Journal of Artificial Intelligence and Technology, 2021, 1, 21-27
https://doi.org/10.37965/jait.2020.0023

ISTP
RESEARCH ARTICLE

Explainable Artificial Intelligence for Workflow Verification in Visual IoT/Robotics Programming Language Environment

Gennaro De Luca and Yinong Chen
Arizona State University, Tempe, AZ 85281 USA

(Received 09 June 2020; Revised 14 October 2020; Accepted 27 October 2020; Published online 28 December 2020)

Abstract: Teaching students the concepts behind computational thinking is a difficult task, often gated by the inherent difficulty of programming languages. In the classroom, teaching assistants may be required to interact with students to help them learn the material. Time spent in grading and offering feedback on assignments removes from this time to help students directly. As such, we offer a framework for developing an explainable artificial intelligence that performs automated analysis of student code while offering feedback and partial credit. The creation of this system is dependent on three core components. Those components are a knowledge base, a set of conditions to be analyzed, and a formal set of inference rules. In this paper, we develop such a system for our own language by employing π-calculus and Hoare logic. Our detailed system can also perform self-learning of rules. Given solution files, the system is able to extract the important aspects of the program and develop feedback that explicitly details the errors students make when they veer away from these aspects. The level of detail and expected precision can be easily modified through parameter tuning and variety in sample solutions.

Key words: explainable AI; π-calculus; VIPLE; education

I. INTRODUCTION

Software development is a complex field with intricacies in every aspect of development. These intricacies range from simple choices such as language choice to more complex problems such as verification of program correctness. As hardware has become more powerful with multicore machines and cloud computing platforms, many of these complexities have become more difficult to handle. To test a sequential, single-threaded program, more straightforward methods could often be taken, such as a list of behaviors and formal requirements. With multithreaded applications, multiple threads need to be verified not only for behavior but also for communication and coordination [1]. Furthermore, multithreaded applications may not always produce the same results due to race conditions and lack of synchronization. Verifying these programs further adds to the difficulty of program verification.

As the difficulty of software development has increased, the difficulty of teaching software development has similarly increased. Although imperative programming is as straightforward to teach as it was in the past, other paradigms (e.g., service-oriented) and other features such as parallelism are more complex to teach.

As the difficulty of teaching increases, so too does the difficulty of grading student work and offering feedback. However, the time of skilled teaching assistants (TAs) is often better spent interacting with and helping students, rather than grading papers and projects. The main goal of our research is to develop an explainable artificial intelligence (AI) that can solve this problem by verifying student code and offering feedback (i.e., explaining its results). Such a solution would reduce the burden of TAs, enabling them to interact more with students and spend less time (or no time) grading.

To resolve this issue in part, we developed the Visual IoT/Robotics Programming Language Environment (VIPLE) for teaching computational concepts. VIPLE enables us to focus on computational thinking, the algorithmic thought process required for students to program. By reducing the difficulty of programming, students are enabled to learn how to program at a higher, more general level. As they move on to more complex languages or concepts, they will have the background required to succeed [2]. VIPLE overcomes the increasing difficulty of programming by handling that complexity behind the scenes, thereby allowing instructors to completely mask or slowly introduce these complexities on their own terms. For example, VIPLE allows students to interface with and control robots. At the simplest level though, students can use a "robot controller" activity to handle communication. With no understanding of how robot communication works, students can immediately jump into robot application development, learning computational thinking in this area rather than focusing on the syntactic complexity.

Our explainable AI solution is based on the concept of a theorem prover and will be able to accomplish the aforementioned tasks, namely the grading of student programs written in VIPLE and automated feedback unique to that student's work.

This paper details the components and their uses in this solution. The main innovation of this research is the ability to extract a set of rules that represent a correct solution from a given solution document. While generating the solution rules, the system will also generate the feedback corresponding to errors with each of the rules. In this way, students will be given feedback on their mistakes (i.e., any deviations from the set of solution rules). This system supports parameter tuning, enabling modification to the level of detail of the solution rules and the feedback. By changing or adding additional solution files, the solution rules can be made more general or more specific.

Corresponding author: Gennaro De Luca (e-mail: gdeluca1@asu.edu).

II. RELATED WORK

Autograding is a valuable aspect of computer science courses. It can be used to provide feedback to students while preserving the time of instructor and TA, thereby facilitating increase in class sizes. Many approaches have been taken to increase the effectiveness of autograders in the classroom, including gamification for student encouragement [3] and the application of testing techniques such as coverage testing to increase accuracy [4]. Various approaches have also been taken to make autograding more accessible, especially in online courses. These approaches include ideas such as relegating autograding to cloud servers [5] and the addition of inline comments to support student learning [6].

