Shelley: a framework for model checking call ordering on hierarchical systems

Carlos Mão de Ferro[1], Tiago Cogumbreiro[2], and Francisco Martins[3]

[1] LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
[2] University of Massachusetts, Boston, USA
[3] Faculdade de Ciências e Tecnologia, Ponta Delgada, Portugal

Abstract. This paper introduces Shelley, a novel model checking framework used to verify the order of function calls, developed in the context of Cyber-Physical Systems (CPS). Shelley infers the model directly from MicroPython code, so as to simplify the process of checking requirements expressed in a temporal logic. Applications for CPS need to reason about the end of execution to verify the reclamation/release of physical resources, so our temporal logic is stated on finite traces. Lastly, Shelley infers the behavior from code using an inter-procedural and compositional analysis, thus supporting the usual object-oriented programming techniques employed in MicroPython code. To evaluate our work, we present an experience report on an industrial application and evaluate the bounds of the validity checks (up to 12^{12} subsystems under 10 seconds on a desktop computer).

1 Introduction

This paper introduces a novel model checking framework to verify a MicroPython code base against a set of requirements stated in a temporal logic on ordered function calls. MicroPython [31] is an implementation of the Python programming language designed for microcontrollers, providing a large subset of standard Python features in a reduced memory footprint. A major challenge of applying formal methods to the development of embedded cyber-physical systems, is the gap between code and the requirements being checked [9, 25, 43, 45, 56, 57]. To bridge this gap, our approach is to automatically infer the model from the code and let the user focus on stating requirements in a way that is close to the subject matter.

Our research is guided by three main goals. Firstly, *the requirements and the system's behavior should be represented as ordered actions* (which denote function calls), not as a transition system. Since the behavior being analyzed is a call-order graph, then the model and its requirements should closely mirror the given abstraction. In contrast, general-purpose model checkers express their models as state-transitions systems and the requirements are stated in terms of variables of these state-transition systems [7,12,13,27,40,52,59,66]. Additionally, model-checking approaches are usually focused on process communication, which is outside of the scope of the subject of our research.

Secondly, *our domain of interest is finite*, so our temporal logic must be stated on finite traces. When developing code that handles cyber-physical resources, it is crucial to reason (formally) about the release of such resources. Our model checking framework features linear temporal logic on finite traces ($\mathrm{LTL_f}$) [3, 18]. While it is possible to encode $\mathrm{LTL_f}$ in model checker that uses infinite linear temporal logic, such an encoding must be carefully implemented to avoid subtle mistakes [17]; any encoding to infinite traces should be handled automatically.

Thirdly, *code reuse is encouraged.* Since our modeling language is the code being run, then our analysis must support behavior (*i.e.,* function calls) that spans across multiple procedures (say, methods, or functions). A major feature of our model checking framework is to support a compositional interprocedural analysis that follows the usual abstraction and encapsulation techniques of MicroPython codes. Further, Shelley automatically guarantees the correct usage of each system, through function calls, according to their specifications, which reduces the number of correctness claims needed to be written for each system. The idea behind our analysis is akin to protocol conformance in the context of behavior protocols [67].

In summary, our paper makes the following contributions:

1. A domain-specific language to specify stateful systems while abstracting away the internal details of the implementation. (§3)
2. A formalization of generating a system's internal behavior and of checking its validity and decidability results. (§4.2 and §4.3)
3. A toolchain that model-checks requirements expressed by a temporal logic at different hierarchy levels, ensuring correct-by-construction software. (§4.4)
4. An evaluation of our framework: verifying the Aquamote® software written in MicroPython; assessing the performance impact of behavior checking (up to 12^{12} subsystems under 10 seconds on a desktop computer). (§5)

Finally, Section 6 discusses related work and Section 7 concludes the paper. Shelley is open-source and available online [4]. We provide a **demonstration video** of our tool[5] and **an artifact** [26].

2 Overview

In this section, we motivate the challenge of verifying the order of method calls in a object hierarchy, and we overview using Shelley to enforce specified behaviors. The running examples in this paper are taken from an industrial application that motivated our research: Aquamote® is a battery-operated wireless controller that switches water valves according to a scheduled irrigation plan. Following, we verify the controller software that automatically adapts its plan based on the weather forecast and sensor information yielding optimal water consumption results. Listing 2.1 shows the Shelley model of our running example, which is automatically extracted from MicroPython code.

[4] https://github.com/cajomferro/shelley
[5] https://www.youtube.com/watch?v=ZiGPZRQHTWc

Listing 2.1: Shelley specifications for our running example.

```
base Valve {
 initial test -> open, clean;
 open -> close;
 final close -> test;
 final clean -> test;}

Sector (a: Valve, b: Valve) {
 initial try_open_1 -> close {
  a.test; a.open; b.test; b.open;
 }
 initial try_open_2 -> fail {
  { a.test; a.clean; } +
  { a.test; a.open; b.test; b.clean; a.close;}
 }
 initial try_open -> try_open_1, try_open_2 {}
 final fail -> try_open {}
 final close -> try_open {a.close; b.close;}

 check (!b.open) W a.open;
}
```

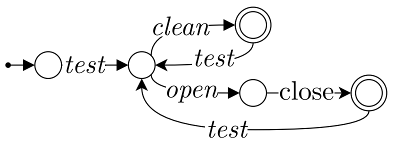
```
AppV1 (a: Valve, b: Valve) {
 final main_1 ->  {
  a.test; a.open; b.test;
  b.open; a.close; b.close;
 }
 final main_2 ->  {
  a.test; a.open; b.test;
  b.clean; a.close;
 }
 final main_3 ->  {
  a.test; a.clean;
 }
 initial main ->
  main_1, main_2, main_3 {}

 check (!b.open) W a.open;
}

AppV2 (s: Sector) {
 final main_1 ->  {
  {s.try_open; s.close;} +
  {s.try_open; s.fail;}
 }
 initial main -> main_1 {}}
```

Finite behaviors. Shelley is designed to verify finite behaviors. For applications that run on battery, it is paramount to specify the explicit release of resources, *e.g.,* turning off the WiFi before suspending. Otherwise, the programmer risks exhausting the device's battery while in suspend mode. Since reasoning about finite executions on a temporal logic based on infinite traces can lead to subtle errors [17], Shelley features LTL_f and the behavior of our models are all finite. Note that Shelley can easily verify long-running applications. Indeed, there is no notion of battery in Shelley, just termination. The specification in Listing 2.1 could very well be of a control software that is connected to an electrical grid. The key point is that Shelley lets us reason about the eventual termination of a program, *e.g.,* specify and enforce that all resources are freed before halting.

Model extraction. This paper discusses the verification of Shelley models. Our tool infers Shelley models from MicroPython code automatically. However, the discussion of the inference process and of its correctness are outside of the scope of this work. Shelley over-approximates the behavior of MicroPython programs in the following ways, *i.e.,* admits false alarms. 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). Non-terminating programs and recursive calls are unsupported. 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.

Listing 2.2: Class `Valve` and a diagram specifying its intended behavior.

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



2.1 Restricting the behavior and usage of systems

As an example of a requirement, a user must test a valve before opening it, so as to minimize the chance of clogging that valve, which would render it unusable. Similarly, to conserve battery, we may want to enforce that the user must test the valve before cleaning up debris. Next, we describe code annotations we defined to achieve the ordering specified in the diagram of Listing 2.2. Our verification goal is to enforce that usages of class `Valve` follow the order specified by its code.

