Formalizing Model Inference of MicroPython

Carlos Mão de Ferro
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Lisboa, Portugal
0000-0001-6835-3097

Tiago Cogumbreiro
University of Massachusetts Boston
Boston, MA, USA
0000-0002-3209-9258

Francisco Martins
Universidade dos Açores
Ponta Delgada, Portugal
0000-0002-2379-7257

***Abstract*—Model checking has often been used for verifying Cyber-Physical Systems (CPS). A major challenge is how to capture a model that represents the actual behavior of the software. Model extraction can introduce errors that can affect the accuracy of the analysis including loss of precision, inconsistency, non-conformance, and over- and under-approximations.**

In this paper, we formalize and prove the correctness of extracting a model from a subset of the MicroPython programming language with respect to a trace-based semantics. The extracted models capture the order of method calls and can be model checked using Shelley. We formalize the extraction process from an intermediate representation of MicroPython codes and prove that the behavior of our intermediate representation is a regular language. Our formalization and theoretical results are fully mechanized using the Coq proof assistant.

***Index Terms*—model-extraction, model-checking, formalization, micropython**

I. Introduction

Model extraction [1], [2] is the process of automatically generating a model that captures the behavior of a system and its key properties. Model extraction is a balancing act between two approaches: *over-approximation* where the extracted model is too general and includes more behaviors than in the original system, *e.g.,* may trigger alarms that do not appear in practice; *under-approximation* where the extracted model is incomplete and misses certain behaviors, *e.g.,* may miss certain alarms. Further, the extraction process must carefully select a level of detail of a system: adding too much detail makes the model too big and impossible to analyze due to the state-explosion problem. Additionally, inaccuracies and bugs in the extraction process may lead to spurious behaviors that lead to incomplete or incorrect verification results [3].

Model extraction is particularly important in the context of cyber-physical system (CPS) applications, where the correctness and safety of the system are critical [4]–[10]. CPS applications typically involve the integration of physical systems, such as sensors, actuators, and control systems. This paper focuses on *formalizing* the model extraction process of CPS applications written in MicroPython [11] to be used in the context of model checking.

Formalizing model extraction has the following benefits. Firstly, the formalization offers a clear and precise description of the extraction process, which improves the understanding of the capabilities of our model checker. Secondly, having the extraction process formalized allows us to reason about properties of the source system and of the target model. Importantly, we can characterize the *expressiveness* of the extraction.

This paper specifies the model extraction of Shelley [12], a model-checking framework that features linear temporal logic on finite traces (LTL_f) [13], [14]. Similar tools exist for other languages including Java [15]–[17], C++ [18], and Lustre [16], [19] although their focus is usually on concurrency. In contrast, our temporal claims are written in terms of function calls that manipulate physical resources that can be specified with ordering constraints. MicroPython classes that directly manipulate physical resources (called constrained classes/objects) are annotated with ordering constraints, akin to typestates [20]. Further, and without resorting to any more ordering annotations, our analysis verifies the correct usage of constrained objects, and infers the ordering constraints of methods using other constrained objects. Unlike most model checkers, Shelley models do not capture state change of program-variables, nor the communication of concurrent systems [19], [21]–[26].

In summary, our paper makes the following contributions:

1) We formalize the process of extracting a Shelley model from a small imperative language.
2) We prove the *correctness* of the extraction process: a trace is produced by the source program if, and only if, a trace is produced by the behavior of a program. We show that the behavior of a program is a regular language.
3) The formalization and the results of this paper are fully mechanized using the Coq proof assistant and are available in [27].

The rest of the paper is organized as follows. Section II overviews our approach through a series of examples. Section III presents our model inference, formalizes the behavior extraction of a program (which represents the code of a method declaration), and establishes our main result of correctness. Section IV discusses related work. Finally, Section V concludes the paper and considers the next steps in our research.

II. Model checking with Shelley

In this section, we detail how to use the Shelley framework to model check a MicroPython program. Our tool provides a MicroPython Application Programming Interface (API) that

TABLE I: Shelley's annotations, where to apply them in a MicroPython program, and their meanings.

Annotation	Applies to	Meaning
`@claim`	class	temporal requirement
`@sys`	class	base class
`@sys(["`s_1`",...,"`s_n`"])`	class	composite class
`@op_initial`	method	invoke in first place
`@op_final`	method	invoke in last place
`@op_initial_final`	method	invoke in first and last places
`@op`	method	invoke in between an initial and final methods

enables the automatic verification of the order of method calls in an object hierarchy. We use annotations, according to Table I, so that developers can provide more information about the expected behavior of methods and prevent an unintended order of actions from occurring. This way, we enforce the behavior of an object and how it can be used in such a way that we can model check temporal requirements. Shelley includes a visualization tool that automatically generates behavior diagrams based on the code annotations and based on the control flow of the code under analysis.