While these approaches may have proven efficacy, they fail to address one of the major shortcomings of autograding. By nature of their implementation, standard autograders work by analyzing the output of student programs given various inputs. Some platforms will perform additional analysis of the output. For example, Vocareum allows variance in student output [7]. However, autograders typically do not look at the semantics of the code, focusing only on input and output. To overcome this issue, several novel approaches have been presented, tested, and verified in recent literature.

The techniques that aim to resolve this issue are based on the concept of program synthesis. Program synthesis is concerned with the construction of programs from a set of logic rules. Such platforms employ AI techniques such as machine learning (ML) and Boolean satisfiability problem (SAT) solving to transition from logic rules to program code [8]. One sample platform is Refazer, which allows learning of code transformations. By analyzing code changes as input–output pairs, Refazer can learn how to move from incorrect code to correct code [9]. By applying Refazer, powerful autograders can be developed that can offer student feedback directly into their code, rather than offering simple feedback about their program's input and output.

One such example platform that was developed using Refazer creates two models, one that analyzes student changes while they work and one that analyzes changes offered by the instructor. Using this information, the platforms can directly offer programmatic feedback to the user. The instructor can label the different types of errors with feedbacks, enabling more class-specific feedback [10].

While the approach in that platform was shown to be effective [10], it has several weaknesses. Most notably, that platform does not link the errors to general concepts and objectives of the course. Another platform aims to solve these issues by starting with a concept focused assignment development phase. The instructor then employs the aforementioned coverage testing approach using a test suite. By analyzing the outputs, the various types of errors can be grouped according to the concept, and feedback specific to that concept is offered. This approach was shown to produce a scalable platform that can offer more personalized feedback that targets concepts rather than errors in lines of code [11].

Despite the scalability, personalization of feedback, and program synthesis techniques, these autograders fail to analyze the code the student has. Rather, these approaches analyze either the code they do not have, such as in [10], or the types of errors in the student output according to different test cases [11]. The approach discussed in this paper takes a program verification approach rather than a program synthesis approach in an effort to solve these issues and offers a scalable autograding experience with personalized feedback based on what the student wrote, rather than samples from other students. The next section will cover requirements to accomplish this task in VIPLE.

III. THEOREM PROVER PREREQUISITES

There are several prerequisites that must be accomplished before program verification can be performed. The first of these requirements is the development of a mathematical framework for VIPLE. We developed such a framework by creating a modified version of π-calculus. π-calculus was chosen as it offers a simple mathematical framework for representing languages that can be viewed as a network. As a workflow language, VIPLE is well suited to π-calculus for that reason. This novel version of π-calculus supports automated conversion from VIPLE programs to mathematical representation. It includes information on communication and coordination as standard π-calculus would [12]. However, it also includes a foundation for representing behavior of individual activities in VIPLE [13]. As such, we have a full framework with which we can perform reasoning on VIPLE programs.

The other major prerequisite is a framework for performing program analysis with the π-calculus representation. We chose to employ Hoare logic for this task. While there are a variety of complexities introduced by trying to analyze a service-oriented and multithreaded programming language, our analysis demonstrated that the combination of VIPLE and π-calculus enabled thorough analysis of program behavior using Hoare logic [13].

With VIPLE, its modified π-calculus representation, and the Hoare logic foundation, the explainable self-learning theorem prover can be formally defined. The approach we 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 [14]. They are (a) a knowledge base that contains all the information about the program being analyzed, (b) a set of conditions to be analyzed for veracity (i.e., "sentences"), and (c) a formal set of rules that enables inference and reasoning about the knowledge base to verify the conditions.

The remainder of this paper will demonstrate each of these three points to illustrate the formal definition of the theorem prover. We will also discuss how we introduced explainability into our system and how instructors can interact with our system for classroom use. Our goal is not only to create such a system but also to ensure that instructors do not require domain knowledge of program correct, π-calculus, etc., to use our system.

IV. KNOWLEDGE BASE

To understand the knowledge base we use for our theorem prover, we must first think about how π-calculus works. π-calculus is based on the concept of sending and receiving data across channels. It does not concern itself with how each node in the network modifies the data. Rather, it treats each of them as black boxes [15].

