given While activity. After the counter reaches 10, the execution no longer returns to the Merge, thereby ending the loop.

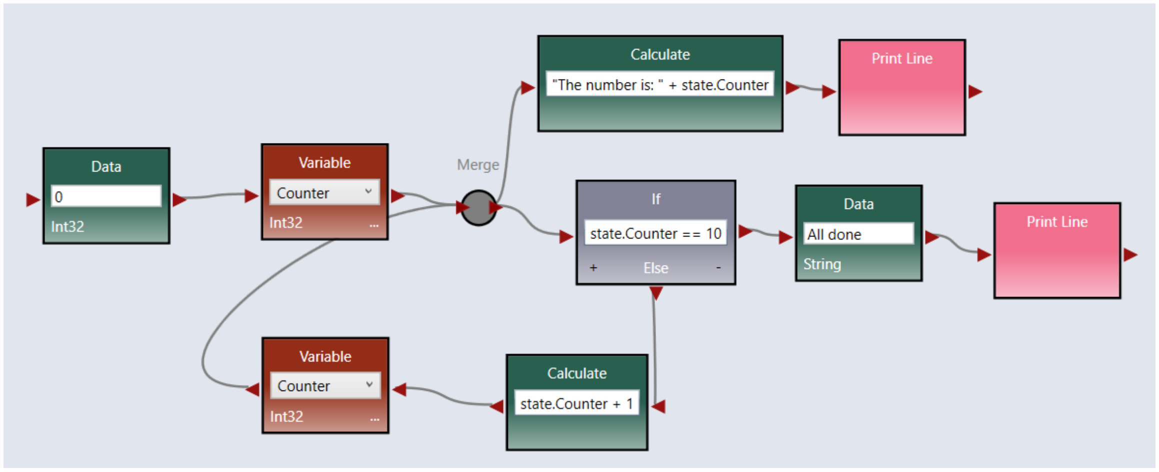
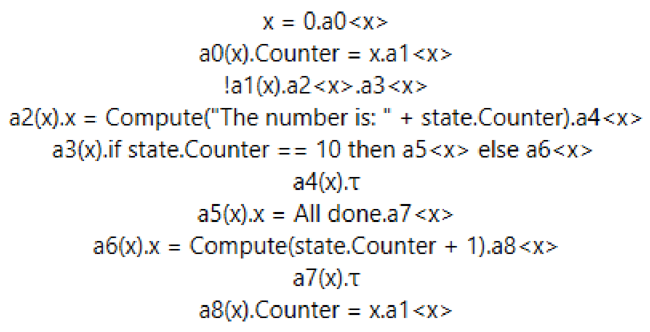
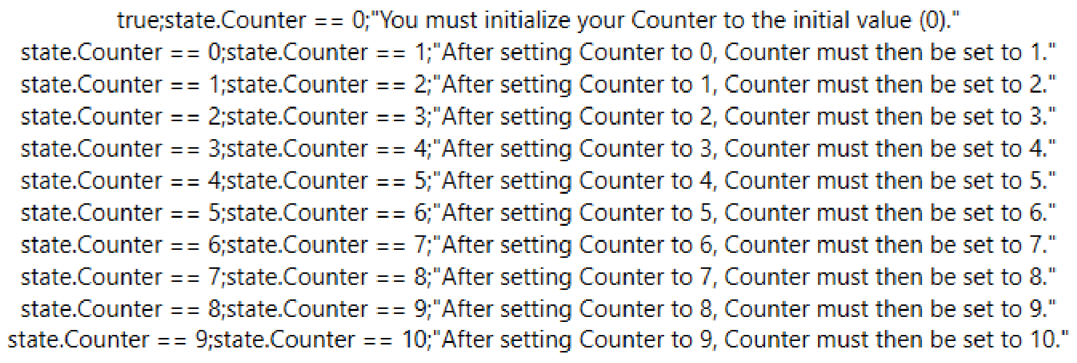


Figure 5.4 Merge Loop Counter



```
x = 0.a0<x>
a0(x).Counter = x.a1<x>
!a1(x).a2<x>.a3<x>
a2(x).x = Compute("The number is: " + state.Counter).a4<x>
a3(x).if state.Counter == 10 then a5<x> else a6<x>
a4(x).τ
a5(x).x = All done.a7<x>
a6(x).x = Compute(state.Counter + 1).a8<x>
a7(x).τ
a8(x).Counter = x.a1<x>
```