To make our analysis scalable, Shelley makes two important design decisions: a restricted programming model, and an over-approximated behavior. Regarding the former, Shelley does not support non-terminating (infinite) behaviors, ignores object aliasing, and only considers method invocation of fields. Regarding the latter, our approach is closer to type checking of a behavioral type, than to model check the full behavior of a program. The code of each method must be expressed as a regular expression representing any possible sequence of method calls. Shelley features sequencing, nondeterministic choice, and terminating loops (via the Kleene-star operator). Further, our tool disregards the program's internal state, *e.g.,* the arguments of method calls, the condition used to branch, and the loop bounds.

We now give a brief guide on the annotations that Shelley offers to verify a MicroPython class. Our running example is based on an industrial use case in [28], a battery-operated wireless controller that switches water valves according to a scheduled irrigation plan.

A. Specifying a class behavior

Valves are electromechanical devices that can be programmed to control water flow. Class `Valve` uses general-purpose input-output pins to operate a physical valve, as seen in Listing II.1. Our verification goal is to minimize the chance of clogging the physical valve, so users of the `Valve` must test the status of the valve before opening it. Additionally, to conserve battery, we want to verify that users of `Valve` always test the status of the valve before cleaning up any debris. The class annotation `@sys`, in Line 1, tells Shelley to verify class `Valve`. The method annotations `@op_initial`, `@op`, and `@op_final` tell Shelley which methods to consider in the verification. We list our annotations in Table I. To meet our verification goals, Shelley ensures that any users of `Valve` (*e.g.,* Listing II.2) respect the specified method-ordering.

Listing II.1: Class `Valve`

```
1  @sys
2  class Valve:
3    def __init__(self):
4      self.control = Pin(27, OUT)
5      self.clean = Pin(28, OUT)
6      self.status = Pin(29, IN)
7
8    @op_initial
9    def test(self):
10     if self.status.value():
11       return ["open"]
12     else:
13       return ["clean"]
14
15   @op
16   def open(self):
17     self.control.on()
18     return ["close"]
19
20   @op_final
21   def close(self):
22     self.control.off()
23     return ["test"]
24
25   @op_final
26   def clean(self):
27     self.clean.on()
28     return ["test"]
```

Listing II.2: Class BadSector

```
1  @claim("(!a.open) W b.open"))
2  @sys(["a", "b"])
3  class BadSector:
4    def __init__(self):
5      self.a = Valve()
6      self.b = Valve()
7
8    @op_initial_final
9    def open_a(self):
10     match self.a.test():
11       case ["open"]:
12         self.a.open()
13         return ["open_b"]
14       case ["clean"]:
15         self.a.clean()
16         print("a failed")
17         return []
18
19   @op_final
20   def open_b(self):
21     match self.b.test():
22       case ["open"]:
23         self.b.open()
24         self.a.close()
25         self.b.close()
26         return []
27       case ["clean"]:
28         self.b.clean()
29         print("b failed")
30         self.a.close()
31         return []
```

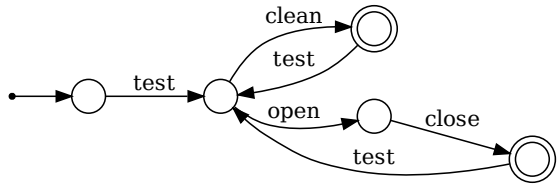

Fig. 1: `Valve` diagram automatically generated by our tool based on the annotations of Listing II.1.

We now explain how to declare the method-ordering shown in Figure 1. The annotation `op_initial` ensures that after creating an instance of `Valve` the only method that can be invoked is `test` (Lines 9 to 13) — multiple methods can be declared as initial. Each annotated method must `return` the set of methods of `Valve` that can be invoked subsequently. For instance, after invoking `test` the return value can be `["open"]`, which means that the user must then invoke method `open`. Alternatively, after invoking `test` the return value can be `["clean"]`, so the user must invoke method `clean`. If a `return` statement allows several calls to follow it, we use the nomenclature `return ["`m_1`", ..., "`m_n`"]` and if no calls shall follow it we return an empty list. Finally, user-return values are also supported by using tuples, where the first position is reserved to specify the next available methods. Some examples are given in Table II.

The decorator `op_final` allows for the declaration of

TABLE II: Examples of return statements and their meanings.

Return statement	Meaning
`return ["close"]`	expecting method ”close” to be invoked next
`return ["open", "clean"]`	expecting methods ”open” or ”clean” to be invoked next
`return ["close"], 2`	expecting operation ”close” to be invoked next and return the integer value 2
`return ["close"], True`	expecting method ”close” to be invoked next and return the boolean value True
`return ["open", "clean"], 2`	expecting methods ”open” or ”clean” to be invoked next and return the integer value 2

“destructor” methods: method `close` must be the last method called, with respect to the object’s lifetime. Importantly, since `open` is *not* marked as final, Shelley guarantees that the valve cannot be left open.

