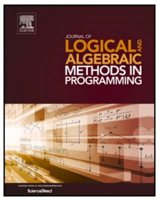
Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



Makina: A QuickCheck state machine library

Luis Eduardo Bueso de Barrio [*], Lars-Åke Fredlund, Ángel Herranz, Clara Benac-Earle, Julio Mariño

Universidad Politécnica de Madrid, Campus de Montegancedo s/n, 28660, Madrid, Spain

ARTICLE INFO

Keywords:
Property-based testing
State machines
Elixir
Model-based testing
Modular specifications

ABSTRACT

This article presents Makina, a library written in the Elixir programming language, and a domain specific language for writing property-based testing models for *stateful* programs. Models written in the domain specific language are translated into standard QuickCheck state machines. Our main goals with Makina are to facilitate the task of developing correct and maintainable models, and to encourage model reuse. To meet these goals, Makina provides a declarative syntax for defining model states and commands. In particular, Makina encourages the typing of specifications, and ensures that such type information can be used by Elixir type checking tools. Moreover, to promote model reuse, the domain specific language provides constructs that allow models to be defined in terms of collections of previously defined ones. To this end a number of operators for combining models have been defined and implemented in our library. A semantics for Makina models is presented in two steps. First, a novel operational semantics for standard QuickCheck state machine models is provided. Then, a translation from a Makina model to a standard QuickCheck state model is given.

1. Introduction

Property-based testing (PBT) [1] is a model-based testing technique in which tests are generated from *a test model*[1] to cover a multitude of scenarios that a human tester may not have considered. The model typically includes *generators* for generating an infinite number of values of some data type according to a probability distribution, and test *properties*. Given a set of generators for input data and a test property, a PBT tool generates a random instantiation of the variables (a test case), executes the system-under-test, and checks that the resulting property is true. For example, when testing a purely functional program $f : \tau \to \tau'$, a model may be obtained from a specification of the form $\forall x \in \tau \,.\, P(x, f(x))$ by providing a (random) generator for values in τ and an implementation of P. Often, we are interested in testing code with side effects rather than pure functions, e.g., when testing the implementation of an API, code with I/O, or code that communicates with other processes. In this case, a variation of the idea above called *stateful* property-based testing is often used instead. Basically, a stateful system can be seen as a set of *states* and a set of *commands* that induce transitions between states. When testing such a system using stateful PBT, the model will typically contain a *model state machine* M, and a generator of sequences of commands. Then, conformance between the actual transitions in the system-under-test, and the corresponding ones in M, must be checked.

* Corresponding author.
E-mail addresses: luiseduardo.bueso.debarrio@upm.es (L.E. Bueso de Barrio), larsake.fredlund@upm.es (L.-Å. Fredlund), angel.herranz@upm.es (Á. Herranz), cbenac@fi.upm.es (C. Benac-Earle), julio.marino@upm.es (J. Mariño).

[1] In this paper we will use the terms *test model* and *model* interchangeably.

https://doi.org/10.1016/j.jlamp.2025.101096
Received 10 March 2023; Received in revised form 2 October 2025; Accepted 7 October 2025

Available online 17 October 2025

The commercial Quviq QuickCheck property-based testing tool for Erlang [2] (EQC from now on) provides a library, `eqc_statem`, which facilitates writing test models for stateful code. A user of the `eqc_statem` library defines a set of commands corresponding to different interactions with the system-under-test. For example, when testing a typical Erlang module, the commands may correspond to the exported functions. An `eqc_statem` test model must provide the following information for each command: (i) a recipe for the generation of a command instance (e.g., deciding on the arguments for the corresponding function call), (ii) a method for how to execute the command on the system-under-test, and (iii) an oracle which decides if the execution of the command was correct, or whether a test failure should be signalled. The `eqc_statem` library has been used to test numerous challenging systems [3,4], and to test software not written in Erlang [5].

Note that, as this article introduces a new library for programming state machines in the Elixir programming language [6], we will be using Elixir syntax when presenting program code. The Elixir language is built on top of the Erlang run-time system, BEAM, and apart from changes in syntax, the languages are very similar.

"Functional-style" state machines In the original `eqc_statem` library, a user provided the command information in the form of the following callback functions:

```
command(state) :: call
precondition(state, call) :: boolean
postcondition(state, call, result) :: boolean
```

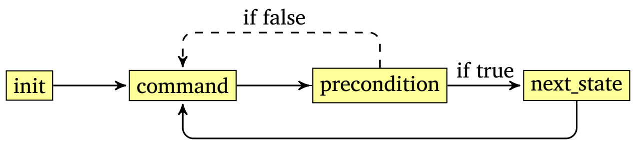
The function `command` is used for generating a "call" (a symbolic representation of a function call to the command, with arguments), the function `precondition` checks whether a call generated by the `command` function is acceptable in the test case being generated, and the function `postcondition` judges whether the call was executed correctly by examining the return value (`result`).

The `eqc_statem` library implements a state machine which interacts with these callback functions to generate test cases comprised of sequences of such calls, and to check whether the system-under-test executes the calls correctly. As the underlying system-under-test may be stateful, a *model state* is needed to both generate sensible sequences of calls (e.g., adding an element to a set is meaningful only after the set has been created), and to check whether the execution of a call was correct in the current model state. To manage the model state, a library user should implement the callback function

```
next_state(state,result,call)
```

which returns a new model state after the execution of a call in the previous model state (`state`). The parameter `result` is either a symbolic unbound variable (during test generation), or the concrete result of executing the command on the system-under-test (during test execution). Thus, the `next_state` callback is executed during both test generation and later during test execution.

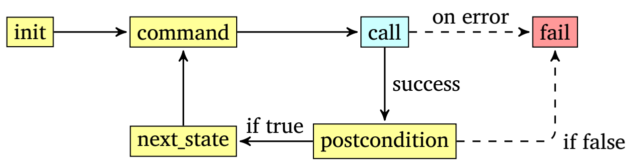
The following figures[2] represent, visually, the interaction between the aforementioned elements. The execution of a test occurs in two phases, the "symbolic" test generation phase and the "real" test execution phase. In the test generation phase, a sequence of abstract representations of calls (commands) is generated from an initial model state:



The preconditions are used to validate the command generated. If this validation fails (dashed arrow), a new command is generated until one is seen as valid. Once a suitable command is found, the state transition for this command is applied to the model state, and the next command is generated similarly. Note that the following model state after a command may refer to the result of executing the command. As the result is not known when the test case is generated, the result is represented by an unbound "result variable". As such variables may occur in the state, they can form part of subsequent generated commands too.

At the end of this process, a valid sequence of commands will have been generated. Such a sequence of commands is a test case, that may be in part symbolic, referring to unknown results from the execution of the test case.

Once a sequence of commands has been generated, the test execution phase can begin. It is in this phase where the results of executing the commands of the test sequence on the system-under-test are compared to the results predicated from the sequence of abstract calls:



2 Adapted from https://propertesting.com/book_stateful_properties.html.

The test can fail due to several reasons. The actual call may fail. For example, if the command executed is an Elixir function call, a failure occurs when the function call raises an exception that was not anticipated by the test "harness" (the test harness is responsible for catching any exception which is an acceptable command result). Secondly, the postcondition checks whether the result of executing the command is acceptable in the current state. If the postcondition is true, the state transition for the command is applied to the state, and the next command can be processed. Otherwise, the result is again a failure.

"Command-style" state machines The current version of the `eqc_statem` library permits writing test models using a different syntax. Instead of providing the above callback functions, a test model may contain, for each command `cmd`, the following callback functions:

```
cmd_pre(state)
cmd_args(state)
cmd_pre(state,args)
cmd_post(state,args,result)
cmd_next(state,result,args)
```

That is, instead of writing model code which groups the handling of commands on a functionality basis (generation, postconditions, next state computation), one writes model code which groups code on a command basis. This second style of writing test models permits focusing attention on commands rather than test functionality, which, in our opinion, leads to more legible test models.

In the following, we will refer to state machines (or, alternatively, models) which use the syntax in the original `eqc_statem` library as functional-style state machines (or models). State machines that use the newer syntax implemented in more recent versions of the `eqc_statem` library will be referred to as command-style state machines (or models).

Internally, recent versions of the `eqc_statem` library translate test models written in the command-style into models written in the functional-style using a *parse transform*. Informally, a parse transform is an Erlang function that receives an abstract syntax tree (AST) representation of an Erlang module as input and returns a transformed module as an AST.

Shortcomings of existing tools After years of applying PBT in various domains, we have identified some key difficulties that most users face when writing EQC state machines. First, and most importantly, test models are often buggy themselves. Some bugs in the models are type errors that might be caught by a type system. We believe that introducing type information into the models, and using static type analysis, would help detect some of those bugs before running the tests. Secondly, test models tend to be monolithic, combining different concerns, such as *test generation*, *execution*, and *correctness checking*. For example, suppose that we have written a test model and wish to check an additional correctness property. Either we embed the new property in the original test model, which then grows even larger and becomes harder to understand, or we create a new test model and copy large parts of the original model to the new one — but running the risk of introducing model bugs. In practice, experienced test model writers use various strategies to cope with model reuse, e.g., by using the Erlang source code include mechanism, or by configuring test strategies through embedding such meta information as part of the model state. In other words, we believe that there is a lack of support for *model composition* that would facilitate the effective reuse of test models in a clean way.

Makina: a library for writing maintainable models In this paper, we present Makina, a library for writing QuickCheck state machines in Elixir, which is licensed under a BSD licence. Makina has been designed to address the principles of modularity and model reuse discussed above, and to provide effective type checking of test models. Similarly to PropCheck [7], the Makina library uses the Elixir macro facility to provide a domain specific language for implementing command-style state machines. The result of macro processing is an Elixir/Erlang module implementing a functional-style state machine which can be used by both EQC and PropEr.

The Makina domain specific language for writing test models provides a number of major improvements:

Declarative syntax for commands. Commands are declared explicitly. We extend the approach in PropCheck by naming command parameters, and optionally typing them.

Declarative syntax for model states. In Makina a model state comprises a set of attributes, with optional types.

Access to command parameters and state attributes. Command parameters and state attributes are made available as predefined variables in callback functions. If, for instance, a command `cmd` has a parameter `value` then in e.g. the callback function `cmd_post` the variable `value` can be used without a declaration.