As we mentioned above, we developed a modified version of π-calculus where these black boxes are given a representation. We call these actions that are normally represented as black boxes in π-calculus observable actions [13]. This is one of the key changes we made to support VIPLE program verification. Our observable actions in VIPLE do not directly interact with the π-calculus communication or coordination (i.e., they do not directly affect the flow of the execution). Rather, these actions may produce values that can be used to determine branching that is already

defined within the π-calculus. Although the π-calculus does not normally analyze the internal behavior of conditionals, that behavior is occurring regardless. It is not creating or destroying channels, but it does choose which channel is followed in the π-calculus.

Another case where observable actions affect the behavior of the program is in the modification of variable values. To resolve this issue, we will introduce a "variable table." Observable actions will be able to interact with the variable table to send or receive values along existing π-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 π-calculus expressions. By employing these π-calculus expressions and the variable table, a full knowledge base can be automatically generated for any given VIPLE program.

V. VERIFIABLE CONDITIONS

The second core component 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 [13], as we discussed above. These conditions are specified by first defining a set of Hoare triples, P, Q, and R, where P is the precondition, R is the postcondition, and Q is a command [16]. Hoare specifies the application of these three components in verifying aspects of a program in his paper. As explained in [16], "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion." Hoare logic enables the definition of logical statements that are verifiable by our theorem prover. This section will examine each of these three parts of the Hoare triplet and what they look like in the context of this platform.

The first element of the Hoare triplet that we will cover is the command, Q, also known as the "program" in Hoare's paper [16]. 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 π-calculus expressions, the utilization of an event in a Hoare triplet is equivalent to invoking a new π-calculus term. Consider, for example, a program that employs the key press event of "A." Then, it may have a π-calculus term that appears as follows: $!\text{KeyPress}_\text{A}(x_0) \cdot \overline{a_1}x_0$.

This term awaits the receipt of a value along the channel corresponding to A's key press event, namely the event KeyPress_A. Events of any variety may be repeatedly triggered. As such, the formal π-calculus definition reflects this behavior through the replication operator. To test the code gated behind such an operator, a single π-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 paper. Here is the π-calculus expression which performs this task: $x_0 = \text{KeyPressed}_\text{A} \cdot \overline{\text{KeyPress}_\text{A}}\langle x_0 \rangle$.

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 π-calculus expressions until either (1) the postcondition is met or (2) the program terminates. We will formalize this behavior in the later section on π-calculus channel communication and coordination. By employing these dynamic command generation mechanisms, multiple Hoare triples can be analyzed sequentially, enabling a more thorough analysis of a given VIPLE program.

Although there are two remaining aspects of a Hoare triplet, the precondition P and the postcondition R, we will discuss both of them together. Both of these parts are conditions and share syntax. As discussed in the paper by Hoare, the main method of evaluating a program is an assignment condition, $x := f$, where x is a variable identifier and f is an expression from the programming language without side effects [16]. Using this condition, Hoare offers the axiom of assignment which forms the foundation for evaluation of multiple conditions. Namely, $\vdash P_0\{x := f\}P$, where P_0 is obtained by substituting f in place of x in P [16]. Building on this rule, more specific rules can be developed, including rules of consequence, composition, and iteration [16]. 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 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 $\text{true}\{x := 0.5 \times f \times 2\}x := f$ 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 triples. 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: $\text{true}\{x := 2 \times f^{-1} \times k\}$ $x := 27$. 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 [16]. This inference rule is vital in cases with multiple Hoare triples. 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 subcondition is verified and that the subconditions 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.

The final prerequisite of the theorem prover is a formal set of rules that enables inference and reasoning about the knowledge base to verify the conditions. This set of inference rules will enable the theorem prover to reason about the given knowledge base (the π-calculus representation) to verify each of the sentences (the Hoare logic triples). There are two components required for the theorem prover to be able to analyze the π-calculus expressions and verify the Hoare logic triples. First, it must be able to reason about the π-calculus expressions' communication and coordination. Second, it must be able to evaluate the Hoare logic triples using the π-calculus expressions and variable table.

VI. FORMALIZING π-CALCULUS INFERENCE RULES

To accurately represent VIPLE programs, we have employed a specially modified variant of π-calculus unique to VIPLE. To reason about the program, the inference rules need to be examined for correctness despite those modifications. The key aspect of analyzing the π-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 [15]. There are several vital reaction rules that are defined by Milner for π-calculus. Those rules are TAU, the unobservable action rule; REACT, the reaction rule; PAR, the parallelism rule; and RES, the reduction semantics rule.