B. Verifying object usage

We now describe how Shelley verifies that class `BadSector` incorrectly uses instances of `Valve`. Further, we introduce the use of temporal claims to model check the usage of verified instances. Shelley only considers the order of calls and disregards any values used in boolean conditions, loop ranges, and being passed in method calls. Shelley supports branching with `if/elif/else` and `match/case` and looping with `for` and `while`.

Class `BadSector` in Listing II.2 handles the opening of valves in two separate methods — in the irrigation jargon, a *sector* is an irrigation zone where several water valves are grouped together. `BadSector` is a composite class since it uses two instances of `Valve`, `a` and `b`. Thus, depending on the method’s annotations of `BadSector`, we might use the two valves in different ways. Shelley identifies the following invalid usage of `Valve`. Since method `open_a` is decorated with `initial` and `final`, this means that a user of `BadSector` could issue `open_a` without ever issuing `open_b`, potentially leaving valve `a` open, something that, according to the `Valve`’s specification, is an incorrect behavior, as depicted in Figure 2. Shelley outputs the following error message:

```
Error in specification: INVALID SUBSYSTEM USAGE
Counter example: open_a, a.test, a.open
Subsystems errors:
    * Valve 'a': test, >open< (not final)
```

Matching exit points In Lines 10 and 21 we intentionally use the `match` statement to distinguish the cases where the method being called has more than one exit point. For instance, as seen before, the `Valve`’s method `test` can be followed by either `open` or `clean`. Therefore, our tool checks if all possible exit points are being handled.

Checking temporal requirements Correctness claims express temporal properties on the code of the class being verified. Temporal claims are of great importance for software maintenance, as Shelley can check if changes to the class preserve the internal behavior being specified. Besides automatically verifying that each valve is being used according to the specification in Listing II.1, Shelley also verifies temporal requirements. The temporal claim in Line 1 of Listing II.2 states that valve `a` must remain closed until valve `b` is opened. The implemented behavior violates the temporal claim, so Shelley outputs the following error message:

```
Error in specification: FAIL TO MEET REQUIREMENT
Formula: (!a.open) W b.open
Counter example: a.test, a.open, b.test, b.open, a.close, b.close
```

The formula $\phi_1 \,\mathsf{W}\, \phi_2 = (\phi_1 \,\mathsf{U}\, \phi_2) \vee \mathsf{G}\, \phi_1$ is interpreted as a *weak until*, meaning that ϕ_1 has to hold at least until ϕ_2 or ϕ_1 must remain true forever.

III. Model inference

In this section we detail the process of extracting a Shelley model from a MicroPython class. The model checking of a Shelley model is outside of the scope of this paper, so here we only focus on defining what constitutes a Shelley model. The model extraction process consists of the following steps:

1) **method dependency extraction:** (Section III-A) our tool captures the dependencies between methods being defined.
2) **method behavior extraction:** (Section III-B) our tool extracts the behavior of each method being defined.
3) **method invocation analysis:** we check if method invocation of objects under analysis are defined in their respective classes; we also check for exhaustive tests on matches that take the result of a method-invocation under analysis.

A. Method dependency extraction

The method-dependency is defined as a directed graph, where the nodes represent the entry point of each method name and every exit point of a method; the arcs are ordering constraints. There is a single entry node per method. For instance, in Listing III.1, we have 4 methods (`open_a`, `clean_a`, `close_a`, and `open_b`), so there are 4 entry nodes. In addition, there is an exit node per return in each method. For example, in Listing III.1, method `open_a` has 2 return statements, thus we have 2 exit nodes: exit node, say (A), represents `return ["close_a", "open_b"]`, and the exit node, say (B), represents `return ["clean_a"]`. We link each entry node of a method to each of its exit nodes. For instance, in Figure 3, the entry node of `open_a` links to nodes (A) and (B). Finally, for each exit node and each method name being returned, we link the exit node to the entry node of the method being returned. That is, since exit node (A) returns `["close_a", "open_b"]`, we link exit node (A) to the entry node of method `close_a`, and exit node (A) to the entry node of method `open_b`.

B. Method behavior extraction

In this section we formalize how our tool infers the behavior of each method, described as a regular expression. *The formalization included in this paper, along with the*

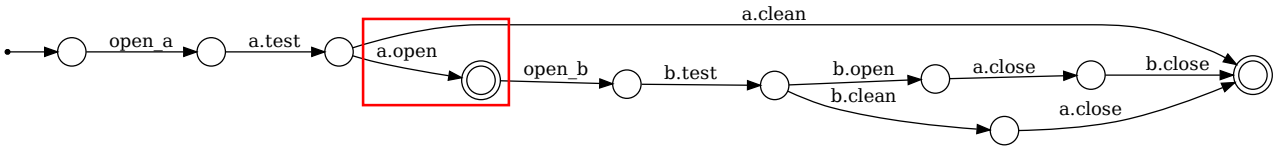

Fig. 2: `BadSector` diagram (invalid usage of valves). After opening valve `a` we reach a possible final state, leaving valve `a` open and not respecting the `Valve` specification.