Typing. The typing information provided by declared commands and the model state is preserved by macros, and distributed to generated functions, to aid static type checking tools such as Dialyzer [8] and run-time type checking libraries such as Elixir TypeCheck [9], in effectively type checking test models. Even if programmers do not provide type annotations, type information about the structure of the state machine (such as the names of the state attributes and the commands) is derived from the expressions of the language. The derived type annotations help detect and diagnose subtle errors, such as missing parameters in a generated command or a function trying to access a non-existent state attribute.[3]

Composition of test models. State machines can be the composition of a number of state machines. Such functionality permits new styles of writing state machine test models. For example, we can write a composite test model which combines two sub-models: a sub-model which only generates commands, and another sub-model which judges the correctness of command execution. Another possible definition style separates the code for each command into its own sub-model. Note that, separating a model into sub-models, does not defeat the careful typing of machines.

[3] Note that when type annotations are missing, as we will see, the tool defaults them to `any()`, as Elixir currently has no built-in type inference capabilities.

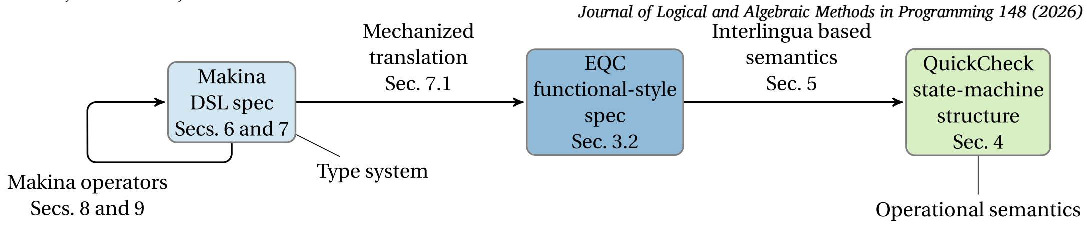

Makina DSL spec Secs. 6 and 7
Mechanized translation Sec. 7.1
EQC functional-style spec Sec. 3.2
Interlingua based semantics Sec. 5
QuickCheck state-machine structure Sec. 4
Makina operators Secs. 8 and 9
Type system
Operational semantics

Fig. 1. The main formalisms used in the article and their roles in the translation of Makina specifications into EQC state machines.

Semantics. The meaning of the Makina constructs is carefully defined through the combination of an operational semantics (explaining how EQC derives tests from a functional-style test model) and a translational semantics which describes how a Makina test model is translated into such a functional-style test model.

Cross-tool usability. Makina models are understood by both the EQC and the PropEr property-based testing tools; this is because state machine test models in both tools internally work with functional-style test models.

In [10], a first version of the Makina library was introduced. Since then, the language for defining state machine models has undergone changes which are documented in this article. Moreover, this article defines a semantics for both standard QuickCheck state machines (not previously published), and relates the Makina state machine models to the classical ones through a new carefully defined translation.

The Makina library is available at https://gitlab.com/babel-upm/makina/makina.git, and the repository https://gitlab.com/babel-upm/makina/examples contains a number of example models written in the Makina domain specific language.

Contributions The contribution of this paper is twofold. On the one hand, we present Makina's novel features in depth, largely extending the preliminary presentation in [10]. A practical example, used throughout the paper, will help us define the Makina domain specific language (DSL), and show the different composition and reuse features in action.

Second, a novel operational semantics framework for state machine-based PBT is provided. Being Makina a tool built on top of QuickCheck, a natural choice is to present its semantics just by means of a translation into QuickCheck. Unfortunately, QuickCheck and similar tools lack a formal semantic foundation, and users simply rely on the tools' documentation and examples to learn their use. Although this may suffice for everyday use, this is clearly insufficient for properly defining the different model composition operators in Makina. How can we be sure that, e.g. the composition of two models is well-defined? What kind of properties must be preserved?

Only after the EQC state machine models are given a well-defined semantics, the translation of Makina models into EQC and the different operators used to compose Makina models acquire a proper meaning. Fig. 1 shows the logical flow from the higher-level model specification defined in Makina to the final EQC machine produced. This translation is carried out as a series of stages in which different languages and formalisms are used: the Makina DSL for state machine models, EQC functional-style state machine models, and an abstract representation of a QuickCheck state machine. As mentioned before, the resulting model can also be used by the PropEr tool.

Article outline The following Section 2 contains a brief survey of related work. Section 3 introduces the running example used in the paper, a simple process-based API to store documents, along with an authentication API. An EQC state machine test model for the store API is presented using the so-called functional-style. Next, Section 4 develops an operational semantics for EQC state machine models, and Section 5 relates the concrete functional-style state machine library in EQC to the operational semantics. Then, Section 6 describes, informally, the domain specific language for test models introduced in the Makina library, which is inspired by the command-style state machines, and Section 7 formalizes syntax and semantics for this DSL. The following Section 8 introduces several mechanisms useful for extending and combining state machines, such as a *composition*, *extension*, etc., and Section 9 provides a formal semantics for these operations. Section 10 details a number of implementation concerns, e.g. discussing how we use the Elixir macro facility to transform state models effectively, while preserving type information. Next, Section 11 describes the experiences in applying the Makina DSL in a number of case studies, evaluating its capacity for reducing and detecting model bugs, and whether its state machine composition operators aid in model design. Finally, Section 12 discusses some issues for further research and development.

2. Related work

Property-based testing of purely functional code was originally introduced by *QuickCheck* for Haskell [1]. Later, EQC adapted the previous ideas to the Erlang programming language and introduced stateful PBT [2], which has also been added to the Haskell version. Since then, a number of PBT tools have appeared which contain support for writing stateful test models, e.g., *ScalaCheck* [11] for Scala, *Hypothesis* [12] for Python, *Hedgehog* [13] for Haskell, F#, R, and Scala, etc.

Quviq provided the `eqc_ex` library [14], which enables using EQC in Elixir.[4] Internally, the `eqc_ex` library relies on the use of so-called parse transforms to parse state machine models. Unfortunately, the support for parse transforms in Elixir is now deprecated, which discourages the writing of `eqc_statem` command-style state machines using the `eqc_ex` library.

A popular alternative to EQC is the PropEr property-based testing tool [15], which is available for free under a GPL version 3 licence. PropEr is written in Erlang too, and similarly to EQC provides a state machine library `proper_statem` for testing stateful systems, notable is for instance [16], which uses the PropEr finite state machine library to test sensor networks. The API of the PropEr state machine library is very similar to the EQC state machine library API; test models are written in the functional-style. PropEr users who wish to write test models in Elixir can use the PropCheck library [7], licensed under the GPL version 3 licence. Among other improvements, the library provides a new domain specific language for writing command-style state machines, which are translated into functional-style machines. Interestingly, this domain specific language is implemented using the Elixir macro facility and thus does not depend on the deprecated support for parse transforms. *PropCheck* enables the use of PropEr in Elixir. However, in contrast with Makina, the PropCheck tool does not permit the declaration of command parameters, and state components cannot be named (and later used in command code). Moreover, there are no state operators defined.

One of the chief motivations for this work is to improve maintainability of test models through providing better tools for detecting model errors caused by changes to the system-under-test. A complementary approach to increase the maintainability of the test model is represented by [17], which develops techniques for the automatic refactoring of test models due to changes in the system-under-test.

Although "state machine" and "finite state machine" are terms used in PBT libraries, they are more similar to abstract state machines [18] than the classical finite state machines from theoretical computer science [19], since arbitrary data structures are allowed in states. Our description of the testing models is quite similar to the statecharts proposed by Harel [20] and their UML incarnation (state diagram). There is also a connection – especially in how we define the composition of state machines – with various process algebras, e.g., LOTOS [21] and ACP [22].

One of the fields where the composition of state machines has been studied is embedded systems. In the classification of the composition of the (extended) state machines given in [23], our composition constructs could be classified as a combination of *side-by-side asynchronous composition* with *cascade composition*.

In the design of the composition constructs, we have been guided by some design principles from object-oriented programming. Maybe the most relevant work on the application of object-oriented practices to the composition of state machines is [24]. In that article incremental constructs permit the description of complex state machines from simple ones. Among these techniques, *subclassing*, *composition*, and *delegation* have been used and adapted when designing our incremental approach to machine definition. Note that although EQC does provide some support for composition of state machine models by using the EQC libraries `eqc_cluster` and `eqc_component` [25], using these libraries requires mastering a new domain specific language for expressing state machine interactions. In contrast, Makina provides a mechanism for *extending* a model with another model, solely based in the callbacks of the state machine formalism itself.

Finally, our approach to model extension owes some inspiration to the Liskov-Wing behavioural notion of subtyping [26], i.e., we expect the behaviour of the resulting model to be somehow *consistent* with the original one.

3. QuickCheck state machines

We will use a simple stateful API service for storing documents as a running example throughout this paper. The store API, in module `Docs`, provides three functions:

```
put(token, key, doc) :: :ok | :error # stores a document
del(token, key) :: :ok | :error      # deletes a document
get(key) :: {:ok, doc} | :error      # retrieves a document
```

The function `put/3` allows a user to store a document `doc` under a `key`. The function returns `:ok` if the document is stored and `:error` otherwise, e.g., if the key is already used for another document. The function `del/2` deletes a previously stored document, while `get/1` retrieves the document stored under `key` – or `:error` if there is no such document.

In the usual way, access to API functions that destructively change the state of the service is restricted using a credential, a token. Such tokens can be generated using `Auth`, an authentication API:

```
reg(user, pass) :: :ok | :error            # registers a user
gen(user, pass) :: {:ok, token()} | :error # generates a token
rev(token) :: :ok | :error                 # revokes a token
val(token) :: :ok | :error                 # validates a token
```

where `reg/2` is used to register a new user. After registering a user, a new access token can be generated using `gen/2`, which requires a password argument. Finally, the functions `rev/1` and `val/1` are used to revoke and validate a token, respectively. Note that `Docs` relies on `Auth` to work. If a user does not specify a valid token, the upload and deleting operations should fail.

When testing a stateful API, we need to ensure that the execution of a command of the API has the intended effect in the internal state of the system-under-test. To accomplish this, in traditional unit testing we usually write sequences of commands that lead to some desirable state. A desirable property for the `Docs` API is that a user cannot use the same key to store two documents, unless

[4] Since EQC uses Erlang macros it was necessary to re-implement these as Elixir macros.

the first document had previously been deleted. To check such a property, we can, for example, write the following sequence of commands (a unit test case):

```
:ok           = reg(user, pass)       # registers a user
{:ok, token} = gen(user, pass)        # generates a token
:ok           = put(token, key, doc1) # stores doc1 in key
{:ok, doc1}  = get(key)               # retrieves doc1
:error        = put(token, key, doc2) # fails to store doc1
:ok           = del(token, key)       # removes doc1
:ok           = put(token, key, doc2) # stores doc2
```

Checking APIs by writing such unit test cases manually has a major drawback. It is very difficult to think of every possible combination of API calls that could exhibit bugs, and moreover, writing manually the large number of needed unit test cases will be a very tedious and time-consuming task. Property-based testing offers the promise to automate part of the task of testing such an API, but the challenge here is to ensure that all relevant test cases can be generated and tested. An interesting comparison on the benefits and drawbacks of using property-based testing compared to other testing techniques can be found in [27].