Figure 5.5 Merge Loop Counter Pi-Calculus



```
true;state.Counter == 0;"You must initialize your Counter to the initial value (0)."
state.Counter == 0;state.Counter == 1;"After setting Counter to 0, Counter must then be set to 1."
state.Counter == 1;state.Counter == 2;"After setting Counter to 1, Counter must then be set to 2."
state.Counter == 2;state.Counter == 3;"After setting Counter to 2, Counter must then be set to 3."
state.Counter == 3;state.Counter == 4;"After setting Counter to 3, Counter must then be set to 4."
state.Counter == 4;state.Counter == 5;"After setting Counter to 4, Counter must then be set to 5."
state.Counter == 5;state.Counter == 6;"After setting Counter to 5, Counter must then be set to 6."
state.Counter == 6;state.Counter == 7;"After setting Counter to 6, Counter must then be set to 7."
state.Counter == 7;state.Counter == 8;"After setting Counter to 7, Counter must then be set to 8."
state.Counter == 8;state.Counter == 9;"After setting Counter to 8, Counter must then be set to 9."
state.Counter == 9;state.Counter == 10;"After setting Counter to 9, Counter must then be set to 10."
```

Figure 5.6 Merge Loop Counter Learned Hoare Triplets

Figures 5.7 and 5.8 are a While-based looping counter that implements the same requirements from the previous example. Namely, this While loop program creates a counter variable and loops until the counter is 10, printing the value each time. Despite the semantic differences, which are also visible in the Pi-Calculus expressions generated for this code, the previously created Hoare triplets can effectively verify this program. This example demonstrates that since the theorem prover analyzes code at a semantic level, rather than syntactic level, different submissions that meet the requirements will still be marked as correct. Furthermore, if a submission is tested that skips a specific value, the theorem prover will detect this issue and explain to the user that the value was skipped.

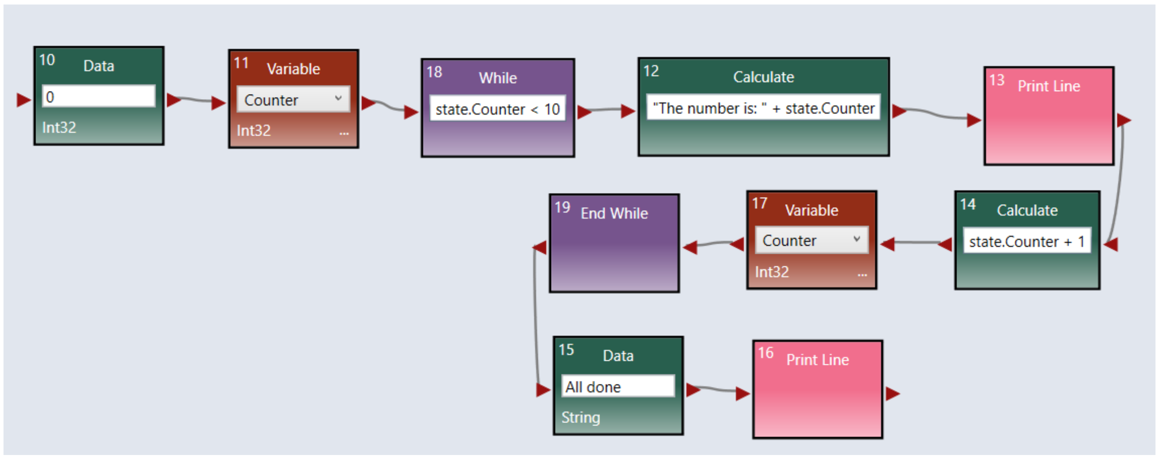


Figure 5.7 While Loop Counter

```
x = 0.a0<x>
a0(x).Counter = x.a1<x>
a1(x).if state.Counter < 10 then a2<x>
!a2(x).a3<x>
a3(x).x = Compute("The number is: " + state.Counter).a4<x>
a4(x).τ.a5<x>
a5(x).x = Compute(state.Counter + 1).a6<x>
a6(x).Counter = x.a7<x>
a7(x).if state.Counter < 10 then a2<x> else a8<x>
a8(x).x = All done.a9<x>
a9(x).τ
```

Figure 5.8 While Loop Counter Pi-Calculus

Figures 5.9, 5.10, and 5.11 are a simple dog finite state machine and its Pi-Calculus and Hoare triplets. This example models the behavior of a dog who may be pet (p) or may see a squirrel (s). If the dog sees a squirrel, it will bark no matter what state it is currently in. If the dog is pet while it is barking, it will shake. If it is pet while it is shaking, it will sit. This simple finite state machine example demonstrates the analysis of programs containing events. As can be seen from the Hoare triplets generated, the learning algorithm finds an ordering of events that causes every state to be reached. Note that since there are only two events, the search depth (i.e. the resulting number from the permutations) is restricted to the default value of two (i.e. the number of events).

