Verifying Ruby state machines with CTL

Christopher Stadler

April 26, 2019

Introduction

The automated verification of programs has been a holy grail of the software engineering community since at least the 1960s [1], [2]. While verification techniques have grown in popularity in some segments of industry — primarily those where high reliability is crucial — they are far from widely used. Compare this situation to that of automated testing, which has become ubiquitious in industry. Testing, like verification, is not free — a harness must be created and test cases must be written — but the value it provides is considered to be worth the cost for almost all systems. And compared to verification testing provides much weaker about program correctness.

How can we make verification more accessible? How can we make it less expensive to apply? Even if this reduces the strength of its conclusions can we move the cost-benefit ratio closer to that testing? This project explores three possible answers to these questions:

1. Eliminate the need to write models.
2. Apply techniques to popular languages.
3. Integrate with existing tooling and workflows.

This is done by implementing a tool — `state_machine_checker` — for verifying finite state machines that appear in Ruby programs.[1]

Verification techniques are expensive because they can rarely be applied directly to source code. Instead, verification tools operate on descriptions of abstract models such as logical formulae or finite state machines. Constructing these models by hand, either as part of the design process or from existing source code, is time consuming and error prone. This is especially problematic in software development environments where there is no clear divide between design and implementation phases, requirements change frequently, and the source code is

[1] Currently the implementation itself consists of 801 lines of Ruby code, with 732 additional lines of tests. The full code is available at https://github.com/CJStadler/state_machine_checker (under the MIT license). The latest commit at the time of this writing is 6fbf86c. Pull requests, and suggestions for a more interesting name, are welcome.

never static.[2]. This results in models becoming out of of date faster than they can be constructed [3].

One strategy to eliminate this divergence is to automatically generate models from source code. This has been implemented in tools such as Modex [3] and C2BP [4]. While this feels like a natural solution — if the goal is to verify a program then of course we want a model of its code — since general purpose programming languages were not designed for this purpose the abstraction process is complex and these tools have not been widely adopted. This project explores an alternative solution: generating source code from a model, or embedding a model directly in source code. If we can encode some of the behavior of a program in a "modeling language" then it should be straightforward to translate into a more expressive general purpose language.

Something akin to this is already widely done, although not for the purpose of verification. There are several popular libraries for the Ruby language which allow the user to model some of the behavior of an object as a finite state machine. These libraries parse a description of a state machine and then generate the code (at runtime) to implement it. Developers find this pattern useful because it makes the high-level behavior of the object explicit, and therefore easier to reason about. Since the implementation is generated from the model there is less room for divergence between the intended and actual behavior than if the model was created only for design purposes and then implemented manually.

These state machine models not only make it easier for humans to reason about program behavior, but also create an opportunity for software verification of that behavior.

Ruby is very far from the ideal language for formal verification as it is difficult to reason about statically: it is untyped, everything is mutable, functions can be defined or overriden at runtime, and encapsulation is weakly enforced. But these weaknesses are strengths for our purpose — if we can demonstrate that a verification technique can be applied usefully to Ruby programs this should suggest that there can be applications to any language.

To be very clear: we are not verifying properties of a full Ruby program, but instead are verifying properties of an abstract model which happens to be embedded in a Ruby program. Although this limits how much can be inferred about the behavior of the program from verification of the model, since it is currently not possible to provide any form of automated verification this would still be a significant step forward. And since the state machine pattern is common if a tool can provide a little value to each project the potential total benefit could be large.

[2]See http://agilemanifesto.org/.

State machines in Ruby

A common pattern in object oriented programming is to represent the behavior of an object as a state machine. This is useful when the object may be in one of several states, its behavior changes depending on the state, and only certain transitions among states are allowed. *Design Patterns* gives the following example:

> A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state. For example, the effect of an Open request depends on whether the connection is in its Closed state or its Established state. The State pattern describes how TCPConnection can exhibit different behavior in each state. [5]

An ad-hoc implementation of this pattern might be for the object to track its current state, and to use conditionals to check the state in methods which depend on it. In Ruby this might look like the following:

```
class TCPConnection
  def initialize
    # Create an instance variable with the initial state
    @state = :closed
  end

  def open
    if @state == :closed
      # Transition to the established state
      @state = :established
    else
      raise "Must be in closed state to open."
    end
  end

  # ...
end
```

The major downsides of this approach are that transitions are defined implicitly by the combination of a conditional check of the current state and a modification of the state variable, and these transitions may be spread throughout the code. To resolve these issues there are several open source Ruby libraries which provide abstractions for explicitly defining state machines.[3] Using the State Machines library the above example could be written as