3.1. *Property-based testing*

Property-based testing (PBT) is a model-based testing technique that uses a model to cover a multitude of scenarios. A PBT test model or property is concerned with: *data generation*, *test execution*, *property checking*, and generating *short counterexamples* when the property is falsified. As an initial example, imagine that we want to test some identity function `id`:

```
forall x in term() do id(x) == x end
```

where `term()` is a generator that can potentially generate any valid Elixir value, and `id(x) == x` is the postcondition that we are trying to check. We can extract values from a generator using the function `pick`, e.g. `pick(term())` generates values like `[]`, `:a` or `3`. If any of the values taken by `x` falsifies the postcondition, the test fails, and the PBT system will, starting from the counterexample found, try to generate an alternative, but smaller, and easier to understand counterexample. The search for such small counterexamples is performed using a procedure called *shrinking*, which aims to systematically reduce the size and complexity of the generated test data. For instance, if the counterexample is a list of items, the shrinking procedure will attempt to delete items from the list and check if the resulting smaller list still fails the postcondition.

In stateful PBT, the generated test data is a sequence of commands, and the property to check is that the execution of the test data on the system-under-test conforms with the expected result of executing the test data according to the test model.

The model used to generate sequences of commands and check the conformance of the system-under-test is a QuickCheck state machine [2]. Concretely, we can specify the conformance property as follows:

```
forall cmds in commands(model) do run_commands(model, cmds) == :ok end
```

where `model` is the state machine, `commands(model)` generates sequences of commands `cmds` using the state machine `model`, and `run_commands(model, cmds)` executes the generated command sequence `cmds` against the system-under-test using the state machine `model`. The invocation of `run_commands(model, cmds)` returns `:ok` when the sequence runs as expected, i.e., when the result of executing a command under the system-under-test conforms to the prediction made by the state machine.

This property states that, for every generated sequence, the result of running the commands against the system under test is correct. In traditional property-based testing, the property encodes both the generator and the postcondition. However, when using QuickCheck state machines, the property is more general, and generation and correctness concerns are encoded in the state machine itself. Note that QuickCheck state machines run twice: first to generate a test case and secondly to run the generated test case against the system under test.

Test case generation The state machine is run first during test case generation to generate a complete sequence of commands (a test case), *without interacting with any system-under-test*. As the results of executing generated commands are thus unknown during this first state machine run, QuickCheck instead replaces them with symbolic variables, which are similar to e.g. future objects in some language constructs for concurrent programming.

However, subsequent calls to the tested API may have to refer to the results of previous API calls, which are represented as symbolic variables accordingly. In our running example, for example, to issue a correct call to the `rev` or `val` functions, a token is needed, which is returned by an earlier call to `gen`.

Thus, note that although the state machine contains code for checking that results are correct (e.g., that retrieving a document using `get(key)` really returns a document), such checks are not executed during test case generation as only a symbolic variable represents the result.

In QuickCheck the `commands/1` generator generates command sequences (in simplified notation) such as the one depicted in Fig. 2where `var1`, ..., `var7` are symbolic variables that represent the future results of the execution of a command under a system-under-test.

Moreover, note, for instance, that the symbolic variable `var2` representing the result of the call to `gen` on line 2 is used as a parameter (the token) in the call to `put` on line 3. However, since the call to `gen` is expected to return a tuple `{:ok, token}` the token must be retrieved from such a tuple; this corresponds to the call `elem(var2,1)` on line 3. Calls to functions where parameters

```
var1 = reg(user, pass)
var2 = gen(user, pass)
var3 = put(elem(var2, 1), key, doc1) # elem/2 extracts the token
var4 = get(key)
var5 = put(elem(var2, 1), key, doc2)
var6 = del(elem(var2, 1), key)
var7 = put(elem(var2, 1), key, doc2)
```

Fig. 2. A symbolic test case generated by `commands/1`.

contain symbolic variables will be referred to as "symbolic expressions". Such symbolic expressions should not be evaluated during test generation, but only when actually running the generated test against a system-under-test (the second run of the state machine), when symbolic variables will be replaced with the concrete result of executing a command on the system-under-test.

Thus, the above test case uses both symbolic variables, e.g. `var2`, and symbolic expressions, e.g., `elem(var2,1)`. Note, moreover, that typically such symbolic results (or symbolic expressions) will migrate to the test model state, too. For example, in the authorization example we probably want to record in the model state the fact that a generated (symbolic) token corresponds to a particular user.

Running a test case Next, the QuickCheck state machine is rerun guided by the generated test case, in the process executing the generated test case commands against the system-under-test. During the running of the generated test case, which is expected to contain symbolic variables and symbolic expressions, a symbolic variable representing the unknown result of a command is replaced with the actual result of executing the command on the system-under-test. In addition, now that symbolic variables have been instantiated, symbolic expressions (code fragments with symbolic variables as parameters) can be evaluated. Moreover, the actual results of executing the command under the system-under-test are verified (e.g., that retrieving a document using `get(key)` really returns a document), to ensure that results coincide with the predictions made by the state machine.

Understanding the above execution model, i.e., that test case generation precedes test case execution, and its implications for writing test models that involve computing with results that will become known only at execution time, is in our experience one of the principal difficulties that novice developers of PBT state machine models face. Clarifying the exact semantics of this execution model is one of the principal aims of this article.

So far we have seen examples of the type of sequences of commands that a QuickCheck state machine should be able to generate. To accomplish this, a state-machine model needs, intuitively, some internal state to keep track of the changes performed by each command; a generator function that creates new commands from a given state; a transition function that modifies the inner state after a command is executed; and a predicate that checks the correctness of the result given by the system-under-test. In the following section we introduce, using an example, a concrete library for coding such state-machine models.

3.2. *Functional-style state machines*

In Fig. 3, we show a real EQC state machine, implemented in Elixir for the authentication example, programmed according to the functional-style. The state is represented using a tuple that stores a map and a list. The map stores the user information where each key is a username and each value is its password, and the list stores the currently active access tokens. We assume the existence of a number of generator functions, i.e. `string()` generates a random string, `pos_integer()` generates a random positive integer, and `oneof(L)`, where `L` is a list of generators, selects randomly one of the generators in `L` and generates a value using that generator. The `let` var `<-` gen_1`, do:` gen_2 construct generates a value using the generator gen_1, assigns that value to the variable var, and generates a new value using the generator gen_2 (normally `var` occurs in gen_2). Note also that any Elixir value is a generator, too, which can only generate itself. A list $[v_1, \dots, v_n]$ is a generator of a list; its elements are generated by the generators $v_1, \dots, v_n$ which may be Elixir values or more complex generators.

Recall from Section 1 the callback functions that a functional-style model must define. The `command` callback is called with the model state as an argument and should generate a command to execute. Commands are represented as tuples with four elements, the first element being the atom `:call`, and the following three elements should be a module name, the name of a function, and the arguments for the function to call. For example `{:call, Auth, :reg, [user, pass]}` represents a command to register a user. In the definition of `command`, beginning on line 3, we see that either there are no users in the state, and then only the `reg` command can be generated, or a call to any of the commands (`reg`,`gen`,`rev` and `val`) is generated randomly.

The precondition (defined in `precondition`) only permits `gen` commands when its user and password pair argument is present in the model state. The next state of the model is computed via a call to `next_state`. Note, for instance, that on line 31 the token is extracted from a result using the code fragment `token = {:call, Kernel, :elem, [result, 1]}`. Finally, the result of executing a command is checked in the `postcondition` function, which verifies that the result of the call (in the parameter `result`) is consistent with the expected result, which is computed from the model state and the generated call.

Note that, even though the program is trivial, and the model is not that complicated, there is a lot of boilerplate code. We have to remember, for example, the arguments of commands everywhere, and we have to pack/unpack calls to the system-under-test inside `{:call, m, f, args}` tuples.

```
defmodule Auth.Functional do
  def initial_state(), do: { %{}, [] }
  def command({users, tokens}) do
    if users == %{} do
      {:call, Auth, :reg, [string(), string()]}
    else
      user_pass = let user <- oneof(Map.keys(users)),
                  do: [user, Map.get(users, user)]
      token = oneof([pos_integer()|tokens])
      oneof([
        {:call, Auth, :reg, [string(), string()]},
        {:call, Auth, :reg, user_pass},
        {:call, Auth, :gen, user_pass},
        {:call, Auth, :rev, [token]},
        {:call, Auth, :val, [token]}
      ])
    end
  end
  def precondition({users, _tokens}, call) do
    case call do
      {:call, Auth, :gen, [user, pass]} -> {user, pass} in users
      _ -> true
    end
  end
  def next_state(state = {users, tokens}, result, call) do
    case call do
      {:call, Auth, :reg, [user, pass]}  ->
        if user in users, do: state,
        else: {Map.put(users, user, pass), tokens}
      {:call, Auth, :gen, [user, pass]} ->
        token = {:call, Kernel, :elem, [result, 1]}
        unless {user, pass} in users, do: state,
        else: {users, [token|tokens]}
      {:call, Auth, :rev, [token]} ->
        unless token in tokens, do: state,
        else: {users, List.delete(tokens, token)}
      _ -> state
    end
  end
  def postcondition({users, tokens}, call, result) do
    case call do
      {:call, Auth, :reg, [user, pass]} ->
        if user in Map.keys(users), do: result == :error,
        else: result == :ok
      {:call, Auth, :gen, [user, pass]} ->
        if {user, pass} in users, do: match?({:ok, _}, result),
        else: result == :error
      {:call, Auth, :rev, [token]} ->
        if token in tokens, do: result == :ok,
        else: result == :error
      {:call, Auth, :val, [token]} ->
        unless token in tokens, do: result == :error,
        else: result == :ok
      _ -> true
    end
  end
end
```

Fig. 3. The store model implemented in the functional-style.

4. An operational semantics for QuickCheck state machines

This section defines an operational semantics that formally describes how QuickCheck state machines are executed. The stateful API `Auth` introduced in Sect. 3 will be used to illustrate the semantics constructs.

We will define QuickCheck semantics independently of the programming language and based on term algebrae.