However, this search depth can be increased to enable larger, more thorough test cases.

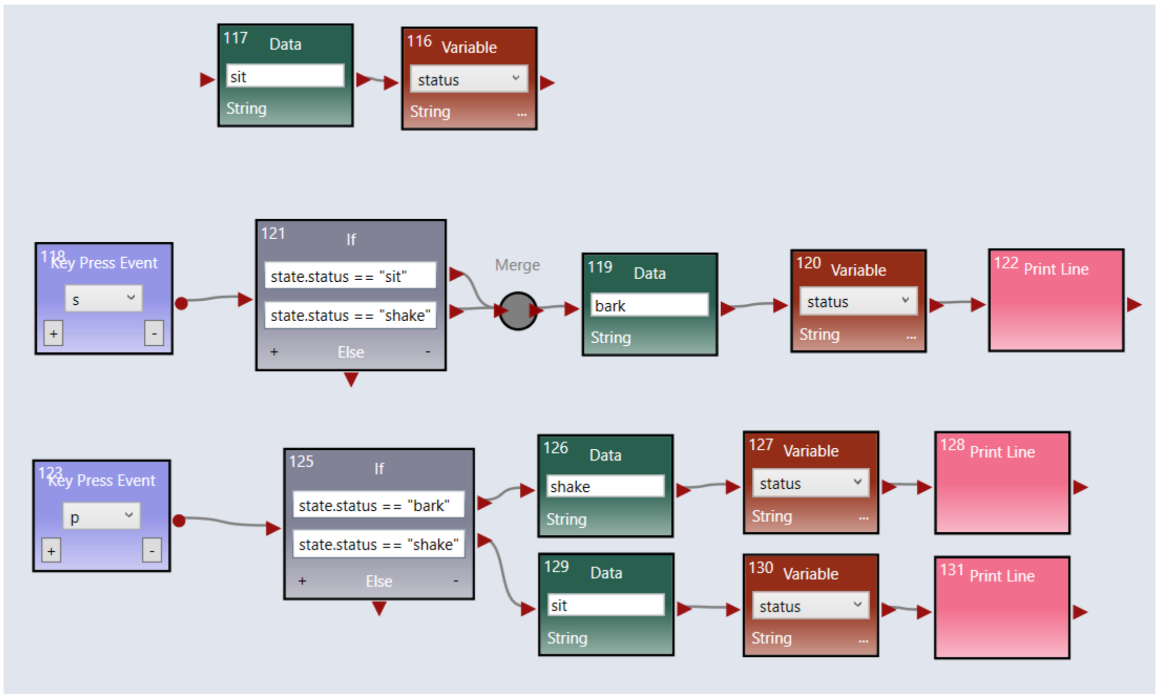
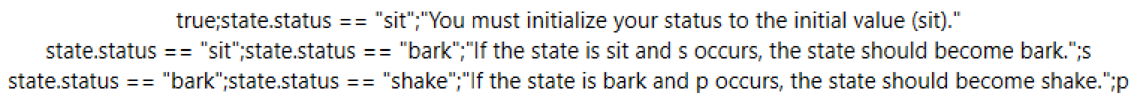


Figure 5.9 Dog Finite State Machine

```
x = sit.a0<x>
!KeyPress(S).a1<x>
!KeyPress(P).a2<x>
a0(x).status = x
a1(x).if state.status == "sit" then a3<x> else if state.status == "shake" then a3<x>
a2(x).if state.status == "bark" then a4<x> else if state.status == "shake" then a5<x>
!a3(x).a6<x>
a4(x).x = shake.a7<x>
a5(x).x = sit.a8<x>
a6(x).x = bark.a9<x>
a7(x).status = x.a10<x>
a8(x).status = x.a11<x>
a9(x).status = x.a12<x>
a10(x).τ
a11(x).τ
a12(x).τ
```