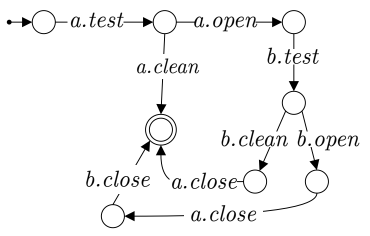
Listing 2.2 shows a high-level API, class `Valve`, written in MicroPython, to control programmatically a valve. Since we need precise control over resource allocation, we declare which methods can be considered safe to execute at the beginning and ending of an object's lifetime. Given that only `test` is marked as `op_initial`, then after creating an instance of `Valve` the only method that can be invoked is `test` (Lines 8 to 12). We then extract the ordering behavior based on the return values. The method `test` returns either an open or clean label, which signifies the following method that can be called. In this case, after testing, the valve can be opened or cleaned, but neither closed nor tested consecutively. When enabled, method `open` opens the valve (Lines 15 to 16); after that we can only close the valve (Lines 19 to 20). Finally, we can clean the valve from debris (Lines 24 to 25). Modifier `op_final` declares that method `close` can be the last method called, with respect to the object's lifetime; method `clean` is also marked as final. Since `open` is *not* marked as final, the valve cannot be left open, as long as the usage of the valve follows its specification.

2.2 Encapsulation complicates verification

We now introduce two versions of the same application, which controls two valves. The first version, called `AppV1`, invokes the valves directly. The second version, called `AppV2`, adds an extra abstraction layer `Sector` that generalizes using both valves as a whole. Our intent is to illustrate the kind of programs we are interested in and how different levels of abstraction complicate verification.

Listing 2.3: Class `AppV1` and a diagram specifying the internal behavior.

```
1  @claim("(! b.open) W a.open")
2  class AppV1:
3   def __init__(self):
4    self.a=Valve();self.b=Valve()
5
6   @op(initial=True, final=True)
7   def main(self):
8     match self.a.test():
9      case "open":
10      self.a.open()
11      match self.b.test():
12       case "open":
13        self.b.open()
14        self.a.close()
15        self.b.close()
16        return ""
17       case "clean":
18        self.b.clean()
19        self.a.close()
20        print("Failed to open valves")
21        return ""
22     case "clean":
23      self.a.clean()
24      print("Failed to open valves")
25      return ""
```



Version 1. Listing 2.3 lists a program that controls two valves, along with a diagram that summarizes its internal behavior. Our program expresses two side-effects: the first represents when both valves are open; and the second is when one of the valves fails to open and must be cleaned. As an example, should we omit the call `open` in Line 13 and Shelley would output the following error message:

```
Error in specification: INVALID SUBSYSTEM USAGE
Counter example: a.test, a.open, b.test, a.close, >b.close<
Subsystems errors:
    * Valve 'b': test, >close< (after test, expecting open or clean)
```

Besides automatically verifying that each valve is being used according to the specification in Listing 2.2, we also want to verify temporal requirements. The claim in Line 1 of Listing 2.3 guarantees that we only open valve `b` after opening valve `a`. For instance, should we switch the calls to subsystems `a` and `b` in such a way that we try to test valve `b` before valve `a` and Shelley would output the following error message:

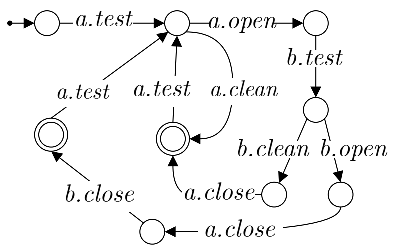
```
Error in specification: FAIL TO MEET REQUIREMENT
Formula: (!b.open) W a.open
Counter example: b.test, b.open, a.test, a.clean, b.close
```

Correctness claims express properties on the ordering of the internal calls to subsystems during the life cycle of that object. Such claims are of great importance for software maintenance, as Shelley checks if code changes preserve the specified internal behavior.

Version 2. In Listing 2.4, our top-level system `AppV2` operates the valves via a `Sector` class (in irrigation jargon, a *sector* is an irrigation zone where sev-

Listing 2.4: Classes `Sector` and `AppV2` and a diagram specifying the `Sector` internal behavior.

```
1  @claim("(! b.open) W a.open")
2  class Sector:
3    def __init__(self):
4      self.a = Valve(); self.b = Valve()
5
6    @op_initial
7    def try_open(self):
8      match self.a.test():
9        case "open":
10         self.a.open()
11         match self.b.test():
12           case "open":
13             self.b.open()
14             return "close"
15           case "clean":
16             self.b.clean(); self.a.close()
17             return "fail"
18       case "clean":
19         self.a.clean()
20         return "fail"
21
22   @op_final
23   def fail(self):
24     print("Failed to open valves")
25     return "try_open"
26
27   @op_final
28   def close(self):
29     self.a.close(); self.b.close()
30     return "try_open"
31
32 class AppV2:
33   def __init__(self):
34     self.s = Sector()
35
36   @op_initial_final
37   def main(self):
38     match self.s.try_open():
39       case "close":
40         self.s.close()
41         return ""
42       case "fail":
43         self.s.fail()
44         return ""
```



eral water valves are grouped together), adding an extra layer of encapsulation. Moreover, to make our code more reusable, class `Sector` abstracts trying to open both valves in method `try_open` (one side effect) and then closing both valves in method `close` (another side effect). When modeling class `Sector`, methods that produce multiple side effects must be distinguished as different operations: we write operation `try_open_1` to express the single trace in method `try_open` that returns `"close"` and we write operation `try_open_2` to express both traces of method `try_open` that return `"fail"`.

Since `Sector` exposes more methods, its behavior is more general than that of `AppV1`. We note, however, that the correctness claim of version 1 also holds in version 2 (Line 1). Deriving the internal behavior is not entirely obvious: *e.g.*, every trace in method `try_open` that returns `"close"` must be able to precede any sequence of method `close`; similarly, every trace of method `try_open` that returns `fail` must be able to precede method `fail`.

Verifying a `Sector` is more complicated than verifying an `AppV1`, not just because the code is scattered across several methods, but because verifying the life cycle of a `Sector` entails reasoning about the internal behavior that arises from all possible usages of `Sector`, which in turn depends on the ordering constraints of each method. Therefore, as an application complexity increases vertically (by arranging systems hierarchically) and horizontally (by having more operations and more systems in each level) the code gets more and more partitioned and we rapidly lose track of the sequence of calls that represent the behavior of our program. Shelley model checks the `Sector` by deriving the internal behavior, which captures all possible internal traces that arise during the life cycle of `Sector`. To this end, Shelley must consider all possible orderings of operations and all possible internal traces that such orderings may generate.

3 The Shelley language

Shelley's specification language precisely defines the ordering constraints of calls when arranging systems hierarchically. We have a specification per system that is usually defined in a text file with the `.shy` extension. We now describe the abstract syntax of Shelley using EBNF notation.

$$S = \mathbf{base}\ X\ s^\star\ c^\star\ \ \mid X\ (x\colon X)^\star\ o^\star\ c^\star \qquad s = \mathbf{initial}?\ \mathbf{final}?\ y \to z^\star$$

$$o = s\ \{e\} \qquad e = \mathbf{skip} \mid x.y \mid e;e \mid \{e\} + \{e\} \mid \mathbf{loop}\ \{e\} \qquad c = \mathbf{claim}\ \phi$$

$$\phi = a \ \mid\ \neg\phi\ \mid\ \phi_1 \land \phi_2\ \mid\ \mathsf{X}\,\phi\ \mid\ \phi_1\ \mathsf{U}\ \phi_2$$

A system S can either be a base or a composite system. The former is identified by keyword **base**, has a name X and a zero or more operation signatures s. Meta-variable X ranges over system names, and meta-variables y, z range over operation names distinct from system names. A composite system uses zero or more subsystems, notation $(x : X)$, with each subsystem having a unique internal identifier x and the name of its system's definition X. Finally, a composite system defines zero or more operations o, each holding a signature and an operation body e. A signature s declares an operation y and has, optionally, an initial and a final modifier; we also declare zero or more operations z that can succeed y. An operation body is a regular expression, where **skip** corresponds to ϵ, $x.y$ corresponds to a call, sequencing is represented by $e;e$, union is given by $\{e\} + \{e\}$, and the Kleene-star is denoted by **loop** $\{e\}$.