Definition 4.1. The following definitions and notational conventions from [28] will be used in the article:

- A ***term algebra*** $T(\Sigma, X)$ is a freely generated algebraic structure over a given signature Σ (with ***function symbols***) and a set of ***variables*** X.

- Elements of $T(\Sigma, X)$ are ***terms***.
- The structure of a term is a tree, let strings of positive integers refer to ***positions*** in a term and let the empty string ϵ refer to the ***root position***.
- A ***substitution*** is a function $\phi : X \to T(\Sigma, X)$.
- Let $\textsc{vars}(t)$ denote the set of ***variables*** in term t.
- A term t is ***ground*** iff $\textsc{vars}(t) = \emptyset$.
- $T(\Sigma, \emptyset)$ is the set of ***ground terms*** of $T(\Sigma, X)$. Let $T(\Sigma)$ refer to $T(\Sigma, \emptyset)$.
- Substitutions can be **extended** to functions on terms: let $\phi(t)$ refer to the term resulting from substituting every occurrence of every variable ψ in t with the term $\phi(\psi)$.

Notation 4.1. The following notational conventions regarding substitutions are used:

- Let [] refer to the *identity* substitution (maps a variable x to itself),
- let $[x \leftarrow t]$ refer to the substitution that maps x to t and any *other* variable x' to itself,
- and let $[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n]$ refer to $[x_n \leftarrow t_n] \circ \cdots \circ [x_1 \leftarrow t_1]$.

Example 4.1 *(Elixir literals).* In spite of the significant differences between the term algebra, mathematical notation, and the syntax of a programming language, conceptually, the literals in the programming language are terms of a term algebra. In the case of Elixir, the signature of values is given by the following constructs:

1. a constant symbol (literals as atoms, strings, and numbers) is a term,
2. `!!` is a term (the empty Elixir tuple),
3. if $t_1 \dots t_n$ are terms, then the Elixir tuple `{`t_1`, ..., `t_n`}` is a term (*syntactic sugar* for $\{\}_n(t_1, \dots, t_n)$, where "$\{\}_n$" is a n-ary function symbol),
4. `[]` is a term (the empty list),
5. if t_1 and t_2 are terms, then non-empty list `[`t_1`|`t_2`]` is a term (*syntactic sugar* for `[|]`(t_1, t_2), where "`[|]`" is a binary function symbol), and
6. if $t_{11} \dots t_{1n}$ are terms and $t_{21} \dots t_{2n}$ are terms, then the map `%{`t_{11}` => `t_{21}`, ..., `t_{1n}` => `t_{2n}`}` is a term (*syntactic sugar* for `%{}`$_n(t_{11}, t_{21} \dots, t_{1n}, t_{2n})$, where "`%{}`$_n$" is a $2n$-ary function symbol).

Example 4.2 *(Elixir functions).* Elixir function names are included among the function symbols of the term algebra:

1. if $t_1 \dots t_n$ are terms, and f is an Elixir function name with arity n, then f_n`(`t_1`, ..., `t_n`)` is a term (observe that overloaded function names have a different function symbol in the term algebra).
2. qualified function names are the actual function symbols in the term algebra, e.g., "`Map.put`" is a function symbol.

Note: To avoid the additional complexity associated with representing lambda expressions directly within the term algebra, we assume that all lambda expressions are lifted [29]. This allows us to treat them as first-class citizens without introducing variable binding mechanisms and scoping rules into the algebra itself.

Notation 4.2. Let $\mathbb{N}$ refer to the nonnegative integer numbers and let $\mathbb{N}^+$ refer to $\mathbb{N}\backslash\{0\}$ (positive integers). Moreover, given a set A, let $\mathbf{2}^A$ refer to the *powerset* of A.

Definition 4.2. Given a set A, the **sequences of** A is the free monoid generated by A. Let A^* (the Kleene star) refer to the sequences of A. Let A^n refer to the sequences of A of length n.

Observe that there is a one-to-one correspondence between the elements of the n-fold Cartesian product and the sequences of A of length n, so let A^n denote also the Cartesian product $\underbrace{A \times \cdots \times A}_{n}$ too.

Notation 4.3. The following notational conventions for sequences are used:

- given a sequence s, $|s|$ denotes its length,
- given a sequence s and an integer i such that $1 \leq i \leq |s|$, s_i denotes the i-th element of the sequence s,
- ϵ is the empty sequence,
- given two sequences s and s', then ss' and $s \cdot s'$ both denote the sequence obtained by concatenating the items in s and s' in order,
- concatenations $x_1 x_2 \dots x_n$ and $x_1 \cdot x_2 \cdot \dots \cdot x_n$, and the tuple $\langle x_1, x_2 \dots, x_n \rangle$, represent the same sequence, and
- given a sequence s, we write $x \in s$ if and only if x is an element of the sequence.

4.1. Formal definition of a QuickCheck property

Definition 4.3. A ***QuickCheck property*** consists of:

1. a *signature* Σ and its *induced term algebra* $T(\Sigma)$,
2. a *generator* $\mathbf{G} \in \mathbf{2}^{T(\Sigma)}$ (a set of terms),[5]
3. a *correctness predicate* $\mathbf{P} \subseteq T(\Sigma) \times T(\Sigma)$ (pairs argument/result),
4. a *shrinking function* $S : \mathbf{G} \to \mathbf{2}^{\mathbf{G}}$ (maps generated arguments to sets of *generable* arguments)

QuickCheck properties are used to *test* functions. The generator contains possible inputs and the correctness predicate is used to check their corresponding output values. Once a test fails, the shrinking function is used to find a *simpler*[6] input. The shrinking function works on terms in the generator set, and ensures that the set returned by the shrinking function is a subset of the generator set. Note that, the returned set may well be equal to the generator set, if, for instance, a generator just generates a single constant value.

Notation 4.4. Given a QuickCheck property p, let $\mathbf{G}, \mathbf{P}$ and S refer to its components. When referring to multiple properties, we use $\mathbf{G}_p$, $\mathbf{P}_p$, and S_p.

Example 4.3 *(QuickCheck property).* One possible QuickCheck property to check a function that adds 1 to an integer is:

- Σ is the set of all Elixir terms (see Example 4.1),
- $\mathbf{G} = \{\texttt{-2}, \texttt{-1}, \texttt{-0}, \texttt{1}, \texttt{2}\}$
- $\mathbf{P}(t, t') = \{(\texttt{-2}, \texttt{-1}), (\texttt{-1}, \texttt{0}), (\texttt{0}, \texttt{1}), (\texttt{1}, \texttt{2}), (\texttt{2}, \texttt{3})\}$,
- the shrinking function $S(t) = \{\texttt{-2} \mapsto \{\texttt{-1}\},\ \texttt{-1} \mapsto \{\texttt{-1}\},\ \texttt{0} \mapsto \{\texttt{0}\},\ \texttt{1} \mapsto \{\texttt{0}\},\ \texttt{2} \mapsto \{\texttt{1}\}\}$

Definition 4.4. Given a QuickCheck property p and a system-under-test $f : T(\Sigma) \to \mathbf{2}^{T(\Sigma)}$, we say that the property p ***holds in the system-under-test*** f if and only if for every term $t \in \mathbf{G}$, and for every result $r \in f(t)$, the predicate $\mathbf{P}(t, r)$ holds.

Note that the system-under-test may be non-deterministic, hence the function representing it returns a set of results ($\mathbf{2}^{T(\Sigma)}$) for each input.

For completeness, we introduce Definition 4.5:

Definition 4.5. Given a QuickCheck property p and a system-under-test f, procedure *check*(p,f) checks that the system-under-test executes a number of tests generated by the property p correctly. The procedure either returns `none` signalling that no test failed, or `some` t indicating that t is a counterexample.

```
function $check(p, f)$
    $n \leftarrow 1$
    while $n \leq MAX\_TESTS$ do
        $t \in \mathbf{G}_p$                                   ▷ generate test
        $r \in f(t)$                                   ▷ execute test
        if $\neg \mathbf{P}_p(t, r)$ then
            return $shrink(p, f, t, r)$                ▷ test failed, shrink it
        $n \leftarrow n + 1$
    return none
```

The procedure generates at most *MAX_TESTS* values. If the system-under-test f executes a test t incorrectly, a second procedure *shrink* attempts to shrink the counterexample t.

```
function $shrink(p, f, t, r)$
    $n \leftarrow 1$
    while $n \leq MAX\_SHRINKS$ do
        for $t' \in S_p(t)$ do                         ▷ shrink test
            if $\neg \mathbf{P}_p(t', r)$ then
                $t \leftarrow t'$
                break                                  ▷ shrunk test failed, shrink it
        $n \leftarrow n + 1$
    return some $t$
```

[5] Actual generators are much more complex, possibly using stochastic methods to produce data according to some programmer defined probability distribution, among other details. In this paper we use sets as an abstraction for the sake of simplicity.

[6] Deciding what constitutes a simpler input is in general a non-trivial task, as it is related to the hard question which test case makes it easier for a human to find the source of an observed error. For this reason we will not elaborate further on shrinking functions in this article.

The shrinking function shrinks the counterexample t, checking if a shrunken test t' is also executed incorrectly by the system-under-test. If that is the case, the new counterexample t' is further shrunk.

Note that the function assumes that the system-under-test has deterministic failures, i.e., a test case either always fails or never fails; an alternative is to repeat the execution of a test case a number of times.

Definition 4.6. Given two QuickCheck properties p and p' and system-under-test $f : T(\Sigma) \rightarrow \mathbf{2}^{T(\Sigma)}$, we say that p' ***refines*** p when:

$$\begin{array}{l}\mathbf{G}_p \subseteq \mathbf{G}_{p'} \\ \forall t \in \mathbf{G}_p,\ r \in f(t)\ \ .\ \ \mathbf{P}_{p'}(t,r) \implies \mathbf{P}_p(t,r)\end{array}$$

This definition establishes an order relation between properties. When a property p' refines a property p, the generator in p' generates at least all terms in p; and for every term t in the generator in p the postcondition in p holds if it holds in p'. Thus, the meaning of property refinement is twofold; the refined property generates at least the same tests as the base property and can make the postcondition stronger by rejecting more results.

Notation 4.5. Let $p \sqsubseteq p'$ refer to p' refines p.