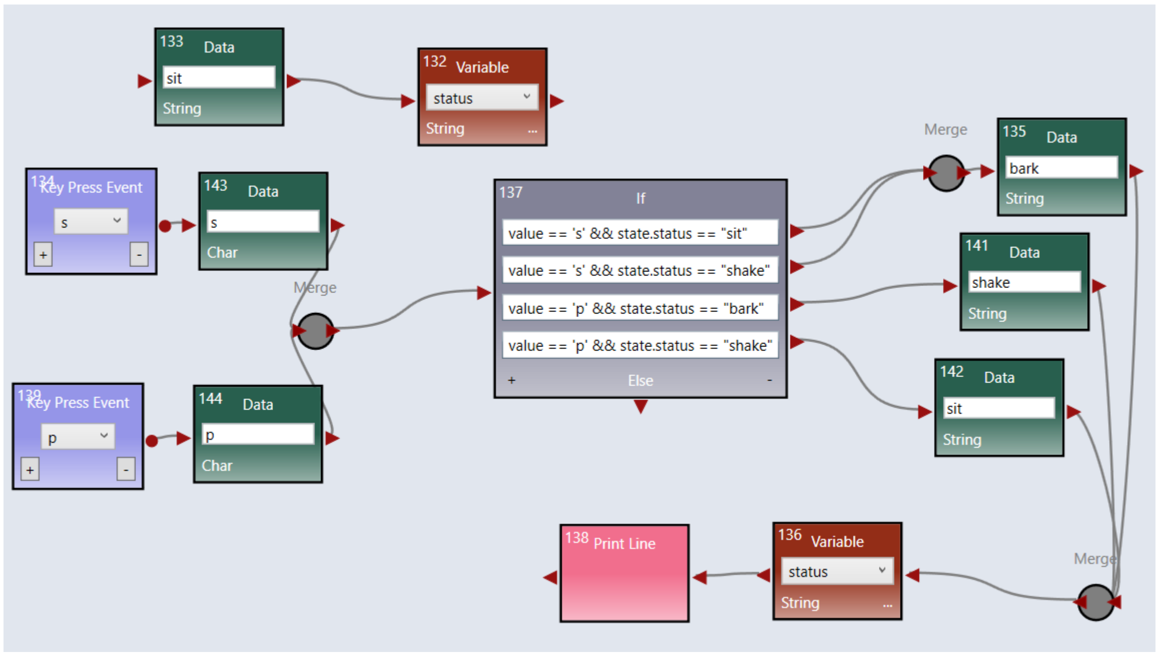
Figure 5.10 Dog Finite State Machine Pi-Calculus



```
true;state.status == "sit";"You must initialize your status to the initial value (sit)."
state.status == "sit";state.status == "bark";"If the state is sit and s occurs, the state should become bark.";s
state.status == "bark";state.status == "shake";"If the state is bark and p occurs, the state should become shake.";p
```

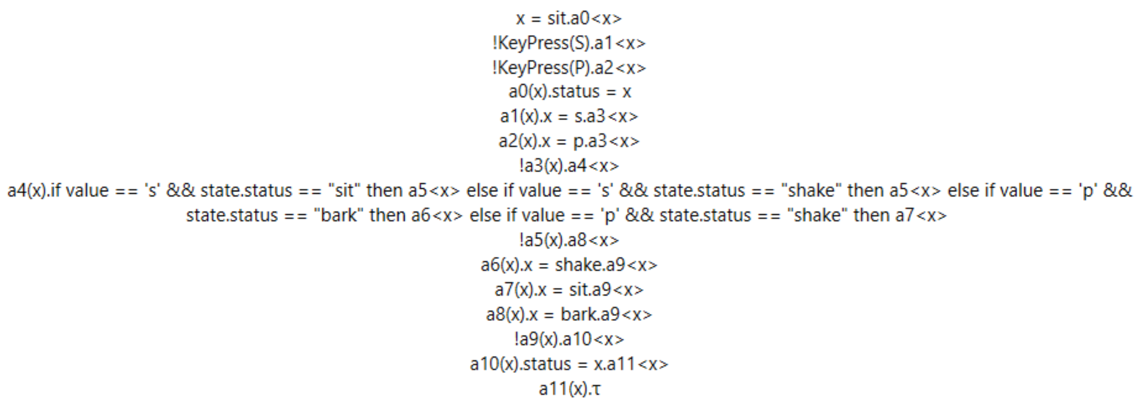
Figure 5.11 Dog Finite State Machine Learned Hoare Triplets