Shelley accepts correctness claims expressed in terms of a linear temporal logic on finite traces (LTL$_\mathrm{f}$) [18]. A formula of LTL$_\mathrm{f}$, notation ϕ, uses the familiar LTL notation. Let $\mathcal{P}$ be a set of propositional symbols (representing operations/calls) closed under the boolean connectives, where $a \in \mathcal{P}$. Formula $\mathsf{X}\,\phi$ says that ϕ holds in the next instant. Formula $\phi_1\ \mathsf{U}\ \phi_2$ states that ϕ_1 holds until ϕ_2 eventually holds. Standard boolean abbreviations are used: true, false, $\lor$ (disjunction), and $\implies$ (implication). Derived formulas include: $\mathsf{F}\,\phi = \mathit{true}\ \mathsf{U}\,\phi$ stands for ϕ eventually holds; $\mathsf{G}\,\phi = \neg\,\mathsf{F}\,\neg\phi$ stands for ϕ hold at every step of the trace; $\phi_1\ \mathsf{W}\ \phi_2 = (\phi_1\ \mathsf{U}\ \phi_2) \lor \mathsf{G}\,\phi_1$ stands for ϕ_1 has to hold at least until ϕ_2 or ϕ_1 must remain true forever. Although LTL and LTL$_\mathrm{f}$ share the same syntax, their semantics differ. The same formula can have different meanings according to its interpretation on finite (LTL$_\mathrm{f}$) or infinite (LTL) traces [17]. The fact that traces can be arbitrarily long but *finite* is a key characteristic of our domain of interest, as we want to verifying what happens at the end of the life cycle of each object, *e.g.,* to permit resource deallocation or protocol termination.

4 The Shelley framework

We depict the structure of the Shelley framework in Figure 1. In the following sections, we detail each step of the framework and we formalize external and internal behavior generation and validity. Our main theoretical result is the decidability of the checking procedure.

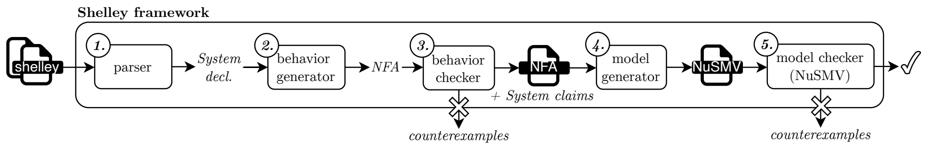


Fig. 1: The structure of the Shelley framework.

Automata theory background. We use standard automata theory to formalize our verification process, *e.g.,* as found in [73]. Here we briefly give the relevant background to make the reading self contained. An *NFA* is a tuple $N = (Q, \Sigma, \Delta, q_0, F)$ consisting of a finite set of states Q, a finite set of input symbols Σ called the alphabet, a transition function $\Delta\colon Q \times \Sigma \cup \{\epsilon\} \to \wp(Q)$, where $\wp(Q)$ is the power set of Q, an initial state $q_0 \in Q$, and a set of final states $F \subseteq Q$. A *DFA* is a tuple $D = (Q, \Sigma, \delta, q_0, F)$ consisting of a finite set of states Q, a finite set of input symbols Σ, a transition function $\delta\colon Q \times \Sigma \to Q$, an initial state $q_0 \in Q$, and a set of final states $F \subseteq Q$. A *word* $w = a_1 a_2 \ldots a_n$ over the alphabet Σ is accepted by an NFA N if, and only if, exists a sequence of states $r_0, r_1, \ldots, r_n$ from Q such that $r_0 = q_0$, $r_{i+1} \in \Delta(r_i, a_{i+1})$, for $i = 0, \ldots, n-1$, and $r_n \in F$. *The language of* N is the set of words accepted by N and denoted by $L(N)$. *The language of a DFA* is defined similarly. Let $f\colon \Sigma \to \Gamma^*$ be a function from one alphabet Σ to words over another alphabet Γ. An extension of function f to $\Sigma^* \to \Gamma^*$ such that $f(\epsilon) = \epsilon$ and $f(w\sigma) = f(w)f(\sigma)$, for any $w \in \Sigma^*$ and $\sigma \in \Sigma$ is called an *homomorphism.* We can extend this function to any language F by letting $f(L) = \{f(w) | w \in L\}$. When X is an automaton, we denote δ_X to be the transition function of X, Q_X denotes the states of X, and F_X denotes the final states of X.

4.1 System declaration

The first step in our framework is to parse the Shelley language into a *system declaration* that is then used throughout the verification process. The following definition makes precise the notion of a system declaration.

Definition 1 (System declaration). *A system declaration is a tuple* $S = (O, I, F, B, C, \sigma, \rho)$ *where* O *is a set of operations a system exposes (its interface),* $I \subseteq O$ *is a set of initial operations, with* $I \neq \emptyset$*,* $F \subseteq O$ *is a set of final operations,* $B \subseteq O \times O$ *is a set of operation transitions (the external system behavior),*

$\sigma\colon \mathcal{U} \to S$ is a function from system names to systems, $\rho\colon O \to D$ is a function from operations to DFAs over subsystems (the internal system behavior), and C is a set of LTL$_f$ formulas (correctness claims).

4.2 Behavior generation

The second step in our framework concerns system's behavior generation. This section formalizes deriving the external (*vide* Definition 2) and internal (*vide* Definition 4) behaviors.

External system's behavior. We make precise the notion of the external behavior by means of an NFA. The set of states includes an initial state q_0 and a state per operation. The transition function can be obtained by following the signature section of each operation. It contains a transition from q_0 to each state that corresponds to an initial operation, and a transition from each operation state to the succeeding operation state. An operation state is accepting whenever the corresponding operation is final.

Definition 2 (External behavior). *Let $S = (O, I, F, B, C, \sigma, \rho)$. The external behavior of S, notation $L_{\text{sys}}(S)$, is defined as $L_{\text{sys}}(S) = L(N_{\text{sys}}(S))$, where $N_{\text{sys}}(S) = (O \cup \{q_0\}, O, \delta, q_0, F)$, for some q_0, and δ is defined below.*

$$\delta(o_1, o_2) = \{o_2\} \text{ if } (o_1, o_2) \in B \qquad \delta(q_0, o) = \{o\} \text{ if } o \in I$$

A given system is considered a *subsystem* if it is integrated by another system. When declaring a subsystem, a unique name is assigned to it and prefixed to every usage of an operation of that subsystem. Definition 3 makes precise the notion of subsystem behavior.