[3]Three of the most popular are State Machines (https://github.com/state-machines/state_machines), AASM (https://github.com/aasm/aasm), and Statesman (https://github.com/gocardless/statesman).

```ruby
require "state_machines"

class TCPConnection
  state_machine initial: :closed do
    event :open do
      transition from: :closed, to: :established
    end

    # ...
  end
end
```

When the Ruby interpreter loads this class the `state_machine` method provided by the State Machines library is called. This executes the provided block, recording the values of the specified `event` and `transition` calls to construct a state machine definition. It then defines instance methods on the enclosing class corresponding to this definition (e.g. an `open` method to execute the transition, and a `closed?` predicate method). In many ways this is equivalent to generating the source code of the class from a state machine definition, but done at runtime.[4]

Model checking and CTL

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [6, p. 11]

The type of properties that can be checked are statements about paths through a finite state machine (FSM). For example: liveness properties like "eventually a state where p is true will be reached", and safety properties like "From a state where p is true no state where q is true can be reached." Many of these properties can be formally expressed using CTL — computation tree logic.[5]

CTL models time as a tree, where from any state all possible paths that could be taken are contained in its descendents. This allows it to express statements about possible paths: "A state where p is true is reachable, and a state where q is true is reachable."[6] In addition to the standard logical operators CTL provides temporal operators and path quantifiers. These must always be joined in a pair of a path quantifier and temporal operator. The semantics of the operators are given below.

[4] Ruby provides extensive support for reflection and "metaprogramming" — modifying behavior at runtime. See https://ruby-doc.com/core-2.6.2/Module.html#method-i-define_method and https://ruby-doc.com/core-2.6.2/BasicObject.html#method-i-method_missing.

[5] First proposed by Clarke and Emerson in 1981 [7].

[6] See [6, p. 313] for a more detailed overview of CTL and its relation to other temporal logics.

Table 1: Temporal operators.

Operator	Semantics
$\mathsf{X}\Phi$	Φ is true in the next state.
$\mathsf{F}\Phi$	Φ will eventually be true.
$\mathsf{G}\Phi$	Φ will always be true.
$\Phi_1\mathsf{U}\Phi_2$	Φ_1 is true until Φ_2 is.

There are two path quantifiers: A and E. $\mathsf{A}\Phi$ holds if Φ holds for all paths from the current state, while $\mathsf{E}\Phi$ holds if Φ holds for at least one path from the current state. For example, $\mathsf{EF}p$ means that there is a path to a p-state, while $\mathsf{AF}p$ means that all paths lead to a p-state.

CTL formulae are evaluated over Kripke structures, which are finite state machines where every state is also mapped to a set of labels. Each label represents an atomic proposition which holds for that state. For example, the CTL formula $\mathit{processing} \wedge \mathsf{AX}\ \mathit{completed}$ would be satisfied by a Kripke structure where the initial state is labeled *processing*, and all of its successor states are labeled *completed*.

To verify CTL properties of Ruby state machines we will therefore need to translate them into Kripke structures.

Implementation

I have implemented a Ruby library — `state_machine_checker` — which verifies CTL properties of Ruby state machines. When added as a dependency to a Ruby project it allows CTL formulae to be written in Ruby, and provides a method to check whether the state machine of a given object satisfies a given formula. Properties of state machines could then be verified as part of the project's test suite.[7] The following demonstrates how to use the core API:

[7]Thomas Wahl has suggested that it could also be useful to verify properties during the actual execution of a program. For example, an object may behave differently depending on whether some future state is reachable. I have not experimented with this, but there is no technical reason why it would not work. However, since the checker will execute user defined transitions to explore the state graph this should only be done if the state machine has no side effects.

```ruby
require "state_machine_checker"
require "Order" # Should define an `Order` class with a state machine.

include StateMachineChecker
include StateMachineChecker::CTL::API

eventually_done = EF(:done?)
result = check_satisfied(eventually_done, -> { Order.new })

if !result.satisfied?
  raise "Formula not satisfied"
end
```

The API is made up of two components: methods for constructing CTL formulae, and `check_satisfied` to perform the model checking itself.

CTL Representation

Every CTL operator (and logical operator) is represented as it's own class. Each of these classes implement the following methods:

- `atoms`: list the atoms contained in this formula and all sub-formulae.
- `check(model)`: determine which states of the given model satisfy this formula, and provide a witness or counterexample for each.