Listing III.1: Class `Sector` with code elided to only show returns per method.

```
class Sector:
  def open_a(self):
    # ...
    return ["close_a", "open_b"]
    # ...
    return ["clean_a"]

  def clean_a(self):
    return ["open_a"]

  def close_a(self):
    # ...
    return ["open_a"]

  def open_b(self):
    # ...
    return []
    # ...
    return []
```

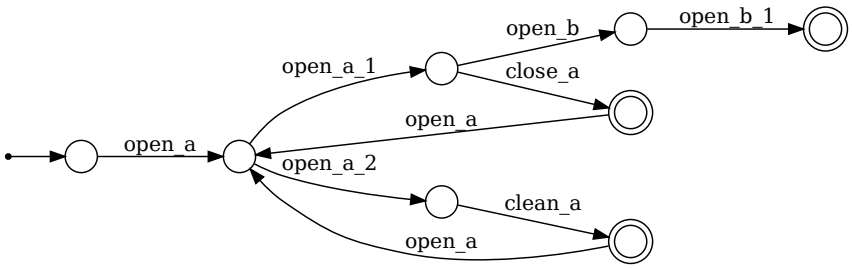

Fig. 3: Shelley model of class `Sector`.

theoretical results presented, are fully mechanized using the Coq proof assistant. The syntax of the source language is an abstraction of MicroPython, that captures the control flow of the program and function calls — our input language ignores the intermediate values being calculated. The syntax of the source language, its semantics, and the behavior inference is given in Figure 4.

Syntax. The syntax of a program p consists of: $f()$ to encode a method call (discarding the arguments); `skip` represents any MicroPython instruction that is of no interest to the analysis; `return` to return a value, where the actual value being returned is ignored at this stage of the analysis; `if`$(\star)$ $\{p\}$ `else` $\{p\}$ represents a non-deterministic choice, as the condition is not represented; `loop`$(\star)$ $\{p\}$ represents a loop that runs p an unknown number of iterations. *Supported Python constructs:* our analysis represents `for` and `while` as a loop, `match` and `if` as a conditional; our analysis does not model Python exceptions.

Semantics. We give semantics to our syntax in terms of the sequences of labels that a program outputs. The judgment $s \vdash l \in p$ denotes that trace l (a sequence of labels f) is output by program p when the program has a certain status s (which is either `O` for ongoing, or `R` for returned). Our intuition behind using the term "returned" is in the sense that no other terms of the trace can be in a returned trace, whereas a "ongoing" trace can be sequenced further. We denote a sequence comma separated between brackets, *e.g.,* a sequence with digits 0, 1, and 2 is denoted by $[0, 1, 2]$. The usual sequence concatenation is denoted by $l_1 \cdot l_2$. Rule CALL states that trace $[f]$ is in program $f()$ when the program is still ongoing (`O`). Rule SKIP states that empty trace $[]$ is in program `skip` when the program is still ongoing (`O`). Rule RETURN states that empty trace $[]$ is in program `skip` when the program has returned (`R`). The rules are that of sequence. Rule SEQ-1 states that if a certain trace l is in p_1 and that trace has returned (due to a return), then program $p_1; p_2$ also outputs trace l. Rule SEQ-2 states what happens if trace l_1 of p_1 is ongoing (`O`), then we can prepend l_1 to any trace l_2 from p_2 in $p_1; p_2$. Rule IF-1 and IF-2 state that if either branch p_1 or branch p_2 output a trace l under status s, then program `if`$(\star)$ $\{p_1\}$ `else` $\{p_2\}$ outputs a trace l under status s. Finally, the loop is governed by three rules. Rule LOOP-1 (akin to Rule SKIP) states that a loop can terminate and in that case we have an empty trace $[]$ and the status is ongoing. Rule LOOP-2 (akin to Rule SEQ-1) captures the case where the loop body p outputs trace l and issues a return. Rule LOOP-3 (akin to Rule SEQ-2) captures the case where the loop body p outputs a trace l_1 and the computation continues by ongoing `loop`$(\star)$ $\{p\}$ with a trace l_2.

Syntax

$$p \ ::= \ f() \ \mid \ \texttt{skip} \ \mid \ \texttt{return} \ \mid \ \texttt{if}(\star)\ \{p\}\ \texttt{else}\ \{p\} \ \mid \ \texttt{loop}(\star)\ \{p\}$$

$$s \ ::= \ \texttt{0} \ \mid \ \texttt{R}$$

Semantics $\boxed{s \vdash l \in p}$

$$\text{CALL}\ \frac{}{\texttt{0} \vdash [f] \in f()} \qquad \text{SKIP}\ \frac{}{\texttt{0} \vdash [] \in \texttt{skip}} \qquad \text{RETURN}\ \frac{}{\texttt{R} \vdash [] \in \texttt{return}}$$