Definition 3 (Subsystem behavior). *Let $S = (O, I, F, B, C, \sigma, \rho)$. We say that the instantiation of S with u, notation $L_{\text{sub}}(S, u)$, is the regular language given by the homomorphism f over $L_{\text{sys}}(S)$ where $f(o) = u.o$, binding the subsystem named u to every operation o of every word in $L_{\text{sys}}(S)$.*

Internal system's behavior. Intuitively, Shelley derives the internal behavior of a system by replacing each operation-edge in the external behavior by the behavior representing each operation body. Definition 4 makes precise the notion of the internal behavior. Figure 2 is the NFA that results from applying the definition below to the `Sector` of Listing 2.1. We denote $X \uplus Y = X \cup Y$ where $X \cap Y = \emptyset$. For brevity, let $L_{\text{int}}(S) = L(N_{\text{int}}(S))$.

Definition 4 (Internal behavior). *Let $S = (O, I, F, B, C, \sigma, \rho)$ and let $M = N_{\text{sys}}(S)$. The internal behavior, $L_{\text{int}}(S)$, is defined as $L_{\text{int}}(S) = L(N_{\text{int}}(S))$, where $N_{\text{int}}(S) = (Q, \Sigma, \delta, q_0, F)$ for $Q = Q_M \biguplus_{o \in O} Q_{\rho(o)}$, $\Sigma = \bigcup_{u \in \text{dom}(\sigma)} \Sigma_u$, Σ_u*

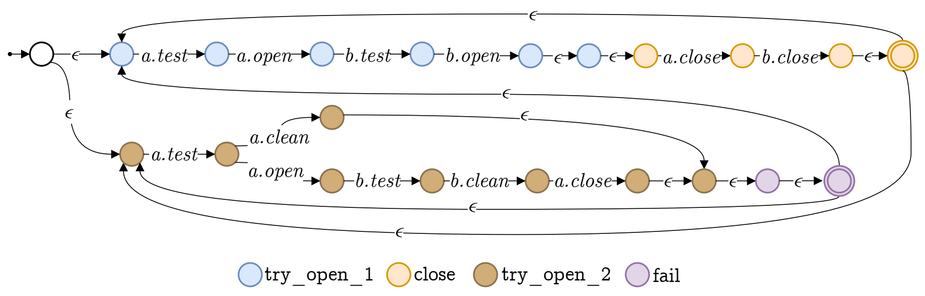


Fig. 2: Internal behavior of `Sector` given as a state diagram (NFA). Sink states are omitted.

is the alphabet of the DFA that recognizes $L_{\mathrm{sub}}(\sigma(u), u)$*, and* δ *is defined below where* $q_{0,o}$ *denotes the initial state of* $\rho(o)$*:*

$$\begin{aligned}\delta(q,\epsilon) &= \{q_{0,o} \mid o \in \delta_M(q,o)\} \quad \textit{if } q \in Q_M\\ \delta(q,u.o) &= \{\delta_{\rho(o')}(q,u.o)\} \textit{ if } q \in Q_{\rho(o')}\\ \delta(q,\epsilon) &= \{o\} \textit{ if } q \in F_{\rho(o)}\end{aligned}$$

4.3 Valid behavior checking

The third step, which concerns behavior checking, ensures that both the external and internal behaviors are valid. This section formalizes both techniques and details how Shelley reports errors in case of an invalid internal behavior.

External behavior validity. Shelley ensures that all operations are reachable from an initial operation in at least one usage of the system.

Definition 5 (Valid external behavior). *An operation* o *is valid if* $o \in t$ *and* $t \in L_{\mathrm{sys}}(S)$ *for some trace* t*. A system's external behavior is valid if all of its operations are valid.*

The algorithm used in Theorem 1 ensures that all operations of a system declaration appear in at least one trace, thus disallowing erroneous cases.

Theorem 1 (Decidability of valid external behavior). *Given a system* S *we can decide whether* S *has a valid external behavior.*

Proof. Let $S = (O, I, F, B, C, \sigma, \rho)$ and let $o \in O$. We must show that it is decidable to find a trace t such that $o \in t$ and $t \in L_{\mathrm{sys}}(S)$. Step 1: build a regular expression with all traces that contain o, using O as the alphabet. Step 2: intersect the regular language of step 1 with the regular language of $L_{\mathrm{sys}}(S)$. Step 3: if the intersection is empty, then $o \in t$ and $t \in L_{\mathrm{sys}}(S)$; otherwise o is invalid. The algorithm is decidable because all steps are build from decidable regular language operations.

Internal behavior validity. Shelley considers the internal behavior of a system valid when every subsystem being used follows its specification. The following definitions make precise the notions of usage behavior and valid internal behavior.

Definition 6 (Usage behavior). *Let* $S = (O, I, F, B, C, \sigma, \rho)$*. The* projection *of* S *on* u*, notation* $\mathrm{proj}(S, u)$*, is the regular language given by the homomorphism* f *from* $N_{\mathrm{int}}(S)$ *into* $\sigma(u)$ *with* $f(u.o) = o$ *and* $f(u'.o) = \epsilon$ *when* $u \neq u'$*.*

Definition 7 (Valid internal behavior). *Let* $S = (O, I, F, B, C, \sigma, \rho)$*. System* S *has a* valid internal behavior *if for all* $u \in \mathrm{dom}(\sigma)$*, then* $L(\mathrm{proj}(S, u)) \subseteq L_{\mathrm{sub}}(S, u)$*.*

Theorem 2 (Decidability of valid internal behavior). *Given a system* S *we can decide whether* S *has a valid internal behavior.*

Proof. Let $S = (O, I, F, B, C, \sigma, \rho)$ and let $u \in \mathrm{dom}(\sigma)$. We must show that $L(\mathrm{proj}(S, u)) \subseteq L_{\mathrm{sub}}(S, u)$ is decidable. $L(\mathrm{proj}(S, u))$ is a regular language, since it is a homomorphism from a regular language ($N_{\mathrm{int}}(S)$). Likewise, $L_{\mathrm{sub}}(S, u)$ is a regular language, since it is also a homomorphism from a regular language $L_{\mathrm{sys}}(S)$. Set inclusion between regular languages is decidable.

Error provenance. A crucial feature of any checker is giving meaningful feedback when verification fails. When a system's internal behavior is invalid, our tool: 1) finds a trace of calls that misuse at least one subsystem (*internal trace*); 2) determines which trace of operations caused that trace of calls; 3) identifies the root-cause of the misusage. To obtain (1) (automata-)subtract the subsystem behavior from the usage behavior; Shelley identifies the *smallest* internal trace in the resulting FSM, with a breadth-first search. To obtain (2) annotate the states of the internal behavior with the operation that produced each call. To obtain (3) use the internal trace to navigate the behavior FSM, transitioning from state to state according to the sequence of calls; if after a transition, the algorithm finds itself in a non-accepting state that cannot reach any accepting state, then the call used to transition is the root-cause of the error.

4.4 Model generation and claim checking

Shelley model checks an NFA against an $\mathrm{LTL}_{\mathrm{f}}$ formula to verify correctness claims. To this end, we rely on NuSMV [13]. Shelley converts an NFA into a Kripke structure, and converts an $\mathrm{LTL}_{\mathrm{f}}$ into an LTL. The NFA may represent either an external behavior or an internal behavior, but such distinction is irrelevant at this stage.