The layout of the program is changed, as shown in Figures 5.12 and 5.13, to employ a single If activity and analyze the events together rather than separately. The Pi-Calculus expressions generated for this program also vary from the previous version. Despite these changes, note that the behavior is identical to the previous example. As such, the previously generated Hoare triplets result in the correct verification of this program.



```
value == 's' && state.status == "sit"
value == 's' && state.status == "shake"
value == 'p' && state.status == "bark"
value == 'p' && state.status == "shake"
```

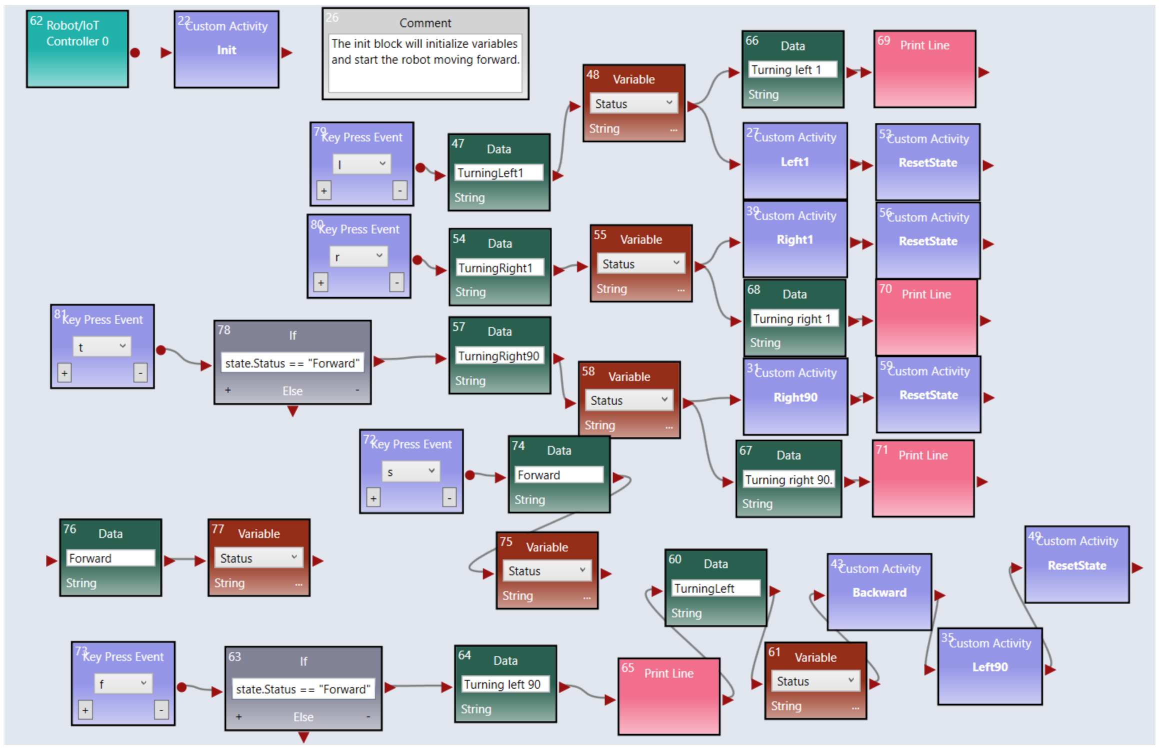
Figure 5.12 Updated Dog Finite State Machine



```
x = sit.a0<x>
!KeyPress(S).a1<x>
!KeyPress(P).a2<x>
a0(x).status = x
a1(x).x = s.a3<x>
a2(x).x = p.a3<x>
!a3(x).a4<x>
a4(x).if value == 's' && state.status == "sit" then a5<x> else if value == 's' && state.status == "shake" then a5<x> else if value == 'p' &&
state.status == "bark" then a6<x> else if value == 'p' && state.status == "shake" then a7<x>
!a5(x).a8<x>
a6(x).x = shake.a9<x>
a7(x).x = sit.a9<x>
a8(x).x = bark.a9<x>
!a9(x).a10<x>
a10(x).status = x.a11<x>
a11(x).τ
```