$$\text{SEQ-1}\ \frac{\texttt{R} \vdash l \in p_1}{\texttt{R} \vdash l \in p_1; p_2} \qquad \text{SEQ-2}\ \frac{\texttt{0} \vdash l_1 \in p_1 \qquad s \vdash l_2 \in p_2}{s \vdash l_1 \cdot l_2 \in p_1; p_2}$$

$$\text{IF-1}\ \frac{s \vdash l \in p_1}{s \vdash l \in \texttt{if}(\star)\ \{p_1\}\ \texttt{else}\ \{p_2\}} \qquad \text{IF-2}\ \frac{s \vdash l \in p_2}{s \vdash l \in \texttt{if}(\star)\ \{p_1\}\ \texttt{else}\ \{p_2\}}$$

$$\text{LOOP-1}\ \frac{}{\texttt{0} \vdash [] \in \texttt{loop}(\star)\ \{p\}} \qquad \text{LOOP-2}\ \frac{\texttt{R} \vdash l \in p}{\texttt{R} \vdash l \in \texttt{loop}(\star)\ \{p\}}$$

$$\text{LOOP-3}\ \frac{\texttt{0} \vdash l_1 \in p \qquad s \vdash l_2 \in \texttt{loop}(\star)\ \{p\}}{s \vdash l_1 \cdot l_2 \in \texttt{loop}(\star)\ \{p\}}$$

Behavior inference $\boxed{[\![p]\!] = (r, s)}$ $\boxed{\text{infer}(p) = r}$

$$[\![f()]\!] = (f, \emptyset) \qquad [\![\texttt{skip}]\!] = (\epsilon, \emptyset) \qquad [\![\texttt{return}]\!] = (\emptyset, \{\epsilon\}) \qquad [\![p_1; p_2]\!] = (r_1 \cdot r_2, \{r_1 \cdot r \mid r \in s_2\} \cup s_1)$$

$$[\![\texttt{if}(\star)\ \{p_1\}\ \texttt{else}\ \{p_2\}]\!] = (r_1 + r_2, s_1 \cup s_2) \qquad [\![\texttt{loop}(\star)\ \{p_1\}]\!] = (r_1^\star, \{r_1^\star \cdot r \mid r \in s_1\})$$

$$\text{where } [\![p_1]\!] = (r_1, s_1) \text{ and } [\![p_2]\!] = (r_2, s_2)$$

$$\text{infer}(p) = r_1 + r_1' + \cdots + r_n' \text{ where } [\![p]\!] = (r_1, s_1) \wedge s_1 = \{r_1', \ldots, r_n'\}$$

Fig. 4: Extracting the behavior from imperative code

Example 1. The trace $[a, c, a, c]$ is in the following program, which exercises a trace that yields from a loop that runs 2 iterations without an early return.

$$\texttt{0} \vdash [a, c, a, c] \in$$

$$\texttt{loop}(\star)\ \{a(); \texttt{if}(\star)\ \{b(); \texttt{return}\}\ \texttt{else}\ \{c()\}\}$$

Example 2. The trace $[a, c, a, b]$ is in the same program, which highlights the case where the loop runs for one iteration and in the second iteration returns.

$$\texttt{R} \vdash [a, c, a, b] \in$$

$$\texttt{loop}(\star)\ \{a(); \texttt{if}(\star)\ \{b(); \texttt{return}\}\ \texttt{else}\ \{c()\}\}$$

Definition 1 (Behavior)**.** *Let* $L(p) = \{l \mid s \vdash l \in p\}$ *denote the* behavior *of program* p*.*

Behavior inference. We now introduce the process of extracting the program's behavior as a regular expression. Let r denote a regular expression, defined as follows.

$$r \ ::= \ \epsilon \ \mid \ \emptyset \ \mid \ f \ \mid \ r \cdot r \ \mid \ r + r \ \mid \ r^\star$$

where ϵ denotes the empty string, $\emptyset$ denotes the empty set, f denotes a set only containing f, $r \cdot r$ denotes set concatenation, $r + r$ denotes set union, and $r^\star$ denotes the Kleene-star operator. Function $[\![p]\!] = (r, s)$ takes a program p and outputs a pair that holds a regular expression r of the ongoing behavior and a set of returned behaviors s, where s is a finite set of regular expressions. Intuitively, the inference's output captures the two kinds of status: the first component r represents $\texttt{0} \vdash l \in p$, and the second component s represents $\texttt{R} \vdash l \in p$. The case for $f()$ yields regular expression f and no returned behaviors. The case for `skip` yields ϵ and no returned behaviors. The case for `return` is more interesting: the function yields $\emptyset$ so that no behavior can ensue the return, and the empty string in the set of returned behaviors arises from $\texttt{R} \vdash [] \in \texttt{return}$ in Rule RET. The case for $p_1; p_2$ the s_1 on the right-hand side of the output is due to any early return of p_1 (Rule SEQ-1); the $r_1 \cdot r_2$ on the left-hand side and $\{r_1 \cdot r \mid r \in s_2\}$ capture the behavior of program p_1 without early returns (Rule SEQ-2). The conditional is represents the union of behaviors. The case for loop returns the Kleene-star of the left-hand side the behavior r_1 of p without early returns; on the right-hand side the function prepends zero or more iterations of r_1 followed by the behavior of each early return r. Finally, $\text{infer}(p)$ merges the running and all the halted behaviors.