Atomic propositions are represented by the `Atom` class, which also implements the above interface and is the terminal formula. An atom can be constructed either from a symbol which is the name of a predicate method to call on the object, or from a function which takes the object as it's only argument. The apply method evaluates an atom on the given object. The following demonstrates the API:

```ruby
even = atom(->(num) { num.even? })
odd = atom(:odd?)

even.apply(4) # => true
even.apply(5) # => false
odd.apply(4)  # => false
odd.apply(5)  # => true
```

To make writing formulae less verbose an API wrapper is provided, illustrated below:

```
# Constructor shortcuts.
EX(p) # instead of EX.new(p)

# A symbol can be passed to any constructor instead of an atom.
EX(:open?) # instead of EX(atom(:open?))

# Infix operators are implemented as methods on all formulae.
f1.and(f2) # instead of And.new(f1, f2)
f1.EU(f2) # instead of EU.new(f1, f2)
```

Once a formula has been constructed it can be passed to `check_satisfied` to perform the model checking. This method also takes a thunk which must generate an instance of an object in the initial state. The thunk is used to extract the finite state machine and to generate labels. The checking algorithm has three steps:

1. Generate a Kripke structure from the given object.
2. Determine which states satisfy the given CTL formula.
3. Return the result for the initial state.

Generating a Kripke Structure

A Kripke structure is made up of four components:

- A set of states.
- An initial state.[8]
- A set of transitions, each of which has a start and end state.
- A mapping from each state to a set of labels.

The first three are provided by the `state_machines` library. This defines a `state_machine` method on the target class (e.g `Payment.state_machine`), returning a data structure representing the finite state machine. We get a reference to the class by calling the given thunk, which should return an instance of the class. An adapter is used to wrap the FSM data structure, reducing coupling with the `state_machines` library.[9]

To generate labels we first call the `atoms` method of the given formula. To determine which of these atoms are true for a state s we need to evaluate each atom on an object in s. Using depth first search from the initial state, for each state s we first call the thunk to generate a new instance in the initial state. We then execute the transitions from the initial state to s on this instance (by calling the corresponding methods). Now that the instance is in s we can evaluate every atom, collecting those that are true as the labels for s.

[8] This is normally a set, but we currently allow only one.

[9] There are several places in the current code which depend on specifics of the `state_machines` library. I believe these could be easily generalized to work with other state machine libraries, but for the initial implementation I focused instead on simplicity.

The resulting Kripke structure is represented by an instance of the `LabeledMachine` class. In addition to providing access to the sets of states and transitions this implements the following methods which are used for model checking:

- `labels_for_state(state)`: returns a set of atoms which hold for the given state.
- `traverse(from_state, reverse, block)`: This performs a depth first search of the graph from the given state. If `reverse` is true transitions are followed against their normal direction. Each state is yielded to the block, along with a snapshot of the stack of transitions traversed to reach that state.

Checking CTL properties

Each class representing a CTL formula implements its model checking logic in the `check(model)` method. There are three types of formulae: atoms, general logical operators, and CTL specific operators.

Atoms are themselves the labels of our Kripke structure, so an atom a is satisfied for a state s if and only if a is a member of the labels for s.

The implementations of the standard logical operators are straightforward:

- `f1.and(f2)` is satisfied for all states where `f1` and `f2` are satisfied. If both are satisfied than the witness for the first is used, otherwise the counterexample for whichever is unsatisfied is used.
- `f1.or(f2)` is satisfied for all states where either `f1` or `f2` is satisfied. The witness of the first satisfied formula is used. If neither formula is satisfied then the counterexample for `f2` is used.
- `neg(f)` reverses the result of `f` for each state, turning witnesses into counterexamples, and counterexamples into witnesses.

The algorithms implemented for each E operator are as follows:

- `EX(f)`: for each state s satisfying `f` find all transitions leading to s, mark their source states as satisfied, and construct a witness for each by prepending the transition to the witness for s.
- `EF(f)`: for each state s satisfying `f` perform a reverse depth first search from s, while maintaining a stack of transitions. Mark each reached state t as satisfied and construct a witness for it by appending the transitions from t to s with the witness for s.
- `EG(f)`: a new FSM is constructed, containing only those states which satisfy `f`. Kosaraju's algorithm is used to identify the strongly connected components of this projection [8, pp. 615–617]. For each state s of each component perform a reverse depth first search of the projection. Mark every reached state t as satisfied. The witness for t consists of the transi-

tions from t to s, and then a loop from s to s within the strongly connected component.