4.2. Formal definition of QuickCheck state machine

Definition 4.7. A ***QuickCheck state machine*** consists of:

1. a *signature* $\Sigma = \Sigma_d \uplus \Sigma_f \uplus \Sigma_c$ (the union of disjoint signatures for *data constructors* Σ_d, *symbolic functions* Σ_f, and *command names* Σ_c), a countable set of *symbolic variables* $\mathbf{V}$, and the induced *term algebra* $T(\Sigma, \mathbf{V})$,
2. a set of *states* $\mathbf{S}$ (a subset of terms with no command names),[7]
3. a set of *commands* $\mathbf{\Lambda}$ (a subset of terms with just one command name in the root position),
4. a *transition function* $\delta : \mathbf{S} \times \mathbf{\Lambda} \times \mathbf{V} \rightarrow \mathbf{S}$ (maps tuples of states, commands and variables to states),
5. a *generator function* $\mathbf{G} : \mathbf{S} \rightarrow \mathbf{2}^{\mathbf{\Lambda}}$ (maps states to sets of commands),
6. a *postcondition predicate* $\mathbf{P}$ that is a subset of[8]
$$\mathbf{S} \cap T(\Sigma_d)\ \times\ \mathbf{\Lambda} \cap T(\Sigma_{cd})\ \times\ T(\Sigma_d)$$
(tuples of ground states, ground commands and ground terms, without symbolic functions),
7. an *initial state* $\mathbf{s_0}$ that is a term in $\mathbf{S} \cap T(\Sigma_d)$ (a ground state with no symbolic functions),
8. and an *interpretation function* $[\![\_]\!] : \mathbf{S} \cap T(\Sigma_{df}) \rightarrow \mathbf{S} \cap T(\Sigma_d)$ (maps ground states with symbolic function applications into ground states with no symbolic function applications).

The third argument in the transition function δ represents the (unknown) result of the executed command. It is assumed that δ and $\mathbf{G}$ cannot "invent" new variables, i.e., we restrict the generator and transition functions to ones satisfying the following conditions:

$$\begin{array}{l}\forall s, \alpha, \psi, s'\ \ .\ \ \delta(s, \alpha, \psi) = s' \implies \text{VARS}(s') \subseteq \text{VARS}(s) \cup \{\psi\} \\ \forall s, \alpha\ \ .\ \ \alpha \in \mathbf{G}(s) \implies \text{VARS}(\alpha) \subseteq \text{VARS}(s)\end{array}$$

In addition, the interpretation function cannot interpret data constructors:

$$\forall d \in T(\Sigma_d)\ .\ [\![d(t_1, \ldots, t_n)]\!] = d([\![t_1]\!], \ldots, [\![t_n]\!])$$

Notation 4.6. Given a QuickCheck state machine, let Σ, Σ_d, Σ_f, Σ_c, $\mathbf{V}$, $\mathbf{S}$, $\mathbf{\Lambda}$, δ, $\mathbf{P}$, $\mathbf{s_0}$, and $[\![\_]\!]$ refer to its components.

Next, let us define some sanity checks on the transition and generator functions:

Definition 4.8. A QuickCheck state machine is ***non-terminating*** if for all $s \in \mathbf{S}$, the property $\mathbf{G}(s) \neq \emptyset$ holds.

This definition shows that, for every state, the generator function should always be able to generate new command calls. One of the most common problems when writing QuickCheck state machines arises when a state machine *terminates* and is not capable of generating new command calls, which results in an execution error.

[7] Avoiding command names in state helps to distinguish commands from data in the formalization.

[8] $\Sigma_{cd} = \Sigma_c \cup \Sigma_d$ and $\Sigma_{df} = \Sigma_d \cup \Sigma_f$.

Definition 4.9. A QuickCheck state machine is ***non-inspective*** if for every state $s \in \mathbf{S}$, command $\alpha \in \mathbf{\Lambda}$, symbolic variable $\psi \in \mathbf{V}$, and substitution $\phi : \mathbf{V} \to T(\Sigma_d)$ the following properties hold:

$\phi(\delta(s, \alpha, \psi)) = \delta(\phi(s), \phi(\alpha), \psi)$, and
$\alpha \in \mathbf{G}(s) \iff \phi(\alpha) \in \mathbf{G}(\phi(s))$

Intuitively, the definition expresses that the transition and generator functions must not inspect or interpret terms that might contain symbolic variables in any way. For novice QuickCheck state-machine programmers, understanding the subtle issues that arise when dealing with unknown result values is quite difficult; this sometimes leads to incorrectly defined transition functions or generators. Checking for "inspective" transition functions or generators can help prevent obscure model bugs.

As a practical example, suppose we represent the state of the state machine as a map and store the command results as keys in the transition function. This implementation of the transition function does not satisfy the inspective condition above. Since each command result is encoded by a distinct variable, a test case of size n yields a state map with exactly n entries. In contrast, when a substitution ϕ maps several result variables to the same value, the final size map is strictly smaller than n. Moreover, it is clear that such a transition function might well result in practical testing problems too, e.g., by failing to generate some desirable test case.

Example 4.4 *(Command names).* In our running example the command names of the QuickCheck state machine alphabet Σ_c are `reg`, `gen`, `rev`, `val` with the following arities: ar(`reg`) = 2, ar(`gen`) = 2, ar(`rev`) = 1 and ar(`val`) = 1.

Example 4.5 *(State machine).* For our running example, the corresponding QuickCheck state machine is:

- Σ_d is the set of all Elixir constructors (see Example 4.1), Σ_f is the set of all function symbols accessible from the module of the running example (e.g. `Map.put/3`, `elem/2` or `List.delete/2`), and Σ_c is `reg`, `gen`, `rev`, `val` (the commands)
- $\mathbf{V}$ is a countable set of variables,
- $\mathbf{S}$ are pairs {*users*, *tokens*} where *users* is a map with registered users and passwords and *tokens* is a list of valid tokens.
- $\mathbf{\Lambda}$ is the set of all commands of the form `reg(`*user*`,`*pass*`)`, `gen(`*user*`,`*pass*`)`, `val(`*token*`)`, and `rev(`*token*`)` where *user*, *pass*, and *token* are terms that represent users, passwords and tokens.
- δ updates the state of the model when the `reg`, `gen` or `rev` commands are invoked with correct parameters, and it is left unchanged otherwise:
  δ({*users*, *tokens*},`reg(`*user*`,`*pass*`)`,t) =
  {{`Map.put(`*users*`, `*user*`, `*pass*`)`}`, `*tokens*}
  when *user* is not in `Map.keys(`*users*`)`
  δ({*users*, *tokens*},`gen(`*user*`,`*pass*`)`,t) =
  {*users*`, [elem(`t`,1) | `*tokens*`]`}
  when *users*`[`*user*`]` evaluates to *pass*
  δ({*users*, *tokens*},`rev(`*token*`)`,t) =
  {*users*`,List.delete(`*tokens*`, `*token*`)`}
  $\delta(s, \alpha, t) = s$ otherwise
  Note that in the second clause, corresponding to the case when a token is generated for an existing user with the correct password, the transition function stores a function term `elem(`t`,1)`. This function term, represents a symbolic expression, thus will be evaluated when the state machine is used to check the execution of a system-under-test, and will retrieve the token from the result `{:ok, token}` of applying the `gen` command.
- $\mathbf{G}$ *generates* commands that should execute successfully as well as commands that *fail*. $\mathbf{G}$ receives a state {*users*, *tokens*} as argument, and which with varying probabilities (not shown) will generate a command according to the following choices:

`reg(`*user*`,`*pass*`)`	where *user* and *pass* are strings
`gen(`*user*`,`*pass*`)`	where *user* and *pass* are strings, or *users*`[`*user*`]` evaluates to *pass*
`val(`*token*`)`	where *token* is integer, or *token* is in *tokens*
`rev(`*token*`)`	where *token* is integer, or *token* is in *tokens*

  Since the `gen` (generate token) command requires the username and password combination to have been previously registered (using a `reg` command), choosing an arbitrary username and password is likely to generate a `gen` command that will be rejected by the system-under-test. Thus, a `gen` command is generated either with an arbitrary username and an arbitrary password, or is generated from a username and password combination present in the model state, corresponding to generating both positive and negative test cases. As shown above, the transition function δ applied to a `reg` command stores its arguments in the model state.
- $\mathbf{P}$ establishes the relation between a state, a command and the expected response from the system-under-test. Thus, first the postcondition predicate checks that the state meets some requirement for a given command (e.g., in order to register a user, that user cannot be already registered). If that requirement is fulfilled the system-under-test must return `:ok`, and otherwise it should

return `:error`. Below a schematic definition of the postcondition predicate is provided, where each command type is treated in a separate clause:

$$\begin{array}{l}
\mathbf{P}(\{\mathit{users}, \mathit{tokens}\}, \mathtt{reg}(u,p),\ r) \iff \{u,p\} \in \mathit{users} \wedge r = \mathtt{:error} \vee r = \mathtt{:ok} \\
\mathbf{P}(\{\mathit{users}, \mathit{tokens}\}, \mathtt{gen}(u,p),\ r) \iff \{u,p\} \in \mathit{users} \wedge \mathtt{elem}(r,0) = \mathtt{:ok} \vee r = \mathtt{:error} \\
\mathbf{P}(\{\mathit{users}, \mathit{tokens}\}, \mathtt{val}(t),\ r) \iff t \in \mathit{tokens} \wedge r = \mathtt{:ok} \vee r = \mathtt{:error} \\
\mathbf{P}(\{\mathit{users}, \mathit{tokens}\}, \mathtt{rev}(t),\ r) \iff t \in \mathit{tokens} \wedge r = \mathtt{:ok} \vee r = \mathtt{:error}
\end{array}$$

- $\mathbf{s_0}$ is `{%{}, []}`,
- And the interpretation function $[\![\_]\!]$ will interpret function symbols according to the operational semantics of Elixir. Note that no such semantics currently exists. However, the Elixir programming language is strongly related to the Erlang programming language for which a number of operational semantics have been developed, e.g. in [30] and [31].

After introducing the static semantics of QuickCheck state machines, a presentation of the dynamic semantics follows, paying attention to the following functionalities: (i) generating test cases, i.e. sequences of symbolic commands (in Definitions 4.11 and 4.13), and (ii) determining if an execution by a system-under-test is permitted by the state machine (Definition 4.16).

4.3. Formal definition of a test run of a QuickCheck state machine

Definition 4.10. Given a QuickCheck state machine, a ***one-step transition***

$$s \xrightarrow[\alpha]{\psi} s'$$

represents that $\delta(s, \alpha, \psi) = s'$.