Example 3. We can now translate the program used in Examples 1 and 2.

$$[\![\texttt{loop}(\star)\ \{a(); \texttt{if}(\star)\ \{b(); \texttt{return}\}\ \texttt{else}\ \{c()\}\}]\!] =$$

$$\big((a \cdot ((b \cdot \emptyset) + c)\big)^\star, \{\big(a \cdot ((b \cdot \emptyset) + c)\big)^\star \cdot a \cdot b\})$$

Theorem 1 (Soundness)**.** *If $l \in L(p)$, then $l \in \text{infer}(p)$.*

Proof. To establish this result, we must first show the following lemma (1). If $[\![p]\!] = (r, s)$ and $\mathtt{0} \vdash l \in p$, then $l \in r$. The proof follows by induction on the derivation of $\mathtt{0} \vdash l \in p$.

Next, we show (2) that if $[\![p]\!] = (r, s)$ and $\mathtt{R} \vdash l \in p$, then there exists an $r' \in s$ and $l \in r'$. The proof follows by induction on the derivation of $\mathtt{R} \vdash l \in p$. The interesting cases are Rule SEQ-1 and Rule LOOP-3, which require lemma (1).

Our goal is to show that $l \in \text{infer}(p) = l \in r + r'_1 + \cdots + r'_n$ where $s = \{r'_1, \ldots, r'_n\}$ and $[\![p]\!] = (r, s)$. Given our assumption $l \in L(p)$, then there are two cases to consider. Case $\mathtt{0} \vdash l \in p$, then we apply (1), and obtain that $l \in r$, hence $l \in r + r'_1 + \cdots + r'_n$.

Case $\mathtt{R} \vdash l \in p$, then we apply (2) to know that there exists an $r' \in s$ and $l \in r'$. Given assumption $l \in r'$, then by induction on the structure of s we can derive that $l \in r + r'_1 + \cdots + r'_n$. □

Theorem 2 (Completeness)**.** *If $l \in \text{infer}(p)$, then $l \in L(p)$.*

Proof. The proof follows the same structure of Theorem 1. We show (1): If $[\![p]\!] = (r, s)$ and $l \in r$, then $\mathtt{0} \vdash l \in p$. The proof follows by induction on the derivation of $l \in r$.

Next, we show (2) that if $[\![p]\!] = (r, s)$, $r' \in s$, and $l \in r'$, then $\mathtt{R} \vdash l \in p$. The proof follows by induction on the structure of p. The interesting cases are sequence, and loop, both of which require the use of (1). To conclude the case for loop we also require an auxiliary result: if $\mathtt{0} \vdash l_1 \in \mathtt{loop}(\star)\ \{p\}$ and $s \vdash l_2 \in p$, then $s \vdash l_1 \cdot l_2 \in \mathtt{loop}(\star)\ \{p\}$.

We have that $l \in \text{infer}(p) \equiv l \in r + r'_1 + \cdots + r'_n$ where $s = \{r'_1, \ldots, r'_n\}$ and $[\![p]\!] = (r, s)$. We must show that $\exists s$ such that $s \vdash l \in p$. By induction on the output of $\text{infer}(p)$ we can show that either $l \in r$ or there exist r' such that $r' \in s$ and $l \in r'$. For the former we use (1) and for the latter we use (2). □

Our main theoretical result is the correctness of our extraction process (function $\text{infer}(p)$, which states that the behavior of a program is a regular language.

Corollary 1. *We have that $L(p)$ is a regular language.*

Proof. $L(p)$ is regular since $\text{infer}(p)$ recognizes $L(p)$. □

IV. Related work

There have been different approaches to model check programming languages. Regarding C++, several tools focus on memory safety issues and specific features such as exception handling [18], [29]–[31]. Some tools analyze an intermediate language, *e.g.,* LLVM [32]–[34] but then face the challenge of losing context information. In [35], authors present a comparative evaluation of fully automatic software verifiers for C and Java [16], [36]–[38]. In particular, Java PathFinder [15] is a very well-known tool in this domain. For Lustre: JKind [16], [19]. For Python: MSVL [39]. Domain-specific languages that can be verified for correctness include P [40], [41], Rebeca [42], and synchronous reactive languages [43]–[45]

Utilizing Finite State Machines (FSMs) for program behavior modeling is a prevalent approach. For instance, in [46], the authors suggest a technique for extracting finite-state models of object-oriented class interfaces automatically and they check whether the program exhibits the expected behavior. VeriSolid [47], [48] applies formal methods to verify smart contracts specified as transition-systems, and includes a visualization tool. JavaBIP [17] is a framework that uses annotations directly on Java code in order to coordinate existing concurrent software components.