Figure 5.13 Updated Dog Finite State Machine Pi-Calculus

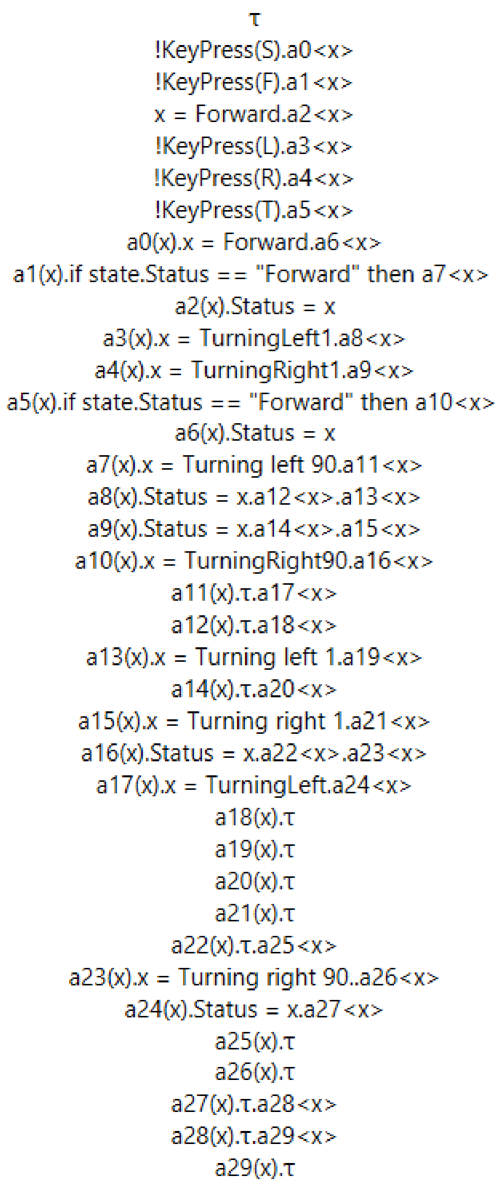
Figures 5.14, 5.15, and 5.16 demonstrate the potential of this self-learning algorithm. The shown code is a right wall following maze navigation algorithm with self-adjusting code to account for imperfections in physical robots. Despite the complexity of the code, the algorithm is able to learn a set of simple Hoare triplets that correctly identify the expected behavior of the program. Note that in all of these examples, the code used for learning is made as a sort of solution key. This key is assumed to be made by the instructor and is a bare bones solution to each problem. Furthermore, although this algorithm provides a set of Hoare triplets, the results can easily be added to or otherwise modified in order to more precisely define requirements or restrictions for students.


Comment

The init block will initialize variables and start the robot moving forward.

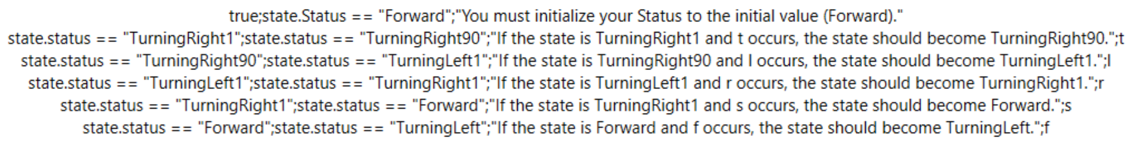
62 Robot/IoT Controller 0
22 Custom Activity Init
26
48 Variable Status String
66 Data Turning left 1 String
69 Print Line
27 Custom Activity Left1
53 Custom Activity ResetState
79 Key Press Event l + -
47 Data TurningLeft1 String
80 Key Press Event r + -
54 Data TurningRight1 String
55 Variable Status String
39 Custom Activity Right1
56 Custom Activity ResetState
68 Data Turning right 1 String
70 Print Line
81 Key Press Event t + -
78 If state.Status == "Forward" + Else -
57 Data TurningRight90 String
58 Variable Status String
31 Custom Activity Right90
59 Custom Activity ResetState
72 Key Press Event s + -
74 Data Forward String
67 Data Turning right 90. String
71 Print Line
76 Data Forward String
77 Variable Status String
75 Variable Status String
60 Data TurningLeft String
43 Custom Activity Backward
49 Custom Activity ResetState
73 Key Press Event f + -
63 If state.Status == "Forward" + Else -
64 Data Turning left 90 String
65 Print Line
61 Variable Status String
35 Custom Activity Left90