Translating from an $\mathbf{LTL_f}$ claim into an LTL claim. Our implementation follows [17]. Given an $\mathrm{LTL}_{\mathrm{f}}$ formula ϕ, function $[\![\cdot]\!]$ yields an equivalent (infinite) LTL. The idea is to use a sentinel variable *end* that encodes the end of a finite trace. Let $\mathcal{P}$ represent the set of propositional symbols. Variable *end* must be distinct from all variables mentioned in ϕ, *i.e., end* $\notin \mathcal{P}$. The translation must ensure that: variable *end* eventually holds, $\mathsf{F}\,\mathit{end}$; once *end* is true it remains true, $\mathsf{G}\,\mathit{end} \implies \mathsf{X}\,\mathit{end}$; and, no other variable in $\mathcal{P}$ becomes true after end is true, $\mathsf{G}\,(\mathit{end} \implies \bigwedge_{a\in\mathcal{P}} \neg a)$. Finally, we define $[\![\cdot]\!]$ as follows:

$$[\![a]\!] = a \qquad [\![\neg\phi]\!] = \neg[\![\phi]\!] \qquad [\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \wedge [\![\phi_2]\!] \qquad [\![\mathsf{X}\,\phi]\!] = \mathsf{X}\,([\![\phi]\!] \wedge \neg \mathit{end})$$

$$[\![\phi_1 \,\mathsf{U}\, \phi_2]\!] = [\![\phi_1]\!] \,\mathsf{U}\, ([\![\phi_2]\!] \wedge \neg\mathit{end})$$

Translating from an NFA into a NuSMV model. We implement the word-acceptance decision procedure of an automata in NuSMV. Let NFA $N = (Q, \Sigma, \Delta, q_0, F)$. Variable `state` ranges over Q and is initialized to q_0; variable `action` ranges over Σ and represents the *next* character of the string being recognized; boolean variable `end` represents the end of the string being recognized. The key insight is to use variables `state` and `end` to represent the *current* state of automata N and variable `action` to represent the next character of the string being recognized. A NuSMV simulation should only proceed until variable `end` becomes true. While `end` $\neq$ *true* update each variable as follows. Update `action` non-deterministically from Σ. Update `state` by applying δ to the current state and the next action, *i.e.,* $\delta(\texttt{state}, \texttt{action})$. Update `end` by checking if the *upcoming* state is final; the intent is to let NuSMV non-deterministically stop if the following state is final, otherwise it should continue (and `end` be set to false). Formally, if $\delta(\texttt{state}, \texttt{action}) \in F$, then set variable `end` to any boolean non-deterministically. Otherwise set variable `end` to false. Our encoding requires a fairness constraint on variable `end`.

5 Evaluation

In Section 5.1 we present statistics of the Aquamote® verification, along with correctness claims and counterexamples. We assess the bounds of the validity checks of Shelley in a benchmark (Section 5.2), by increasing the number of levels of hierarchy (vertical), and by increasing the number of operations and calls (horizontal). To further exercise the correctness of our implementation we run Shelley against a test suite of 297 specifications, which include 33 negative tests.

Setup. Our experiments run on an 8-core Apple M1 Chip with 16GiB of RAM, and Python 3.10.5. We follow the start-up performance methodology detailed by Georges *et al.* [32], taking 11 samples of the execution time of each benchmark and discarding the first sample. Next, we compute the mean of the 10 samples with a confidence interval of 95%, using the standard normal z-statistic.

5.1 Verifying Aquamote® with Shelley

Our use case is based on the Aquamote®, a wireless controller that switches water valves according to a scheduled irrigation plan. The software consists of 9 classes, which yield 9 Shelley system declarations. Class `App` is the entry point and it uses an instance of class `Controller`. The latter encapsulates handling the

Table 1: Checking Aquamote® with Shelley.

	MicroPython		Shelley					NuSMV
System	LoC	Annot.	LoC	Claims	Subs.	Oper.	Calls	LoC
App	34	3	6	1	1	2	19	103
Controller	72	12	29	5	3	9	18	237
HTTP	177	12	12	1	0	10	0	-
Power	13	3	3	0	0	2	0	-
Sectors	45	7	14	1	5	4	12	152
Timer	12	2	2	0	0	1	0	-
Valve	17	3	3	0	0	2	0	-
WiFi	71	8	8	1	0	6	0	-
Wireless	84	13	30	4	2	11	21	301
TOTAL	525	63	107	13	11	47	70	793

success/error conditions of the communication layer (one instance of `Wireless`), decides when to operate the group of valves (one instance of `Sectors`), and decides when to suspend (one instance of `Power`). Class `Sectors` (an extension from Listing 2.4) integrates four valves (`Valve`) and one timer (`Timer`), encapsulating the behavior where the four valves are open and the four valves are shut, mediated by a timer. Class `Wireless` integrates a Wi-Fi client and an HTTP client, encapsulating both protocols within a single communication interface. The remaining classes are all base classes.

Statistics. Table 1 lists statistics for each system declaration. For instance, class `Wireless` has 90 lines of code, 13 lines are source code annotations to generate the specification, which includes 4 claims, one per line. Our MicroPython extension automatically generates a `Wireless` specification of 30 lines of Shelley code. Shelley then generates the external and internal behaviors from the specification, checks the validity of both behaviors, and, finally, generates a NuSMV model with 301 lines of code that corresponds to the internal specification. For each system we report information about 1) MicroPython source code: lines of code (LoC) and number of Shelley annotations (Annot.); 2) Shelley: lines of code and number of correctness claims (Claims), subsystems (Subs.), system operations (Oper.) and calls (Calls); and 3) NuSMV model: lines of code. The NuSMV model is only generated for systems with claims. We present the verification time in Figure 3.

Checking requirements. We illustrate how claims can be used to enforce strict temporal properties that relate more than one subsystem. We discuss an example of a requirement that is checked using a temporal claim taken from the specification of `Sectors`: after turning a valve on, a timer must be waited upon, and exactly after that, the valve must be turned off. The claim shown below applies this requirement to the four valves:

```
check G ((v1.on -> X (t.wait & (X (v1.off)))) & (v2.on -> X (t.wait & (X (v2.off))))
  & (v3.on -> X (t.wait & (X (v3.off)))) & (v4.on -> X (t.wait & (X (v4.off)))));
```

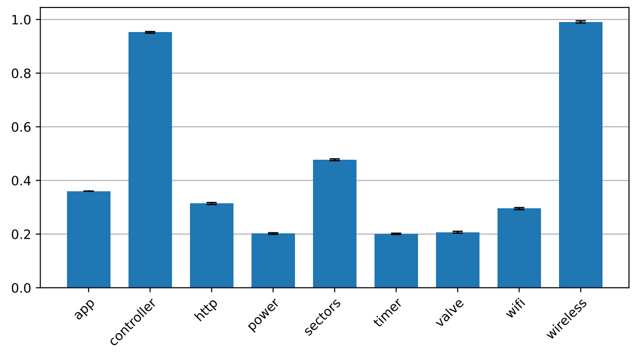


Fig. 3: Mean verification time in seconds for each specification, which includes generating the FSMs for the external and internal behaviors, the validity checks, generating the NuSMV model, and invoking NuSMV. Total verification time for all systems is 3.99 seconds.

The specification of `Timer` states that method `wait` can be invoked without restrictions. However, our temporal claim constraints the timer when it is used after a valve is turned on. Calling `t.wait` twice in between valve activation results in the following error message:

```
Error in specification: FAIL TO MEET REQUIREMENT
Formula: G ((v1.on -> X (t.wait & (X (v1.off)))) & (v2.on -> X (t.wait & (X (v2.off))))
  & (v3.on -> X (t.wait & (X (v3.off)))) & (v4.on -> X (t.wait & (X (v4.off)))))
Counter example:  v2.on, t.wait, v2.off, v1.on, >t.wait, t.wait<, v1.off
```

We further note that the temporal claim disallows opening more than one valve at the same time, which is crucial in the Aquamote® use case given that in many cases the water pressure is insufficient.