These rules succinctly define several important reaction concepts that are vital to the analysis of the π-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 π-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 following π-calculus expression: $(P \cdot x(y) \cdot \mathbf{0} + M) | (\bar{x}\langle z\rangle \cdot Q + N)$.

From the PAR rule, P can execute in parallel with 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: $(\mathbf{0})|(\bar{x}\langle z\rangle \cdot Q + N)$. The nil activity further reduces as follows: $(\bar{x}\langle z\rangle \cdot Q + N)$. 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 π-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 π-calculus channels, they do not affect the π-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) $\cdot P + M \rightarrow P$. Identical rules can be constructed for variable assignment and data values.

As mentioned above, the variable table is completely independent from the π-calculus channels, so interfacing with the table is also an acceptable action that does not affect the reaction rules for π-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 π-calculus. As such, the polyadic π-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. To support the polyadic π-calculus, the reaction rule must be updated to support vectors, as explained by Milner [15]. The updated REACT rule is as follows (note that the vectors $\vec{z}$ and $\vec{y}$ must have the same length): REACT: $(x(\vec{y}) \cdot P + M)|(\bar{x}\langle\vec{z}\rangle \cdot Q + N) \rightarrow \{\vec{z}$ replaced by $\vec{y}\}P|Q$.

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

VII. EVALUATING HOARE LOGIC WITH π-CALCULUS

By employing those reaction rules, the VIPLE program's π-calculus representation can be analyzed for specific communication or coordination behaviors. However, the conditions to be analyzed are written as Hoare logic triples, and therefore use the VIPLE syntax. We considered two solutions for resolving this issue. First, the Hoare triples could be written in the modified π-calculus syntax. However, π-calculus expressions are restrictive in terms of specifying the behavior of a VIPLE program. By introducing new π-calculus expressions, the behavior of the system changes since communication will be sent to these new terms. One of the rules of Hoare triples is that expressions must not have side effects, but π-calculus expressions would break that rule. Furthermore, we wanted a solution that could be feasibly employed by instructors without requiring an understanding of π-calculus. VIPLE syntax, on the other hand, is much simpler and lacks side effects. The solution we chose to pursue is an evaluation of the Hoare triplet conditions by analyzing the effects of the π-calculus expressions.

There are three components required to analyze Hoare triples according to the effects of the π-calculus expressions. They are (a) the translation of state variables to the variable table, (b) the analysis of values being sent along channels, and (c) the evaluation of VIPLE expressions. Furthermore, the analysis of Hoare logic triples is dependent on the concept of dynamic command definition, as mentioned above. To dynamically define commands, π-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, we employed a modified backtracking approach. Essentially, a tree is constructed based on the π-calculus expressions for the VIPLE program. A breadth-first analysis is conducted. All π-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) can be easily resolved by employing the variable table, which is already independent of the π-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 π-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 π-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 resolved 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 π-calculus expressions.

VIII. SELF-LEARNING HOARE TRIPLES

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 π-calculus expressions can be viewed as a tree. To maximize the correctness of the Hoare logic triples, 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 triples, 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 π-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 triples that will be learned.

IX. CASE STUDY

The semantic autograder can be used for problems of varying difficultly, pursuant to the needs of the class. This case study will examine the application of the autograder to a lab where students