Definition 4.11. A ***test run of a QuickCheck state machine*** is a finite sequence of one-step transitions where the commands are generated by the generator function:

$$s_0 \xrightarrow[\alpha_1]{\psi_1} s_1,\ s_1 \xrightarrow[\alpha_2]{\psi_2} s_2,\ \ldots,\ s_{n-1} \xrightarrow[\alpha_n]{\psi_n} s_n$$

such that

$$\begin{array}{l}
\forall i \in \mathbb{N}\ .\ i < n \implies \alpha_{i+1} \in \mathbf{G}(s_i), \text{ and} \\
\forall i \in \mathbb{N}\ .\ i < n \implies \psi_{i+1} \notin \bigcup_{j=0}^{i} \mathrm{VARS}(s_j) \cup \bigcup_{k=1}^{i} \mathrm{VARS}(\alpha_k), \\
\forall i, j \in \mathbb{N}^+\ .\ i \le n \wedge j \le n \wedge \psi_i = \psi_j \implies i = j
\end{array}$$

Notation 4.7. Let

$$s_0 \xrightarrow[\alpha_1]{\psi_1} s_1 \xrightarrow[\alpha_2]{\psi_2} s_2 \cdots s_{n-1} \xrightarrow[\alpha_n]{\psi_n} s_n$$

refer to the test run

$$s_0 \xrightarrow[\alpha_1]{\psi_1} s_1,\ s_1 \xrightarrow[\alpha_2]{\psi_2} s_2,\ \ldots,\ s_{n-1} \xrightarrow[\alpha_n]{\psi_n} s_n$$

Definition 4.12. Given a QuickCheck state machine, let **TR** be the ***set of all possible test runs***, that is:

$$\{s_0 \xrightarrow[\alpha_1]{\psi_1} s_1,\ \ldots,\ s_{n-1} \xrightarrow[\alpha_n]{\psi_n} s_n \mid \exists s_0, \ldots, s_n\ .\ \forall i \in \{1, \ldots, n\}\ .\ \alpha_i \in \mathbf{G}(s_{i-1}) \wedge s_i = \delta(s_{i-1}, \alpha_i, \psi_i)\}$$

Definition 4.13. Given a test run of a QuickCheck state machine

$$s_0 \xrightarrow[\alpha_1]{\psi_1} s_1 \xrightarrow[\alpha_2]{\psi_2} s_2 \cdots s_{n-1} \xrightarrow[\alpha_n]{\psi_n} s_n$$

its ***corresponding test case*** is the sequence of pairs

$$\langle \alpha_1, \psi_1 \rangle \cdot \langle \alpha_2, \psi_2 \rangle \cdots \langle \alpha_n, \psi_n \rangle$$

Thus, to compute a set of test cases (a *test suite*) from a QuickCheck state machine, we can run the state machine a number of times and extract a test case from each test run. Note also that given a test case, the corresponding test run can be reconstructed since the transition relation is a function (and every test run starts in the initial state s_0).

Example 4.6 *(Test run)*. An example test run for the `Auth` API that can be generated by the state machine is the following sequence of commands and variables, which registers a new user, generates a token, validates that token, and finally revokes it:

$$s_0 \xrightarrow[\texttt{reg("u","p")}]{\psi_1} s_1 \xrightarrow[\texttt{gen("u","p")}]{\psi_2} s_2 \xrightarrow[\texttt{val(elem(}\psi_1\texttt{,1))}]{\psi_3} s_3 \xrightarrow[\texttt{rev(elem(}\psi_1\texttt{,1))}]{\psi_4} s_4$$

where $s_0 = \{\texttt{\%\{\},[]}\}$, $s_1 = \{\texttt{\%\{"u"=> "p"\},[]}\}$, $s_2 = \{\texttt{\%\{"u"=> "p"\},[elem(}\psi_2\texttt{,1)]}\}$, $s_3 = s_2$, and $s_4 = s_1$. A test case can be extracted from the test run by simply removing the states:

$$\langle \texttt{reg("u","p")},\psi_1\rangle, \langle \texttt{gen("u","p")},\psi_2\rangle, \langle \texttt{val(elem(}\psi_2\texttt{))},\psi_3\rangle, \langle \texttt{rev(elem(}\psi_2\texttt{))},\psi_4\rangle$$

4.4. Formal definition of a correct execution

Definition 4.14. Given a QuickCheck state machine, a **system-under-test** p is a total function from sequences of ground commands to sets of sequences of ground result Σ-terms:

$$p : (\mathbf{\Lambda} \cap T(\Sigma_{cd}))^* \rightarrow \mathbf{2}^{(T(\Sigma_d))^*}$$

such that

$$\forall c \in \text{domain}(p), \forall r \in p(c) \;\; . \;\; |c| = |r|$$
$$\forall c, c', r \;\; . \;\; c \in \text{domain}(p) \wedge c \cdot c' \in \text{domain}(p) \wedge r \in p(c) \implies \exists r' \;\; . \;\; r \cdot r' \in p(c \cdot c')$$

Note that the definition permits non-deterministic systems-under-test. Moreover, the conditions require (i) that the function p returns a sequence of equal length to its argument sequence, and (ii) if a command sequence $c \cdot c'$ extends a prefix c that has a result sequence r, then the extended sequence $c \cdot c'$ has a result sequence r' which coincides with r for the length of the prefix.

Example 4.7 *(System-under-test)*. The definition above models a system-under-test as a possibly nondeterministic system. A system-under-test for our running example might relate the sequence of commands

$$\texttt{reg("u","p")} \cdot \texttt{gen("u","p")} \cdot \texttt{val("t")} \cdot \texttt{rev("t")}$$

to the following two sequences of *non-deterministic results*:

$$\left\{ \begin{array}{ccccccc} \texttt{:ok} & \cdot & \texttt{\{:ok,"t"\}} & \cdot & \texttt{:ok} & \cdot & \texttt{:ok} \;, \\ \texttt{:ok} & \cdot & \texttt{:error} & \cdot & \texttt{:error} & \cdot & \texttt{:error} \end{array} \right\}$$

Then, `:ok·:error·:error·:error` and `:ok·{:ok,"t"}·:ok·:ok` are two possible "executions" of the system-under-test.

Definition 4.15. Given a QuickCheck state machine, the corresponding test case of a test run

$$tc = \langle \alpha_1, \psi_1 \rangle \cdot \langle \alpha_2, \psi_2 \rangle \cdots \langle \alpha_n, \psi_n \rangle$$

and a system-under-test p, an **execution of** tc **under** p is a substitution $\phi = \phi_n$ such that

$$\phi_0 \overset{\text{def}}{=} [\,]$$
$$\phi_i \overset{\text{def}}{=} \phi_{i-1} \circ [\psi_i \leftarrow r_i] \text{ if } i > 1$$

where the sequence $r_1 \cdot r_2 \cdots r_n$ are the returned values of p

$$r_1 \cdot r_2 \cdots r_n \in p(\llbracket \phi_1(\alpha_1) \rrbracket \cdot \llbracket \phi_2(\alpha_2) \rrbracket \cdots \llbracket \phi_n(\alpha_n) \rrbracket)$$

Clearly $\phi_1(\alpha_1)$ is a ground term, and so is $\phi_i(\alpha_i)$ for $i > 1$ since α_i can only contain variables $\psi_1, \ldots, \psi_{i-1}$ that are bound to ground variables in the substitution ϕ_i. Note that the sequence $r_1 \cdot r_2 \cdots r_n$, if it exists, can be constructed step-by-step, with the help of the restrictions on p in Definition 4.14.

Definition 4.16. Given a QuickCheck state machine, a test run

$$tr = s_0 \rightarrow [\alpha_1]\psi_1 s_1 \rightarrow [\alpha_2]\psi_2 s_2 \;\cdots\; s_{n-1} \rightarrow [\alpha_n]\psi_n s_n$$

its corresponding test case

$$tc = \langle \alpha_1, \psi_1 \rangle \cdot \langle \alpha_2, \psi_2 \rangle \cdots \langle \alpha_n, \psi_n \rangle$$

a system-under-test p, and an execution ϕ of tc under p, we say that **the execution** ϕ **is correct** if and only if

$$\forall i \in \mathbb{N}^+ \ . \ i < n \implies \mathbf{P}([\![\phi(s_{i-1})]\!], [\![\phi(\alpha_i)]\!], [\![\phi(\psi_i)]\!])$$

let us refer to this predicate as $\mathbf{CE}(tr, \phi)$.

Example 4.8 *(Test execution).* Let's explore the test case in Example 4.6:

$$s_0 \xrightarrow[\texttt{reg("u","p")}]{\psi_1} s_1 \xrightarrow[\texttt{gen("u","p")}]{\psi_2} s_2 \xrightarrow[\texttt{val(elem(}\psi_2\texttt{,1))}]{\psi_3} s_3 \xrightarrow[\texttt{rev(elem(}\psi_2\texttt{,1))}]{\psi_4} s_4$$

As mentioned in Example 4.7, an execution of the test case might be

$$\phi = [\psi_1 \leftarrow \texttt{: ok}, \psi_2 \leftarrow \{\texttt{: ok}, \varepsilon t \varepsilon\}, \psi_3 \leftarrow \texttt{: ok}, \psi_4 \leftarrow \texttt{: ok}]$$

Note that in the third and fourth commands in the test case, `elem(`ψ_2`,1)` is replaced with the interpretation $[\![$`elem({:ok,"t"},1)`$]\!]$ resulting in the actual commands `val("t")` and `rev("t")`.

Example 4.9 *(Execution States).* Continuing with the example, considering the states $s_0, \dots, s_4$ in the test case, and the execution, the actual states are the result of applying ϕ:

$\phi(s_0) = \{$`%{},[]`$\}$
$\phi(s_1) = \{$`%{"u"=> "p"},[]`$\}$
$\phi(s_2) = \{$`%{"u"=> "p"},["t"]`$\}$
$\phi(s_3) = \{$`%{"u"=> "p"},["t"]`$\}$
$\phi(s_4) = \{$`%{"u"=> "p"},[]`$\}$

Example 4.10 *(Correct execution).* The execution in Example 4.8 is a correct execution of the test case in Example 4.6 as all matched states (see Example 4.9), commands and results (see Example 4.8) satisfy the postcondition predicate defined in Example 4.5.

4.5. A QuickCheck property for QuickCheck state machines

Definition 4.17. Given a QuickCheck state machine m, the ***QuickCheck property on*** m is:

1. the signature Σ_m,
2. the induced term algebra $T(\Sigma, \mathbf{V})_m$,
3. the generator $\mathbf{G}$ is the set of all possible valid test runs that can be generated by m, which corresponds to Definition 4.12, that is $\mathbf{G} = \mathbf{TR}_m$
4. the postcondition $\mathbf{P}$ corresponds to the notion of correct execution in Definition 4.16, that is
   $\mathbf{P}(tr, \phi) \iff tr \in \mathbf{G} \wedge \mathbf{CE}_m(tr, \phi)$
5. a shrinking function for sequential test cases.