5.2 Performance impact of behavior checking

In this section, we measure the performance of behavior generation (*cf.* Section 4.2) and valid behavior checking (*cf.* Section 4.3). In this experiment, the models have no claims, so the claim checking algorithm does not run.

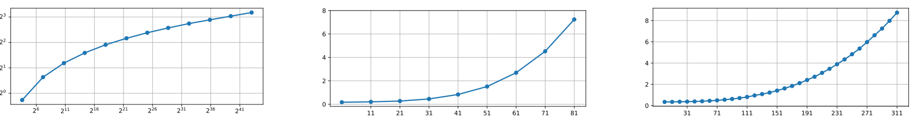
Experiment A. We evaluate the effect of increasing the number of hierarchy levels in terms of time, by ranging from 1 to 12 levels. Level 1 instantiates twelve base systems. Every subsequent level instantiates twelve systems of the level below, *e.g.,* the system in level 5 instantiates twelve systems of level 4.

Discussion. In Figure 4a, we see that the verification time grows logarithmically. The verification takes ∼1.5 seconds per level and each level grows exponentially in the number of systems. Shelley verifies a total of 12^{12} ($\sim 2^{43}$) systems (approximately 9 trillion system) under 10 seconds. Since every system has exactly one operation, this is also equivalent to checking a specification with 9 trillion operations. The benchmarks show that the vertical growth follows a logarithmic increase, which contrasts with the exponential increase of the horizontal growth.

Experiment B. We explore the limits of increasing the number of operations that can be verified under 10 seconds of time, as shown in Figure 4c. We define different systems with an increasing number of operations. Figure 4b shows the impact of varying the number of operations between 1 and 81, in increments of 10.

Experiment C. We explore the limits of increasing the number of operations, and, separately, the number of calls, that can be verified under 10 seconds of time, as shown in Figure 4c. We define different systems with an increasing number of calls. Every system has exactly one operation and one subsystem. Figure 4c shows the impact of varying the number of calls between 1 and 311, in increments of 10.

Discussion of experiments B and C. Adding calls has a smaller impact in the verification time than adding operations. For instance, checking 71 operations takes ∼4, 5 seconds while checking the same number of calls takes less than 1 second. These results highlight the importance of encapsulation to achieve compositional verification. For instance, checking a single system that integrates a total of 71 calls takes ∼9 seconds (Figure 4b), versus verifying a system with 12^2 subsystems that integrates a total of 144 calls takes less than 2 seconds (Figure 4a).



(a) Experiment A: versus total number of subsystems (log-log scale). We increase the number of hierarchy levels in terms of time, by ranging from 1 to 12 levels, where subsequent level instantiates twelve systems of the level below.

(b) Experiment B: versus number of operations (linear scale). We define different systems with an increasing number of operations ranging between 1 and 81, in increments of 10.

(c) Experiment C: versus number of calls per operation (linear scale). We define different systems with an increasing number of calls ranging between 1 and 311, in increments of 10. Every system has exactly one operation and one subsystem.

Fig. 4: Measuring the behavior checking time, in seconds.

6 Related work

Many model checkers address concurrency problems focusing on process communication and internal state-change, rather than on ordering constraints. This includes well-known tools such as SPIN [40], MCLR2 [35], UPPAAL [52], NuSMV [13], LTSA [55], and TLA+ [49]. Java PathFinder [33] and the Bandera Tool Set [38], for Java, and JKind [27], for Lustre, are examples of model checkers targeting general-purpose programming languages [27,33,38] but again they focus on concurrency rather than ensuring specific requirements about the behavior of a program. Assume-guarantee reasoning [58,68] specifies requirements in terms of an internal state and pre-/post-conditions, and overcomes the problem of state explosion with compositional verification [10,64]. This technique has been used by different modeling tools [1,53,55]. The Gamma Statechart Composition Framework [60] offers a modeling language to compose Statecharts [37,69], which can then be model-checked. Statecharts have been applied in many different contexts, including object-oriented languages [14] and verified using process calculi as well [70]. VeriSolid [59,63] applies formal methods to verify smart contracts specified as transition-systems, and includes a visualization tool.

Typestates [77] refine the concept of type with information about which operations can be used in a particular context. Multiple authors apply typestates to object-oriented programming [2,8,11,16,24,28,29,44,48,61,62,78]. Plaid is a programming language designed from the ground up to explore typestates [2,8,28,78]. The main challenges being tackled include object aliasing, linearity, and access permission. Some authors are applying typestates to general-purpose programming languages [4,11,19,24,44,48,61,81]. Shelley explores a similar notion but from a model checking perspective; moreover typestates are based on state-change, rather than on call ordering constraints.

Type systems allow for the verification of a fix set of properties that are guaranteed by the type discipline itself. Type-and-effect systems [34,47,65,75,76,79] and permissive interfaces [39] are concerned with checking that a program respects a certain effect discipline. Sequential effect systems [79] reason about the program order.

Session types [15,20,41,42,50,51,82], a form of behavioral type, encode the data flow in a conversation between two or more parties [20,51,82] and focus on reasoning about the data flow in a conversation between two or more parties. Session types have been also explored for object-oriented languages [23,30,71,72,80]. Similar to Shelley, session types express ordered operations, but it is not possible to compose parties hierarchically and achieve a modular verification.

Behavior protocols [46,54,66,67] have been used to verify software components. They can describe stateful component systems, be automatically checked for behavior validity (known as protocol conformance) [54], and model-checked at different levels [46,66] but lack any form of component hierarchy, as this formalism was envisioned to describe peer-to-peer and client-server architectures.

Finally, the following are programming languages that can be verified for correctness, but lack the notion of hierarchy of events that we explore in this paper. P [22] is a domain-specific programming language to specify a system as a collection of interacting state machines that asynchronously communicate with each other using events. ModP [21] extends P with a notion of compositionality expressed as an actor system. Rebeca [74] follows similar principles and focus on implementations details. JavaBIP [9] is a framework that uses annotations directly on Java code in order to coordinate existing concurrent software components. Synchronous reactive languages [5, 6, 36] share a focus with Shelley on ordered event systems.

7 Conclusion

In this paper, we introduce Shelley, a domain-specific model checker where the models represent ordered actions and the requirements are $\mathrm{LTL_f}$ formulas. We formalize the process of obtaining the internal behavior from a Shelley model, as well as a decision procedure to check its validity with respect to the given ordering constraints, which we prove to be decidable. Further, we present a translation from a model's behavior into an off-the-shelf model checker. We assess our approach on an industrial case study, which includes detailed statistics of our specification, *e.g.,* 107 lines of Shelley generate 793 lines of NuSMV, verified in less than 4 seconds. Finally, we evaluate the performance of our integration checker on three scenarios and show that Shelley can check 12^{12} subsystems under 10 seconds, highlighting the importance of our modular verification.