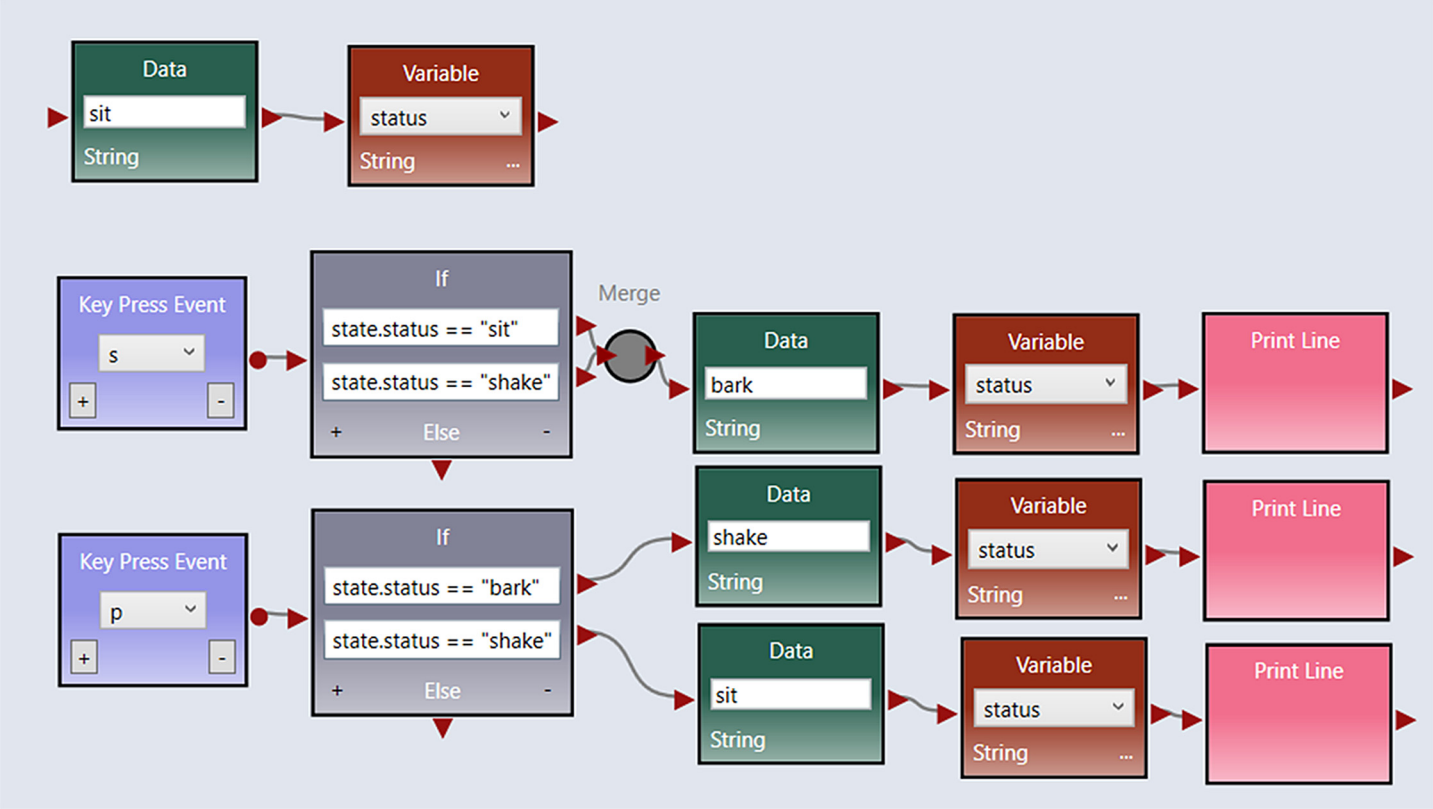

Fig. 1. Dog example.

```
true;~state.status != null && state.status.GetType()!=typeof(String);"Your status variable should be a String."
true;state.status == "sit";"You must initialize your status to the initial state (sit)."
state.status == "sit";state.status == "bark";"If the sitting dog sees a squirrel (S), it should bark.";s
state.status == "bark";state.status == "shake";"If the barking dog is pet (P), it should shake.";p
state.status == "shake";state.status == "bark";"If the shaking dog sees a squirrel (S), it should bark.";s
state.status == "bark";state.status == "shake";"";p
state.status == "shake";state.status == "sit";"If the shaking dog is pet (P), it should sit.";p
```

Fig. 2. Dog rules.

are expected to program a robot dog to perform various actions. By default, the dog is expected to be sitting. From there, the dog is pet (p) or sees a squirrel (s). If the dog sees a squirrel, it should bark. If it is pet and is barking, it shakes. If it is pet and is shaking already, it sits. Fig. 1 shows a correct implementation in VIPLE of this lab.

Students may correctly program this application using alternative syntactic structures such as a single if conditional or a switch statement. They may print out different values or print out nothing at all. Unlike conventional autograders, this autograder is not analyzing the output.

For this example, the rules were autogenerated. From there, the feedback was updated to clarify the meaning of the key presses p and s as well as the meaning of the status variable. These modifications are optional but build on the rules offered to produce clearer feedback. The rules are shown in Fig. 2.

The next step of the semantic autograder is to convert the code into the modified π-calculus representation. This representation is shown in Fig. 3.

With the π-calculus representation, the Hoare triples are verified using the automated theorem prover. In this case, the sample code is correct, so no errors are found. However, suppose that the student mistakenly told the dog to shake if it is pet while already shaking. In this case, an error message will be given to the student explaining the error and the expected behavior as shown in Fig. 4.