Definition 4.17 defines the QuickCheck property on QuickCheck state machines. The property generator is a set that contains every valid test run derivable from the state machine m. The postcondition of the property holds when the given test run is a valid test run (which means that it can be generated by the property generator) and the postcondition of the state machine holds for every step in the test run.

Definition 4.18. Given the QuickCheck property on a QuickCheck state machine m, and a system-under-test f, we say that the property holds of the program f if and only if for every test run $tr \in \mathbf{G}$ and for every execution ϕ of tr under f, the postcondition $\mathbf{P}(tr, \phi)$ holds.

Definition 4.19. Given two QuickCheck state machines m and m', and their associated QuickCheck properties, we say that m' ***refines*** m (written $m \sqsubseteq m'$), when, for all system-under-tests f:

$$\mathbf{G}_m \subseteq \mathbf{G}_{m'}$$
$$\forall t \in \mathbf{G}_m, r \in f(t) \ . \ \mathbf{P}_{m'}(t, r) \implies \mathbf{P}_m(t, r)$$

Intuitively, the notion of refinement on QuickCheck state machines means that every test-run that can be generated by the base state machine m can also be generated by the refined state machine m', and that for every test run that can be generated by the base machine m that is accepted by the refined machine m' must be also accepted by m.

Example 4.11 *(Proving refinement).* Let us modify the running example in 4.5 by adding a new conjunction (restriction) to the `gen` transition:

$\mathbf{P}(\{users, tokens\}, \texttt{gen}(u, p), \ r) \iff$
$\quad \{u, p\} \in users \wedge \texttt{elem}(r, 0) = \texttt{:ok} \wedge unique(\texttt{[elem}(r, 1) \mid tokens\texttt{]}) \vee r = \texttt{:error}$

where the Elixir predicate *unique* expresses that none of its list elements are duplicated, up to normal Elixir term equality (`==`). Let us refer to the new model as m', and the original model as m. We proceed to prove that m' refines m (according to Definition 4.19).

Trivially the first property holds, i.e., $\forall t \in \mathbf{G}_m \ . \ t \in \mathbf{G}_{m'}$, since $\mathbf{G}_m$ is identical to $\mathbf{G}_{m'}$ (only the postcondition predicate is changed). It remains to prove that $\forall t \in \mathbf{G}_m, r \in f(t) \ . \ \mathbf{P}_{m'}(t,r) \implies \mathbf{P}_m(t,r)$. Again this is trivial since the postcondition predicate $\mathbf{P}_{m'}(t,r)$ is identical to $\mathbf{P}_m(t,r)$, except for the `gen` transition where m' rejects some results accepted by m.

Example 4.12 *(Proving non-inspection)*. Let us exemplify the definition of non-inspection in Definition 4.9 by attempting to prove that the running example model Example 4.5 is non-inspective. That is, that for every state $s \in \mathbf{S}$, command $\alpha \in \mathbf{\Lambda}$, symbolic variable $\psi \in \mathbf{V}$, and substitution $\phi : \mathbf{V} \to T(\Sigma_d)$ the following properties hold:

$$\phi(\delta(s,\alpha,\psi)) = \delta(\phi(s), \phi(\alpha), \psi), \text{ and}$$
$$\alpha \in \mathbf{G}(s) \iff \phi(\alpha) \in \mathbf{G}(\phi(s))$$

First, let us establish an invariant over the reachable model states: every state s reachable through the transition relation δ is of the shape $\{users, tokens\}$ where $\text{VARS}(users) = \emptyset$.

Proof sketch: the initial state $\{[],[]\}$ has this shape and has no variables, and all transitions (`reg`,`gen`,`val`,`rev`) preserve the shape, and moreover symbolic variables migrate only to the *tokens* state element.

Moreover, since *user* and *pass* do not contain variables, the equivalences $user = \phi(user)$, $users = \phi(users)$, and $pass = \phi(pass)$ will be frequently utilized.

Now, let us proceed to prove the first property for all transitions. Consider first as an example $\texttt{reg}(user, pass)$. We assume that the transition is enabled, i.e., *user* is not in $\texttt{Map.keys}(users)$.

$$\begin{aligned}
&\phi(\delta(\{users, tokens\}, \texttt{reg}(user, pass), \psi)) \\
&= \phi(\{\texttt{Map.put}(users, user, pass), tokens\}) && \text{unfolding } \texttt{reg} \\
&= \{\phi(\texttt{Map.put}(users, user, pass)), \phi(tokens)\} \\
&= \{\texttt{Map.put}(\phi(users), \phi(user), \phi(pass)), \phi(tokens)\} \\
&= \delta(\{\phi(users), \phi(tokens)\}, \texttt{reg}(\phi(user), \phi(pass)), \psi) && \text{folding } \texttt{reg} \\
&= \delta(\phi(\{users, tokens\}), \phi(\texttt{reg}(user, pass)), \psi)
\end{aligned}$$

Note that the derivation omits the step to prove that the derived transition is enabled, i.e., that $\phi(user)$ is not in $\texttt{Map.keys}(\phi(users))$. Since $\phi(user) = user$ and $\phi(users) = users$, this follows immediately from the fact that the original transition was enabled.

Consider next $\texttt{rev}(token)$.

$$\begin{aligned}
&\phi(\delta(\{users, tokens\}, \texttt{rev}(token), \psi)) \\
&= \phi(\{users, \texttt{List.delete}(tokens, token)\}) && \text{unfolding } \texttt{rev} \\
&= \{\phi(users), \phi(\texttt{List.delete}(tokens, token))\} \\
&=_? \{\phi(users), \texttt{List.delete}(\phi(tokens), \phi(token))\} \\
&= \delta(\{\phi(users), \phi(tokens)\}, \texttt{rev}(\phi(token)), \psi) && \text{folding } \texttt{rev} \\
&= \delta(\phi(\{users, tokens\}), \phi(\texttt{rev}(token)), \psi)
\end{aligned}$$

The crucial proof step is $\phi(\texttt{List.delete}(tokens, token)) =_? \texttt{List.delete}(\phi(tokens), \phi(token))$. Unfortunately, this is simply not true for all ϕ, *tokens* and *token*. Consider, for example, $tokens = [elem(\psi_0, 1), elem(\psi_1, 1), elem(\psi_2, 1)]$, $token = elem(\psi_2, 1)$ and $\phi = \{\langle\psi_0, \{\texttt{:ok}, 1\}\rangle, \langle\psi_1, \{\texttt{:ok}, 2\}\rangle, \langle\psi_2, \{\texttt{:ok}, 1\}\rangle\}$.

Then $\phi(\texttt{List.delete}([elem(\psi_0, 1), elem(\psi_1, 1), elem(\psi_2, 1)], elem(\psi_2, 1))) = \phi([elem(\psi_0, 1), elem(\psi_1, 1)]) = [1,2]$. However, $\texttt{List.delete}(\phi([elem(\psi_0, 1), elem(\psi_1, 1), elem(\psi_2, 1)]), \phi(\psi_2)) = \texttt{List.delete}([1,2,1], 1) = [2,1]$!

What has gone wrong? The code for the `rev` transition *inspects* the symbolic token variable through a comparison in the definition of `List.delete`, which is not permitted. There are a number of ways in which the problem can be addressed. First, we can modify the example state to be a tuple with three components: $\{users, tokens, revoked\}$. Next, the `rev` transition is modified to instead of deleting a token from the tokens component of the state, rewrite the state as $\{users, tokens, [token|revoked]\}$. Finally, the postcondition of the `val` transition is strengthened:

$\mathbf{P}(\{users, tokens\}, \texttt{val}(t), \ r) \iff$
$\quad t \in tokens \wedge \texttt{t} \notin revoked \wedge r = \texttt{:ok} \vee r = \texttt{:error}$

NOTATION	ELIXIR CODE	MEANING
defs(M)	`M.__info__(:functions)`	Set of functions defined in M as a list of pairs $\{f,n\}$ (function name and arity)
f/n	$\{f,n\}$	A tuple with function name f and arity n
$f/n \in l$	`Enum.member?(l, {f,n})`	Tuple $\{f,n\}$ is in the list l
B **is a behaviour**		`behaviour_info/1` $\in$ *defs*(B)
callbacks(B)	`B.behaviour_info(:callbacks)`	Set of functions (callbacks) required to be implemented in a behaviour B as a list of tuples $\{f,n\}$ (function name and arity)
behaves(M,B) (M ***behaves as*** B)		*callbacks*$(B) \subseteq$ *defs*(M)

Fig. 4. Notation used for dealing with modules and behaviours.

Moreover, the `gen` transition, which generates a new token, has to guarantee in its postcondition that the generated token has not been revoked; we omit the specification from here.

Another alternative, that will be used here, is to assume that all generated tokens are unique, which is probably what a good implementation should require anyway. Under this condition, the troublesome equality $\phi(\texttt{List.delete}(tokens, token)) =_? \texttt{List.delete}(\phi(tokens), \phi(token))$ does hold. However, now we have the obligation to check that implementations generate unique tokens; to accomplish this, we assume the *refined* model developed in Example 4.11 (and which was shown to refine the original model m). This model ensures, via an invariant proof, that the *tokens* field of the model state contains no duplicate elements. With this change, the proof for non-inspection goes through (further proof details omitted).

5. A semantics for functional-style state machines

In this section, we define the formal meaning of a functional-style state machine (Sect. 3.2) implemented in an Elixir module M by deriving from it a QuickCheck state machine (defined in Definition 4.7 in Sect. 4). That is, we provide a transition function $\boldsymbol{\delta}$, a command generator $\mathbf{G}$, a postcondition predicate $\mathbf{P}$ and an initial state $\mathbf{s_0}$ by translation from the callback functions of M.

In the following, we assume that if the execution of a test on a system-under-test raises an exception, then the result of the test is represented as the term $\{\texttt{:exception}, class, reason\}$ where *class* is one of `:error`, `:exit` or `:throw`, and *reason* is an arbitrary value.

Notation 5.1. To be able to reason about models written as Elixir modules, like the one in Fig. 3, we need some way to inspect the contents of a module. Elixir offers some constructs to inspect module contents. These constructs are a bit verbose, so we have decided to introduce a notation to reflect them, shown in Fig. 4.