- `f1.EU(f2)`: For each state s satisfying `f2` perform a reverse depth first search. For each state t reached if it satisfies `f1` then mark it as satisfied and continue. If it does not satisfy `f1` then do not visit any ancestors (because we are searching in reverse) of t.[10] A witness consists of the transitions from t to s.

Model checking logic is not implemented directly for A operators. Instead, they are translated into an equivalent formula using the following definitions:

- $\mathsf{AX}\Phi = \neg\mathsf{EX}\neg\Phi$
- $\mathsf{AF}\Phi = \neg\mathsf{EG}\neg\Phi$
- $\mathsf{AG}\Phi = \neg\mathsf{EF}\neg\Phi$
- $\mathsf{A}\Phi_1\mathsf{U}\Phi_2 = \neg(E[\neg\Phi_2\mathsf{U}\neg(\Phi_1 \vee \Phi_2)] \vee \mathsf{EG}\neg\Phi_2)$

For example, the corresponding code for AF:

```
def check(model)
  Not.new(CTL::EG.new(Not.new(subformula))).check(model)
end
```

The `check_satisfied` method takes the result of the `check` call to the outermost formula and extracts the result for the initial state. The final return value is an instance of the `StateResult` class. This exposes a `satisfied?` predicate, as well as `witness` and `counterexample` methods, each of which returns a list of transitions (if available). However, the expectation is that users will normally not call `check_satisfied` or interact with the result object directly. Instead this will be handled by integrations with testing frameworks.

Tooling support

By integrating with existing tooling we can make it easier for projects to use verification techniques. `state_machine_checker` is designed to be run as part of a test suite because developers are familiar with tests and many projects already have systems set up running test suites as part of their workflows and build pipelines.

To make this seamless `state_machine_checker` includes an integration specifically for the Ruby RSpec testing framework. The RSpec library is commonly used in Ruby projects to write specifications in the form of tests. This makes it a natural fit because we can enable the user to write specifications as verifiable CTL properties instead of only tests. To integrate with RSpec `state_machine_checker` includes a custom "matcher" which wraps the core API. This provides a more natural DSL-like syntax, and for failing specs it

[10] This implements "strong until" — `f2` must be satisfied eventually.

prints an error message containing the formula and a counterexample (if one could be generated).

For example, without the custom matcher we could write the following spec:

```
it "cannot fail after completed" do
  formula = AG(atom(:completed?).implies(neg(EF(:failed?))))
  result = check_satisfied(formula, -> { Payment.new })
  expect(result.satisfied?).to eq(true)
end
```

But if this is violated the resulting message is not very useful:

```
Payment cannot fail after completed
Failure/Error: expect(result.satisfied?).to eq(true)
  expected: true
       got: false
```

The developer could then manually instrument the code to print the counterexample. Using the custom matcher we can instead write

```
it "cannot fail after completed" do
  formula = AG(atom(:completed?).implies(neg(EF(:failed?))))
  expect { Payment.new }.to satisfy(formula)
end
```

Which prints a more helpful message when it is violated:

```
Payment cannot fail after completed
Failure/Error: expect { Payment.new }.to satisfy(formula)
  Expected state machine for Payment#state to satisfy
    "AG((completed?) => (¬(EF(failed?))))" but it does not.
  Counterexample:
    [:started_processing, :complete, :started_processing, :pend, :failure]
```

Similar integrations for other testing frameworks could also be easily provided.

Limitations

There are many limitations on the conclusions one can draw about a program based on the results of `state_machine_checker`. The first is that the tool itself could have an error, potentially reporting a property as satisfied when it is not. There is currently a test suite, but peer-review is definitely needed. The second set of limitations come from the potential divergence between the generated model and the execution of the code itself.

In Ruby all data is mutable and can be mutated from anywhere in a program. This means that there is no way to guarantee that the state machine we use as our model may not be altered during the execution of a program. For example, the state machine for a class could be replaced at any point:

```ruby
Payment.state_machine # returns the state machine
Payment.define_singleton_method(:state_machine) { "not a state machine" }
Payment.state_machine # returns "not a state machine"
```

But in practice such blunt tools are rarely used. More likely is code that mutates the state directly, ignoring the allowed transitions:

```ruby
class Payment
  state_machine initial: :checkout do
    event :start do
      transition from: :checkout, to: :processing
    end
    event :complete do
      transition from: :processing, to: :completed
    end
  end

  # Allow a transition to :checkout from any state.
  def emergency_reset
    @state = :checkout
  end
end
```

The given state machine would satisfy the formula $AG(processing \Rightarrow \neg(EF complete))$, but it would be incorrect to infer that an instance of `Payment` will never be in the `checkout` state after it is `processing`. This is an inherent limitation of the tool and there is no solution beyond documenting it.