Figure 5.14 Self-Adjusting Right Wall Following Algorithm



```
τ
!KeyPress(S).a0<x>
!KeyPress(F).a1<x>
x = Forward.a2<x>
!KeyPress(L).a3<x>
!KeyPress(R).a4<x>
!KeyPress(T).a5<x>
a0(x).x = Forward.a6<x>
a1(x).if state.Status == "Forward" then a7<x>
a2(x).Status = x
a3(x).x = TurningLeft1.a8<x>
a4(x).x = TurningRight1.a9<x>
a5(x).if state.Status == "Forward" then a10<x>
a6(x).Status = x
a7(x).x = Turning left 90.a11<x>
a8(x).Status = x.a12<x>.a13<x>
a9(x).Status = x.a14<x>.a15<x>
a10(x).x = TurningRight90.a16<x>
a11(x).τ.a17<x>
a12(x).τ.a18<x>
a13(x).x = Turning left 1.a19<x>
a14(x).τ.a20<x>
a15(x).x = Turning right 1.a21<x>
a16(x).Status = x.a22<x>.a23<x>
a17(x).x = TurningLeft.a24<x>
a18(x).τ
a19(x).τ
a20(x).τ
a21(x).τ
a22(x).τ.a25<x>
a23(x).x = Turning right 90..a26<x>
a24(x).Status = x.a27<x>
a25(x).τ
a26(x).τ
a27(x).τ.a28<x>
a28(x).τ.a29<x>
a29(x).τ
```

Figure 5.15 Maze Navigation Pi-Calculus



```
true;state.Status == "Forward";"You must initialize your Status to the initial value (Forward)."
state.status == "TurningRight1";state.status == "TurningRight90";"If the state is TurningRight1 and t occurs, the state should become TurningRight90.";t
state.status == "TurningRight90";state.status == "TurningLeft1";"If the state is TurningRight90 and l occurs, the state should become TurningLeft1.";l
state.status == "TurningLeft1";state.status == "TurningRight1";"If the state is TurningLeft1 and r occurs, the state should become TurningRight1.";r
state.status == "TurningRight1";state.status == "Forward";"If the state is TurningRight1 and s occurs, the state should become Forward.";s
state.status == "Forward";state.status == "TurningLeft";"If the state is Forward and f occurs, the state should become TurningLeft.";f
```

Figure 5.16 Maze Navigation Learned Hoare Triplets

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

This paper followed the creation of the automated semantic verifier for VIPLE applications. By leveraging the techniques for developing a standard theorem prover, an AI-based autograder was created. The application of Pi-Calculus allowed for an initial definition of VIPLE programs that could be automatically generated. By modifying this Pi-Calculus representation, the unobservable actions, τ, could be made observable in the context of a VIPLE program. By instituting the use of the Variable Table, the complete VIPLE program behavior could be fully modelled.

The application of Hoare Logic enabled the definition of conditions for specifying the expected behavior of the VIPLE program. These Hoare triplets defined the precondition, command, and postcondition for the behavior of the program. As the commands are defined dynamically by the autograder, the conditions are the main focus of the Hoare triples. Using VIPLE expressions, which have no side effects, the expected behavior of the VIPLE program can be defined.

By employing the inference rules of Pi-Calculus and Hoare Logic, a conversion from VIPLE expressions to the modified Pi-Calculus expressions can be performed. By performing this conversion, the Hoare triplets could be directly employed in the theorem prover, despite the difference in description languages. Combining the ontology, Hoare triplet conditions, and inference and reduction rules from both Pi-Calculus and Hoare Logic, a given VIPLE program could be analyzed semantically. Furthermore, the well-defined nature of this theorem prover enabled a high degree of explainability, namely the ability of the AI to explain why a given program passed or failed verification.

Building on these concepts, the theorem prover was improved to support self-learning of conditions. By giving the self-learning program a sample solution, the required conditions are learned and transcribed as Hoare triplets. Furthermore, this learning algorithm provides details for each of the generated conditions, explaining why that condition was chosen. As a result, unexpected behavior in the condition generation can be directly linked to the given input program.

This self-learning program is the main future work offered by this project. In the current version, learning is very limited. It inputs a single solution program and outputs a single set of test cases. Furthermore, the use of events results in the self-learning program defaulting to permutations of length n, where n is the number of distinct events in the program. Future work may address both of these problems. The event issue may need a more detailed analysis of the program to determine the appropriate depth of permutation. Rather than a single solution, multiple different solutions can be employed. In this way, the self-learning program will learn more precisely the expected behavior of the VIPLE programs, rather than overfitting according to a single program. Research can also be performed to determine if multiple test cases would be beneficial, especially in cases with event-driven programs where ordering matters. Choosing a different first event, for example, may result in an irreversible state change that cannot be thoroughly verified through a single set of Hoare triplets.