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

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, USA, 1st edn. (2010)
2. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-Oriented Programming. In: OOPSLA. p. 1015–1022. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1639950.1640073
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. Journal of Logic and Computation **20**(3), 651–674 (2010)
4. Beckman, N.E., Kim, D., Aldrich, J.: An empirical study of object protocols in the wild. In: Mezini, M. (ed.) ECOOP. Lecture Notes in Computer Science, vol. 6813, pp. 2–26. Springer (2011). https://doi.org/10.1007/978-3-642-22655-7_2
5. Benveniste, A., Le Guernic, P., Jacquemot, C.: Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics. Sci. Comput. Program. **16**(2), 103–149 (1991). https://doi.org/10.1016/0167-6423(91)90001-E
6. Berry, G., Cosserat, L.: The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In: SC. Lecture Notes in Computer Science, vol. 197, pp. 389–448. Springer (1984). https://doi.org/10.1007/3-540-15670-4_19
7. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification. p. 184–190. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)
8. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA. pp. 301–320. ACM (2007). https://doi.org/10.1145/1297027.1297050

9. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Exogenous coordination of concurrent software components with javabip. Softw. Pract. Exp. **47**(11), 1801–1836 (2017). https://doi.org/10.1002/spe.2495
10. Bourbouh, H., Farrell, M., Mavridou, A., Sljivo, I., Brat, G., Dennis, L.A., Fisher, M.: Integrating formal verification and assurance: An inspection rover case study. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NFM. Lecture Notes in Computer Science, vol. 12673, pp. 53–71. Springer (2021). https://doi.org/10.1007/978-3-030-76384-8_4
11. Bravetti, M., Francalanza, A., Golovanov, I., Hüttel, H., Jakobsen, M., Kettunen, M., Ravara, A.: Behavioural types for memory and method safety in a core object-oriented language. In: d. S. Oliveira, B.C. (ed.) APLAS. Lecture Notes in Computer Science, vol. 12470, pp. 105–124. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_6
12. Bunte, O., Groote, J.F., Keiren, J.J., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.: The mCRL2 toolset for analysing concurrent systems: improvements in expressivity and usability. In: Proceedings of TACAS. pp. 21–39. Springer (2019)
13. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
14. Coleman, D., Hayes, F., Bear, S.: Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. IEEE Trans. Software Eng. **18**(1), 9–18 (1992). https://doi.org/10.1109/32.120312
15. Coppo, M., et al.: A Gentle Introduction to Multiparty Asynchronous Session Types. In: SFM. Lecture Notes in Computer Science, vol. 9104, pp. 146–178. Springer (2015). https://doi.org/10.1007/978-3-319-18941-3_4
16. Dai, Z., Mao, X.and Lei, Y., Qi, Y., Wang, R., Gu, B.: Compositional mining of multiple object api protocols through state abstraction. TheScientificWorldJournal (2013). https://doi.org/10.1155/2013/171647
17. De Giacomo, G., De Masellis, R., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: AAAI. p. 1027–1033. AAAI Press (2014)
18. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. p. 854–860. AAAI Press (2013)
19. DeLIne, R., Fahndrich, M.: The fugue protocol checker: Is your software baroque? Tech. Rep. MSR-TR-2004-07 (January 2004), https://www.microsoft.com/en-us/research/publication/the-fugue-protocol-checker-is-your-software-baroque/
20. Deniélou, P., Yoshida, N.: Multiparty Session Types Meet Communicating Automata. In: ESOP. Lecture Notes in Computer Science, vol. 7211, pp. 194–213. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_10
21. Desai, A., Phanishayee, A., Qadeer, S., Seshia, S.A.: Compositional programming and testing of dynamic distributed systems. Proc. ACM Program. Lang. **2**(OOPSLA) (2018). https://doi.org/10.1145/3276529
22. Desai, A., et al.: P: Safe Asynchronous Event-driven Programming. In: PLDI. pp. 321–332. ACM (2013)
23. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session types for object-oriented languages. In: Thomas, D. (ed.) ECOOP. Lecture Notes in Computer Science, vol. 4067, pp. 328–352. Springer (2006). https://doi.org/10.1007/11785477_20

24. Duarte, J., Ravara, A.: Retrofitting typestates into rust. In: Vasconcellos, C.D., Roggia, K.G., Bousfield, P., Collereii, V., Fernandes, J.P., Pereira, M. (eds.) SBLP. pp. 83–91. ACM (2021). https://doi.org/10.1145/3475061.3475082
25. Dutle, A., Muñoz, C.A., Conrad, E., Goodloe, A., Titolo, L., Perez, I., Balachandran, S., Giannakopoulou, D., Mavridou, A., Pressburger, T.: From Requirements to Autonomous Flight: An Overview of the Monitoring ICAROUS Project. In: Luckcuck, M., Farrell, M. (eds.) FMAS. EPTCS, vol. 329, pp. 23–30 (2020). https://doi.org/10.4204/EPTCS.329.3
26. de Ferro, C.M., Cogumbreiro, T., Martins, F.: Shelley: a framework for model checking call ordering on hierarchical systems (May 2023). https://doi.org/10.5281/zenodo.7884206, https://doi.org/10.5281/zenodo.7884206
27. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: CAV. pp. 20–27. Springer (2018)
28. Garcia, R., Tanter, E., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. **36**(4) (2014). https://doi.org/10.1145/2629609
29. Gay, S.J., Gesbert, N., Ravara, A., Vasconcelos, V.T.: Modular session types for objects. Log. Methods Comput. Sci. **11**(4) (2015). https://doi.org/10.2168/LMCS-11(4:12)2015
30. Gay, S.J., Vasconcelos, V.T., Ravara, A., Gesbert, N., Caldeira, A.Z.: Modular session types for distributed object-oriented programming. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL. pp. 299–312. ACM (2010). https://doi.org/10.1145/1706299.1706335
31. George, D.: MicroPython (2022), https://micropython.org
32. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. In: OOPSLA. pp. 57–76. ACM (2007)
33. Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in JavaPathfinder. In: FASE. Lecture Notes in Computer Science, vol. 5503, pp. 94–108. Springer (2009). https://doi.org/10.1007/978-3-642-00593-0_7
34. Gordon, C.S.: Polymorphic iterable sequential effect systems. ACM Trans. Program. Lang. Syst. **43**(1) (2021). https://doi.org/10.1145/3450272
35. Groote, J.F., Keiren, J.J.A., Luttik, B., de Vink, E.P., Willemse, T.A.C.: Modelling and analysing software in mCRL2. In: FACS. pp. 25–48. Springer (2020)
36. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991)
37. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
38. Hatcliff, J., Dwyer, M.B.: Using the Bandera tool set to model-check properties of concurrent Java software. In: CONCUR. Lecture Notes in Computer Science, vol. 2154, pp. 39–58. Springer (2001). https://doi.org/10.1007/3-540-44685-0_5
39. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/FSE. p. 31–40. ACM (2005). https://doi.org/10.1145/1081706.1081713
40. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
41. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: ESOP. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998). https://doi.org/10.1007/BFb0053567
42. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695