Typestates [20], [49] refine the concept of type with information about which operations can be used in a particular context. Multiple authors apply typestates to general-purpose programming languages [50]–[53]. Shelley explores a similar notion but from a modeling checking perspective; moreover typestates are based on state-change, rather than on call ordering constraints.

V. Conclusion and Future Work

In this paper, we present the inference of Shelley specifications from MicroPython classes, which are used in the context of model checking. The inference process consists of three steps: method dependency extraction, method behavior extraction, and method invocation analysis. Our main contribution is formalizing and establishing the correctness of method behavior extraction that specifies the behavior of a small imperative calculus as a regular language.

Future work. Shelley delegates the actual model checking to NuSMV [24], by implementing a translation from a nondeterministic finite automaton (NFA) into a NuSMV model. Our approach is essentially to encode a regular-language as a ω-regular language. We would like to evaluate other approaches that work directly in regular-languages, such as [14] and [54].

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 2204986. This work was supported by FCT through scholarships SFRH/BD/131418/2017 and SFRH/BI/153747/2019, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

References

[1] G. J. Holzmann and M. H. Smith, "Software model checking: extracting verification models from source code," *Softw. Test. Verification Reliab.*, vol. 11, no. 2, pp. 65–79, 2001.

[2] J. K. P., S. Jayaraman, B. Jayaraman, and M. Sethumadhavan, "Finite-state model extraction and visualization from java program execution," *Softw. Pract. Exp.*, vol. 51, no. 2, pp. 409–437, 2021.

[3] I. Bocic and T. Bultan, "Symbolic model extraction for web application verification," in *ICSE*. IEEE / ACM, 2017, pp. 724–734.

[4] F. S. Goncalves, D. Pereira, E. Tovar, and L. B. Becker, "Formal verification of AADL models using UPPAAL," in *SBESC*. IEEE Computer Society, 2017, pp. 117–124.

[5] J. Huselius, J. Kraft, H. Hansson, and S. Punnekkat, "Evaluating the Quality of Models Extracted from Embedded Real-Time Software," in *ECBS*. IEEE Computer Society, 2007, pp. 577–585.

[6] A. Mavridou, H. Bourbouh, P. Garoche, D. Giannakopoulou, T. Pressburger, and J. Schumann, "Bridging the Gap Between Requirements and Simulink Model Analysis," in *REFSQ*, ser. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org, 2020.

[7] M. Schwammberger and V. Klös, "From Specification Models to Explanation Models: An Extraction and Refinement Process for Timed Automata," in *FMAS*, ser. EPTCS, vol. 371, 2022, pp. 20–37.

[8] S. Nejati, K. Gaaloul, C. Menghi, L. C. Briand, S. Foster, and D. Wolfe, "Evaluating model testing and model checking for finding requirements violations in simulink models," in *FSE*. ACM, 2019, pp. 1015–1025.

[9] A. Mavridou, H. Bourbouh, D. Giannakopoulou, T. Pressburger, M. Hejase, P. Garoche, and J. Schumann, "The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained," in *RE*. IEEE, 2020, pp. 300–310.

[10] A. Santos, A. Cunha, and N. Macedo, "Static-Time Extraction and Analysis of the ROS Computation Graph," in *IRC*. IEEE, 2019, pp. 62–69.

[11] D. George. (2022) MicroPython. [Online]. Available: https://micropython.org

[12] C. M. de Ferro, T. Cogumbreiro, and F. Martins, "Shelley, a framework for model checking call ordering on hierarchical systems," to appear in COORDINATION, ser. LNCS. Springer, 2023.

[13] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL semantics for runtime verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[14] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI*. AAAI Press, 2013, p. 854–860.

[15] D. Giannakopoulou and C. S. Pasareanu, "Interface generation and compositional verification in JavaPathfinder," in *FASE*, ser. LNCS, vol. 5503. Springer, 2009, pp. 94–108.

[16] J. Hatcliff and M. B. Dwyer, "Using the Bandera tool set to model-check properties of concurrent Java software," in *CONCUR*, ser. LNCS, vol. 2154. Springer, 2001, pp. 39–58.

[17] S. Bliudze, A. Mavridou, R. Szymanek, and A. Zolotukhina, "Exogenous coordination of concurrent software components with JavaBIP," *Softw. Pract. Exp.*, vol. 47, no. 11, pp. 1801–1836, 2017.

[18] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5-6, pp. 505–525, 2007.

[19] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind model checker," in *CAV*. Springer, 2018, pp. 20–27.

[20] R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, p. 157–171, 1986.

[21] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, p. 279–295, 1997.

[22] P. Parízek, F. Plasil, and J. Kofron, "Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker," in *SEW*. IEEE, 2006, pp. 133–141.