To achieve this level of accuracy in standard autograders, a complete test suite would need to be used for each program, such that every combination of transitions is tested. In this case, the test suite is autogenerated through the analysis of potential paths

```
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
```

Fig. 3. Dog π-calculus representation.

Error: If the shaking dog is pet (P), it should sit.

Fig. 4. Error message.

through the program. In more complex examples, the number of test cases in standard autograders would increase exponentially. This semantic autograder does not need to test every possible combination since it analyzes the logic of the program itself, rather than the results of various permutations of input. In the error example, it finds that the user program performs the transition to shake dependent on the dog being pet and already shaking. In this way, the specific error can be found and feedback can be given that is directly relevant to the error, unlike standard autograders which may only offer the failed test case to the user with no advice on resolution of the error.

X. CONCLUSION

This paper detailed our research and development of an automated theorem prover for use in VIPLE. By introducing aspects of explainability to our work, we are able to not only verify and grade student programs but also offer feedback and partial credit.

While this project is strongly beneficial for use with VIPLE inside the classroom, there are various other programming languages that other universities employ. These languages range from similar workflow languages to vastly different text-based languages with different programming paradigms. With a strong foundation for developing such an explainable AI, future work could attempt to apply these steps to another language. The main difficulty and novelty of such work would likely result from VIPLE's special construction that facilitated this work, such as side-effect free actions, and a more limited syntax.

Further work could also involve the employment of ML within the context of this research. There are many opportunities to employ ML in this area, such as the analysis of students as they interface with the development environment. Another area could involve analysis of human graders in an effort to design a framework that takes into consideration how the humans work rather than only the mathematical constructs we defined.

ACKNOWLEDGMENT

The research is partly supported by general funding at IoT and Robotics Education Lab and FURI program at Arizona State University.

REFERENCES

[1] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT Num. Math.*, vol. 8, pp. 174–186, 1968, DOI: 10.1007/BF01933419.

[2] Y. Chen and G. De Luca, "VIPLE: Visual IoT/robotics programming language environment for computer science education," in IPDPS Workshops, Chicago, IL, USA, 2016, pp. 963–971.

[3] A. Iosup and D. Epema, "An experience report on using gamification in technical higher education," in Proc. SIGCSE, Atlanta, GA, USA, 2014, pp. 27–32, DOI: 10.1145/2538862.2538899.

[4] D. Jackson and M. Usher, "Grading student programs using ASSYST," *ACM SIGCSE Bull.*, vol. 29, no. 1, Mar. 1997, pp. 335–339, DOI: 10.1145/268084.268210.

[5] D. Milojicic, "Autograding in the Cloud: Interview with David O'Hallaron," *IEEE Internet Comput.*, vol. 15, no. 1, Jan. 2011, pp. 9–12, DOI: 10.1109/MIC.2011.2.

[6] T. MacWilliam and D. J. Malan, "Streamlining grading toward better feedback," in Proc. ITiCSE, Canterbury UK, 2013, pp. 147–152, DOI: 10.1145/2462476.2462506.

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

[8] S. Gulwani, "Dimensions in program synthesis," in Proc. Symp. PPDP, Hagenberg, Austria, 2010, pp. 13–24.

[9] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in Proc. ICSE, Buenos Aires, Argentina, 2017, pp. 404–415.

[10] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann, "Writing reusable code feedback at scale with mixed-initiative program synthesis," in Proc. Conf. Learning @ Scale Cambridge, MA, USA, 2017, pp. 89–98.

[11] G. Haldeman, A. Tjang, M. Babeş-Vroman, S. Bartos, J. Shah, D. Yucht, and T. D. Nguyen, "Providing meaningful feedback for autograding of programming assignments," in Proc. SIGCSE, Baltimore, MD, USA, 2018, pp. 278–283, DOI: 10.1145/3159450.3159502.

[12] G. De Luca and Y. Chen, "Visual IoT/robotics programming language in pi-calculus," in Proc. ISADS, Bangkok, Thailand, pp. 23–30.

[13] G. De Luca and Y. Chen, "Semantic analysis of concurrent computing in decentralized IoT and robotics applications," in Proc. ISADS, Utrecht, the Netherlands, 2019, pp. 95–102.

[14] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed., Upper Saddle River, NJ, USA: Pearson, 2009.

[15] R. Milner, *Communicating and Mobile Systems: The π-Calculus*, Cambridge: Cambridge University Press, 1999.

[16] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–583, Oct. 1969.