43. Jacklin, S.A.: Survey of verification and validation techniques for small satellite software development. Tech. rep. (2015)
44. Jakobsen, M., Ravier, A., Dardha, O.: Papaya: Global typestate analysis of aliased objects. In: Veltri, N., Benton, N., Ghilezan, S. (eds.) PPDP. pp. 19:1–19:13. ACM (2021). https://doi.org/10.1145/3479394.3479414
45. Katis, A., Mavridou, A., Giannakopoulou, D., Pressburger, T., Schumann, J.: Capture, analyze, diagnose: Realizability checking of requirements in FRET. In: Shoham, S., Vizel, Y. (eds.) CAV. Lecture Notes in Computer Science, vol. 13372, pp. 490–504. Springer (2022). https://doi.org/10.1007/978-3-031-13188-2_24
46. Kofron, J.: Checking software component behavior using behavior protocols and spin. In: Proceedings of SAC. p. 1513–1517. ACM (2007). https://doi.org/10.1145/1244002.1244326
47. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: CSL-LICS. ACM (2014). https://doi.org/10.1145/2603088.2603138
48. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with mungo and stmungo: A session type toolchain for java. Sci. Comput. Program. **155**, 52–75 (2018). https://doi.org/10.1016/j.scico.2017.10.006
49. Lamport, L.: Who builds a house without drawing blueprints? Commun. ACM **58**(4), 38–41 (2015). https://doi.org/10.1145/2736348
50. Lange, J., Tuosto, E., Yoshida, N.: From Communicating Machines to Graphical Choreographies. In: POPL. pp. 221–232. ACM (2015). https://doi.org/10.1145/2676726.2676964
51. Lange, J., Yoshida, N.: Verifying Asynchronous Interactions via Communicating Session Automata. In: CAV. Lecture Notes in Computer Science, vol. 11561, pp. 97–117. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_6
52. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1-2), 134–152 (1997). https://doi.org/10.1007/s100090050010
53. Liu, J., Backes, J.D., Cofer, D., Gacek, A.: From design contracts to component requirements verification. In: NFM. vol. 9690, p. 373–387. Springer (2016). https://doi.org/10.1007/978-3-319-40648-0_28
54. Mach, M., Plásil, F., Kofron, J.: Behavior protocol verification: Fighting state explosion. International Journal of Computer and Information Science **6**(1), 22–30 (2005)
55. Magee, J., Kramer, J.: Concurrency: state models and Java programs. Wiley, 2 edn. (2006)
56. Mavridou, A., Bourbouh, H., Garoche, P., Giannakopoulou, D., Pressburger, T., Schumann, J.: Bridging the gap between requirements and simulink model analysis. In: Sabetzadeh, M., Vogelsang, A., Abualhaija, S., Borg, M., Dalpiaz, F., Daneva, M., Condori-Fernández, N., Franch, X., Fucci, D., Gervasi, V., Groen, E.C., Guizzardi, R.S.S., Herrmann, A., Horkoff, J., Mich, L., Perini, A., Susi, A. (eds.) REFSQ. CEUR Workshop Proceedings, vol. 2584. CEUR-WS.org (2020)
57. Mavridou, A., Bourbouh, H., Giannakopoulou, D., Pressburger, T., Hejase, M., Garoche, P., Schumann, J.: The ten lockheed martin cyber-physical challenges: Formalized, analyzed, and explained. In: Breaux, T.D., Zisman, A., Fricker, S., Glinz, M. (eds.) RE. pp. 300–310. IEEE (2020). https://doi.org/10.1109/RE48521.2020.00040
58. Mavridou, A., Katis, A., Giannakopoulou, D., Kooi, D., Pressburger, T., Whalen, M.W.: From partial to global assume-guarantee contracts: Compositional realizability analysis in FRET. In: Huisman, M., Pasareanu, C.S., Zhan, N. (eds.) FM. Lecture Notes in Computer Science, vol. 13047, pp. 503–523. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_27

59. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for ethereum. CoRR **abs/1901.01292** (2019), http://arxiv.org/abs/1901.01292
60. Molnár, V., Graics, B., Vörös, A., Majzik, I., Varró, D.: The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In: ICSE. pp. 113–116. ACM (2018). https://doi.org/10.1145/3183440.3183489
61. Mota, J., Giunti, M., Ravara, A.: Java Typestate Checker. In: Damiani, F., Dardha, O. (eds.) COORDINATION. Lecture Notes in Computer Science, vol. 12717, pp. 121–133. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_8
62. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: Harris, G.E. (ed.) OOPSLA. pp. 347–366. ACM (2008). https://doi.org/10.1145/1449764.1449792
63. Nelaturu, K., Mavridou, A., Veneris, A.G., Laszka, A.: Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In: ICBC. pp. 1–9. IEEE (2020). https://doi.org/10.1109/ICBC48266.2020.9169428
64. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S.: A model checking framework for hierarchical systems. In: ASE. pp. 633–636. IEEE (2011). https://doi.org/10.1109/ASE.2011.6100143
65. Nielson, F., Nielson, H.R.: Type and effect systems. In: Olderog, E., Steffen, B. (eds.) Correct System Design, Recent Insight and Advances. Lecture Notes in Computer Science, vol. 1710, pp. 114–136. Springer (1999). https://doi.org/10.1007/3-540-48092-7_6
66. Parízek, P., Plasil, F., Kofron, J.: Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In: Proceedings of SEW. pp. 133–141. IEEE (2006). https://doi.org/10.1109/SEW.2006.23
67. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Software Eng. **28**(11), 1056–1076 (2002). https://doi.org/10.1109/TSE.2002.1049404
68. Pnueli, A.: In Transition From Global to Modular Temporal Reasoning about Programs. In: LMCS. NATO ASI Series, vol. 13, pp. 123–144. Springer (1984). https://doi.org/10.1007/978-3-642-82453-1_5
69. Pnueli, A., Shalev, M.: What is in a Step: On the Semantics of Statecharts. In: TACS. Lecture Notes in Computer Science, vol. 526, pp. 244–264. Springer (1991)
70. Roscoe, A.W., Wu, Z.: Verifying Statemate Statecharts Using CSP and FDR. In: ICFEM. Lecture Notes in Computer Science, vol. 4260, pp. 324–341. Springer (2006). https://doi.org/10.1007/11901433_18
71. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In: Müller, P. (ed.) ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
72. Scalas, A., Yoshida, N.: Lightweight Session Programming in Scala. In: Krishnamurthi, S., Lerner, B.S. (eds.) ECOOP. LIPIcs, vol. 56, pp. 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ECOOP.2016.21
73. Sipser, M.: Introduction to the Theory of Computation. International Thomson Publishing, 1st edn. (1996)
74. Sirjani, M., Jaghoori, M.M.: Ten Years of Analyzing Actors: Rebeca Experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday. Lecture Notes in Computer Science, vol. 7000, pp. 20–56. Springer (2011). https://doi.org/10.1007/978-3-642-24933-4_3

75. Skalka, C.: Trace Effects and Object Orientation. In: PPDP. p. 139–150. ACM (2005). https://doi.org/10.1145/1069774.1069787
76. Skalka, C., Smith, S., Van horn, D.: Types and Trace Effects of Higher Order Programs. J. Funct. Program. **18**(2), 179–249 (2008). https://doi.org/10.1017/S0956796807006466
77. Strom, R.E., Yemini, S.: Typestate: A Programming Language Concept for Enhancing Software Reliability. IEEE Trans. Softw. Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929
78. Sunshine, J., Naden, K., Stork, S., Aldrich, J., Tanter, É.: First-class state change in plaid. In: Lopes, C.V., Fisher, K. (eds.) OOPSLA. pp. 713–732. ACM (2011). https://doi.org/10.1145/2048066.2048122
79. Tate, R.: The Sequential Semantics of Producer Effect Systems. In: POPL. p. 15–26. ACM (2013). https://doi.org/10.1145/2429069.2429074
80. Vasconcelos, V.T.: Sessions, from Types to Programming Languages. Bull. EATCS **103**, 53–73 (2011)
81. Voinea, A.L., Dardha, O., Gay, S.J.: Typechecking Java Protocols with [St]Mungo. In: Gotsman, A., Sokolova, A. (eds.) FORTE. Lecture Notes in Computer Science, vol. 12136, pp. 208–224. Springer (2020). https://doi.org/10.1007/978-3-030-50086-3_12
82. Zeng, H., Kurz, A., Tuosto, E.: Interface Automata for Choreographies. Electronic Proceedings in Theoretical Computer Science **304**, 1–19 (2019). https://doi.org/10.4204/eptcs.304.1