[23] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, 1997.

[24] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An opensource tool for symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364.

[25] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification," in *CAV*. Berlin, Heidelberg: Springer-Verlag, 2011, p. 184–190.

[26] O. Bunte, J. F. Groote, J. J. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. Willemse, "The mCRL2 toolset for analysing concurrent systems: improvements in expressivity and usability," in *TACAS*. Springer, 2019, pp. 21–39.

[27] C. M. de Ferro, T. Cogumbreiro, and F. Martins, "Formalizing Model Inference of MicroPython," may 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7892202

[28] ——, "Shelley: a framework for model checking call ordering on hierarchical systems," may 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7884206

[29] F. R. Monteiro, M. R. Gadelha, and L. C. Cordeiro, "Model checking C++ programs," *Softw. Test. Verification Reliab.*, vol. 32, no. 1, 2022.

[30] N. Blanc, A. Groce, and D. Kroening, "Verifying C++ with STL containers via predicate abstraction," in *ASE*. ACM, 2007, pp. 521–524.

[31] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[32] S. Falke, F. Merz, and C. Sinz, "The bounded model checker LLBMC," in *ASE*. IEEE, 2013, pp. 706–709.

[33] S. Thompson and G. Brat, "Verification of c++ flight software with the mcp model checker," in *IEEE Aerospace Conference*, 2008, pp. 1–9.

[34] Z. Baranová, J. Barnat, K. Kejstová, T. Kucera, H. Lauko, J. Mrázek, P. Rockai, and V. Still, "Model checking of C and C++ with DIVINE 4," in *ATVA*, ser. LNCS, vol. 10482. Springer, 2017, pp. 201–207.

[35] D. Beyer, "Automatic verification of C and java programs: SV-COMP 2019," in *TACAS*, ser. LNCS, vol. 11429. Springer, 2019, pp. 133–155.

[36] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf, "Jayhorn: A framework for verifying java programs," in *CAV*, ser. LNCS, vol. 9779. Springer, 2016, pp. 352–358.

[37] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík, "JBMC: A bounded model checking tool for verifying java bytecode," in *CAV*, ser. LNCS, vol. 10981. Springer, 2018, pp. 183–190.

[38] Y. Noller, C. S. Pasareanu, A. Fromherz, X. D. Le, and W. Visser, "Symbolic pathfinder for SV-COMP - (competition contribution)," in *TACAS*, ser. LNCS, vol. 11429. Springer, 2019, pp. 239–243.

[39] X. Shu, F. Gao, W. Gao, L. Zhang, X. Wang, and L. Zhao, "Model Checking Python Programs with MSVL," in *SOFL*, ser. LNCS, vol. 12028. Springer, 2019, pp. 205–224.

[40] A. Desai *et al.*, "P: Safe Asynchronous Event-driven Programming," in *PLDI*. ACM, 2013, pp. 321–332.

[41] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, "Compositional programming and testing of dynamic distributed systems," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018.

[42] M. Sirjani and M. M. Jaghoori, "Ten Years of Analyzing Actors: Rebeca Experience," in *Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, ser. LNCS, vol. 7000. Springer, 2011, pp. 20–56.

[43] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *SC*, ser. LNCS, vol. 197. Springer, 1984, pp. 389–448.

[44] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[45] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics," *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, 1991.

[46] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in *ISSTA*. ACM, 2002, pp. 218–228.

[47] K. Nelaturu, A. Mavridou, A. G. Veneris, and A. Laszka, "Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid," in *ICBC*. IEEE, 2020, pp. 1–9.

[48] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," *CoRR*, vol. abs/1901.01292, 2019.

[49] R. Garcia, E. Tanter, R. Wolff, and J. Aldrich, "Foundations of typestate-oriented programming," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, 2014.

[50] D. Kouzapas, O. Dardha, R. Perera, and S. J. Gay, "Typechecking protocols with Mungo and StMungo: A session type toolchain for Java," *Sci. Comput. Program.*, vol. 155, pp. 52–75, 2018.

[51] J. Duarte and A. Ravara, "Retrofitting Typestates into Rust," in *SBLP*. ACM, 2021, pp. 83–91.

[52] A. L. Voinea, O. Dardha, and S. J. Gay, "Typechecking Java Protocols with [St]Mungo," in *FORTE*, ser. LNCS, vol. 12136. Springer, 2020, pp. 208–224.

[53] M. Bravetti, A. Francalanza, I. Golovanov, H. Hüttel, M. Jakobsen, M. Kettunen, and A. Ravara, "Behavioural types for memory and method safety in a core object-oriented language," in *APLAS*, ser. LNCS, vol. 12470. Springer, 2020, pp. 105–124.

[54] S. Huang and R. Cleaveland, "A tableau construction for finite linear-time temporal logic," *Journal of Logical and Algebraic Methods in Programming*, vol. 125, p. 100743, 2022.