Another area of future research is the introduction of support for other programming languages. The original research is performed on VIPLE which is a visual workflow language. However, the concepts developed in this work can be applied to other languages as well. The main components required for support of another programming language are variable introspection, reversible execution (required to determine the output of a code segment for the backtracking algorithm mentioned above), and single step execution to support the Pi-Calculus inference rules.

Although the current methodology employs inductive machine learning, future research could employ deductive machine learning. To this end, verification of the generated rules would also become a significantly more important research topic. By combining these research topics together, the generated ruleset could more accurately represent the expectations of the instructor without requiring a strong knowledge of the underlying framework.

REFERENCES

[1] Doshi-Velez, Finale, and Been Kim. "Towards A Rigorous Science of Interpretable Machine Learning." (2017). Web.

[2] Barack Obama, “Computer Science for All”, President Obama 2016 State of the Union Address, January 30, 2016. https://obamawhitehouse.archives.gov/blog/2016/01/30/computer-science-all

[3] Yinong Chen, Gennaro De Luca: "VIPLE: Visual IoT/Robotics Programming Language Environment for Computer Science Education", IPDPS Workshops 2016: 963-971.

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

[5] How a Columbia University Faculty uses Juypter for Coursework and Lectures https://www.vocareum.com/2019/02/08/columbia/.

[6] Gennaro De Luca and Yinong Chen, “Visual IoT/Robotics Programming Language in Pi-Calculus”, 2017 IEEE Thirteenth International Symposium on Autonomous Decentralized Systems: pp. 23-30, 2017.

[7] Gennaro De Luca and Yinong Chen, “Semantic Analysis of Concurrent Computing in Decentralized IoT and Robotics Applications,” 2019 IEEE Fourteenth International Symposium on Autonomous Decentralized Systems, Utretch, the Netherlands, pp. 95-102, March 2019.

[8] IoT, http://en.wikipedia.org/wiki/Internet_of_Things

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] Autonomous Decentralized Systems (ads): https://en.wikipedia.org/wiki/Autonomous_decentralized_system

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

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

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

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

[20] Yinong Chen and Zhizheng Zhou, “Robot as a Service in Computing Curriculum”, the 12th International Symposium on Autonomous Decentralized Systems, Taichung, March 2015, pp. 156-161.

[21] Aceto, L., Larsen, K., & Ingolfsdottir, A. An Introduction to Milner's CCS [PDF document]. Retrieved from http://people.cs.ksu.edu/~schmidt/705a/Lectures/intro2ccs.pdf. Web.

[22] Milner, R. Communicating and Mobile Systems: The π-calculus. Cambridge: Cambridge UP, 1999. Print.

[23] Puhlmann, F., & Weske, M. Using the π-calculus for Formalizing Workflow Patterns [PDF document]. Retrieved from https://bpt.hpi.uni-potsdam.de/pub/Public/FrankPuhlmann/bpm2005-pic.pdf

[24] Yinong Chen, Service-Oriented Computing and System Integration: Software, IoT, Big Data, and AI as Services, 6th edition, Kendall Hunt Publishing, 2018.

[25] Mike Gordon, “Background reading on *Hoare Logic*”, 2016. Retrieved from https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf

[26] The Halting Problem: https://en.wikipedia.org/wiki/Halting_problem

[27] CBS News, “ASU students build robotic seeing-eye dog”, retrived on Feburary 14, 2019, from the site: http://www.azfamily.com/story/38904179/asu-students-build-robotic-seeing-eye-dog?autostart=true

[28] Russell, Stuart J., and Peter Norvig. Artificial Intelligence: A Modern Approach. 3rd ed., Pearson, 2009.

[29] Hoare, C.A.R., “An axiomatic basis for computer programming”, Communications of the ACM, 12, pp. 576-583, October 1969.

APPENDIX A

PERMISSION TO USE CO-AUTHORED PAPERS

Chapters 3 and 4 employ previously published papers that were co-authored with Dr. Yinong Chen. I am the first author of these papers. He has given me permission to use these articles within this dissertation.