Another potential gap between our model and the program's actual execution arises from our technique for generating labels. Our algorithm evaluates each atom on a single instance in each state. Each such instance is produced by walking it through a path from the initial state. However, there could be multiple such paths and we take only one. Atoms must be chosen by the user which only depend on the state of the object — not the path taken, nor any external information. For example,

```ruby
atom(->(x) { x.previous_state == :completed })
```

might be true for a certain state if one path is taken, but not another. This would result in a model where a state is labeled with the atom even though it does not always hold for that state. Again, we can only warn the user about this danger and make it their responsibility to choose appropriate atoms.

The process of generating labels is also dangerous because it can produce side effects. Executing transitions and evaluating atoms both require running code from the target program. We can therefore of course make no guarantees about what that code may or may not do — it may access the network, databases, the filesystem, etc.. This problem is mitigated because the expected use case is running the model checker as part of a test suite. Test suites are often run

in sandboxed environments because tests have the same risk of damaging side effects. In our case though it may be less obvious what code the model checker will execute, so it is important that the user has some understanding of how it works

Results

To test this tool I used it to verify properties of state machines from two open source project. The goal here is to validate that the implementation works — that we can verify properties of existing state machines — not to validate that the tool provides actual value. Since we are verifying high-level properties of the programs (e.g. not looking for universal programming errors like "use after free") it is difficult to know what properties are useful to check without domain knowledge.

The first project I tested is the Spree "e-commerce solution", which has 942,000 downloads from RubyGems [9]. Spree is a natural example because, unlike many open source projects which provide general utilities, Spree implements "business logic", which state machines tend to be used for. Spree uses several state machines but I focused on that of the `Payment` class, consisting of 7 states and 16 transitions. To verify properties of this state machine I first added `state_machine_checker` as a dependency of the project and then added a file to the test suite containing seven CTL formulae to check, one of which was intended to fail.[11] For example:

```
it "can reach completed" do
  expect { new_payment }.to satisfy(EF(:completed?))
end
```

The only difficulty was implementing a `new_payment` method to provide a valid instance of the `Payment` class, as this required some understanding of application. Without modifying any of the source code of the application itself I was able to successfully run the model checker and it reported the expected results. Each formula took less than half a second to run on a laptop.

The second project I tested is the Eye "process monitoring tool", which has over 10 million downloads from RubyGems [10]. Here I examined the state machine for the `Process` class, which has 5 states and 23 transitions. I added a file to the test suite with four CTL formulae to check, including the following:[12]

[11] The full set of changes can be found at https://github.com/CJStadler/spree/commit/f470f371

[12] The full set of changes can be found at https://github.com/CJStadler/eye/commit/6958130

```
it "has no deadlock states" do
  f = neg EF(neg(EX(->(_) { true })))
  expect { new_process }.to satisfy(f)
end
```

Again, these ran successfully and returned the expected results, without requiring any modifications to the application source code. These checks each took less than 30 milliseconds to run.

Conclusion

`state_machine_checker` verifies properties of state machines which have already been written, without modifications, requiring the user only to write the CTL properties to check. In addition, it integrates seamlessly with existing tooling, making it familiar to developers and enabling continuous verification without requiring any changes to workflows or build pipelines. This demonstrates that formal verification techniques can be made accessible and potentially provide value to a large number of software projects. While this tool can provide only weak conclusions to users about the correctness of their programs, full verification is not a realistic goal for most programs and almost all could benefit from some amount of partial verification.

References

[1] R. W. Floyd, "Assigning meanings to programs," *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, vol. 19, Jan. 1967.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[3] G. J. Holzmann and M. H. Smith, "Software model checking: Extracting verification models from source code," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 65–79, Jun. 2001.

[4] T. Ball and R. Majumdar, "Automatic predicate abstraction of c programs," p. 11.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns : Elements of reusable object-oriented software*.

[6] C. Baier and J.-P. Katoen, *Principles of model checking*. Cambridge, Mass: The MIT Press, 2008.

[7] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of programs, workshop*,

1982, pp. 52–71.

[8] T. H. Cormen, Ed., *Introduction to algorithms*, 3rd ed. Cambridge, Mass: MIT Press, 2009.

[9] "Spree RubyGems.org your community gem host." [Online]. Available: https://rubygems.org/gems/spree. [Accessed: 23-Apr-2019].

[10] "Eye RubyGems.org your community gem host." [Online]. Available: https://rubygems.org/gems/eye. [Accessed: 23-Apr-2019].