Most PBT libraries that support state-machines defined in the functional-style, like EQC or PropEr, use a *behaviour* called `StateM`, which describes the API of these state-machines. Informally, a behaviour in Erlang and Elixir is similar to an interface in, e.g., Java. That is, a behaviour enumerates a set of functions (callbacks) that must be implemented. If a module claiming to implement a behaviour does not implement all these functions, then the Erlang/Elixir compiler reports an error. The callbacks defined in the `StateM` behaviour are:

$$callbacks(\texttt{StateM}) = \{\texttt{initial\_state/0}, \texttt{command/1}, \texttt{precondition/2}, \texttt{next\_state/3}, \texttt{postcondition/3}\}$$

Definition 5.1. Given a module M that behaves as `StateM`, we translate the Elixir module M to a QuickCheck state machine in Definition 4.7 as follows:

1. Σ_d is the set of all Elixir constructors, Σ_f is the set of all function symbols accessible from the module of the running example, and Σ_c is the set of all the Elixir atoms that can be used as function names.
2. $\mathbf{V}$ is a countable set of variables represented as Elixir terms $\{\texttt{:var}, i\}$ (i is a natural number).
3. The induced term algebrae $T(\Sigma, \mathbf{V})$,
4. The set of available states $\mathbf{S}$ is represented by the set of all valid Elixir terms.
5. The set of possible commands $\boldsymbol{\Lambda}$ is represented by the set of all possible tuples $\{\texttt{:call},\ M,\ f,\ args\}$ where M is a valid module name, f is a valid function name and *args* is the list arguments where each element is a valid Elixir term.
6. $\boldsymbol{\delta}(s, \alpha, \psi) \overset{\text{def}}{=} [\![M.\texttt{next\_state}(s, \alpha, \psi)]\!]$
7. $\mathbf{G}(s) \overset{\text{def}}{=} \{c \in [\![M\texttt{.command(}s\texttt{)}]\!] \mid [\![M\texttt{.precondition(}s, c\texttt{)}]\!]\}$
8. $\mathbf{P}(s, \alpha, r) \overset{\text{def}}{=} (\neg\exists class, reason\ .\ r = \{\texttt{:exception}, class, reason\}) \wedge [\![M\texttt{.postcondition(}s,\ \alpha,\ r\texttt{)}]\!]$
9. $\mathbf{s_0} \overset{\text{def}}{=} [\![M\texttt{.initial\_state()}]\!]$

10. For all $t \in T(\Sigma_d)$, the interpretation function is defined as

$$[\![t]\!] \overset{\text{def}}{=} \begin{cases} [\![M'.f([\![t_1]\!],\dots,[\![t_n]\!])]\!] & \textbf{if } t = \{\texttt{:call}, M', f, [t_1, \dots, t_n]\} \\ [\![d([\![t_1]\!],\dots,[\![t_n]\!])]\!] & \textbf{otherwise } (t = d(t_1,\dots,t_n)) \end{cases}$$

Note that, traditionally in EQC, thrown exceptions are treated as errors, as can be seen in the postcondition predicate. Thus, to reason about exceptions in a model these must be captured in the system-under-test and be represented as "normal" command results.

Definition 5.2. Given a module M that behaves as `StateM`, we say it is ***non-terminating*** if the QuickCheck state machine obtained after translating M using Definition 5.1 is ***non-terminating***. That is,

$$\begin{array}{l} \forall s \in \mathbf{S} \ . \ \mathbf{G}(s) \neq \emptyset \iff \\ \forall s \in \mathbf{S} \ . \ \{c \in [\![M\texttt{.command}(s)]\!] \mid [\![M\texttt{.precondition}(s,c)]\!]\} \neq \emptyset \iff \\ \forall s \in \mathbf{S} \ . \ \exists c \in [\![M\texttt{.command}(s)]\!] \ . \ [\![M\texttt{.precondition}(s,c)]\!] \end{array}$$

Thus, in order to write a functional-style state machine that does not terminate, the definition of M`.command(`s`)` has to be able to generate at least one command α for every state s such that M`.precondition(`s, α`)` holds.

Definition 5.3. Given a module M that behaves as `StateM`, we say it is ***non-inspective*** if the QuickCheck state machine obtained after translating M using Definition 5.1 is ***non-inspective***. That is,

1. $\forall s,\alpha,\psi,\phi \ . \ s \in \mathbf{S}, \alpha \in \mathbf{\Lambda}, \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d) \ . \ \phi(\delta(s,\alpha,\psi)) = \delta(\phi(s),\phi(\alpha),\psi) \iff$
   $\forall s,\alpha,\psi,\phi \ . \ s \in \mathbf{S}, \alpha \in \mathbf{\Lambda}, \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d) \ . \ \phi([\![M\texttt{.next\_state}(s,\alpha,\psi)]\!]) = [\![M\texttt{.next\_state}(\phi(s),\phi(\alpha),\psi)]\!]$

2. $\forall s,\alpha,\psi,\phi \ . \ s \in \mathbf{S}, \alpha \in \mathbf{\Lambda}, \phi : \mathbf{V} \to T(\Sigma_d) \ . \ \alpha \in \mathbf{G}(s) \iff \phi(\alpha) \in \mathbf{G}(\phi(s)) \iff$
   $\forall s,\alpha,\psi,\phi \ . \ s \in \mathbf{S}, \alpha \in \mathbf{\Lambda}, \phi : \mathbf{V} \to T(\Sigma_d) \ . \ \alpha \in \{c \in [\![M\texttt{.command}(s)]\!] \mid [\![M\texttt{.precondition}(s,c)]\!]\}$
   $\iff \phi(\alpha) \in \{c \in [\![M\texttt{.command}(\phi(s))]\!] \mid [\![M\texttt{.precondition}(\phi(s),\phi(c))]\!]\}$

Thus, there are two properties that should hold in the functions that define a non-inspective functional-style state machine. The first ensures that the state reached after applying substitutions ϕ to the result of `next_state` has to be the same state as if these substitutions were applied to its arguments, as `next_state` defines the transition function of a functional-style state machine. The second shows that if a command α can be generated by M`.command` and accepted by M`.precondition`, then the command after applying the substitutions ϕ should also be able to be generated by M`.command(`s`)` and accepted by M`.precondition` after applying the substitution to their arguments.

5.1. The meaning of QuickCheck state machine functions

QuickCheck provides two principal functions for interacting with state machine callback modules: `commands(Module)` which is a generator for test cases, and `run_commands(TestCase)` which, given a test case, executes that test case and checks that result values satisfy the postcondition.

The generation of a concrete test case using the `commands` function corresponds to the calculation of a run (see Definition 4.11) and its corresponding test case (see Definition 4.13) in the semantics developed in Sect. 4. Similarly, applying the `run_commands` function to a concrete test case corresponds to executing the test case according to Definition 4.15, and checking the postconditions according to Definition 4.16 in the semantics.

Let us recall the Elixir property used in Sect. 3.1 QuickCheck state machines to generate and run test cases of the model `M`:

```
forall run in commands(M) do run_commands(M, run) == :ok end
```

Given the functional state machine `M`, let m be the result of its translation to a QuickCheck state machine in Definition 5.1. Then

- `commands(M)` is a generator of tests runs, equivalent to what we named $\mathbf{G} = \mathbf{TR}_m$,
- the variable `run` is bound to a test case derived from that generator, equivalent to a run tr of m,
- the postcondition predicate $\mathbf{P}(tr,\phi)$ is checked by the execution of function `run_commands(M, run)` which returns the value `:ok` if all commands executed correctly (and otherwise returns a term detailing the execution history). Notice that in QuickCheck substitutions (of result variables in the test case) are handled internally by the function `run_commands`, whereas they are made explicit in the semantics developed in Sect. 3.1.

In a sense, the main difference between the formalization of test generation and execution in the article, and the QuickCheck API, is that in the formalization the substitutions are made explicit, to further decouple the state machine model from the implementation under test.

```
1   defmodule Auth.Makina do
2     use Makina, implemented_by: Auth
3
4     @type user()  :: Auth.user()
5     @type pass()  :: Auth.pass()
6     @type token() :: Auth.token()
7
8     state users:  %{} ::  %{ user() => pass() },
9           tokens:  [] ::  [ symbolic(token()) ]
10
11    invariants unique_tokens: Enum.uniq(tokens) == tokens
12
13    command reg(user :: user(), pass :: pass()) :: :ok | :error do
14      args user: string(), pass: string()
15      valid user not in Map.keys(users)
16      next if valid, do: [users: Map.put(users, user, pass)]
17      post if valid, do: result == :ok, else: result == :error
18    end
19
20    command gen(user :: user(), pass :: pass()) :: {:ok, token()} | :error do
21      pre users != %{}
22      args let user <- oneof(Map.keys(users)),
23           do: [user: user, pass: Map.get(users, user)]
24      valid {user, pass} in users
25      next if valid, do: [tokens: [symbolic(elem(result, 1))|tokens]]
26      post if valid, do: match?({:ok, _}, result), else: result
27    end
28
29    command rev(token :: symbolic(token()) | integer()) :: :ok | :error do
30      args token: oneof([pos_integer()|tokens])
31      valid token in tokens
32      next if valid, do: [tokens: List.delete(tokens, token)]
33      post if valid, do: result == :ok, else: result == :error
34    end
35
36    command val(token :: symbolic(token()) | integer()) :: :ok | :error do
37      args token: oneof([pos_integer()|tokens])
38      valid token in tokens
39      post if valid, do: result == :ok, else: result == :error
40    end
41  end
```

Fig. 5. The authentication model implemented in the Makina style.

6. Makina state machine models

In this section, we describe informally the Makina domain specific language for expressing state machines by means of the example in Fig. 5, which expresses the authentication model using the Makina domain specific modelling language.

Models written in the Makina style have similar callbacks to the callbacks used in EQC command-style state machines, i.e., a command `cmd` may have the callbacks `cmd_pre/1`, `cmd_args/1`, `cmd_post/3` and `cmd_next/3`. Their equivalents in Makina style are **`pre`**, **`args`**, **`post`** and **`next`** respectively. Among the minor differences is that the EQC command-style callback `cmd_pre/2` is here named **`valid_args`** (to prevent confusion with **`pre`**), and the functions `cmd(arg1,...,argN)` – i.e., functions with the same name as a command – which execute the command on the system-under-test, are in Makina named **`call`**.

Note in line 2 in Fig. 5 the declaration

```
  use Makina, implemented_by: Auth
```

The **`use`** `Makina` part will cause the Makina model to be translated to a functional-style model when the module containing the model is compiled, and the `implemented_by: Auth` part informs the Makina library that the implementation (system-under-test) to be tested will be provided by the Elixir `Auth` module. Furthermore, note on line 25 the use of the “symbolic” construct, e.g., `symbolic(elem(result,1))`. This notifies Makina that the corresponding expression is a symbolic one containing uninstantiated variables – the result of executing the command – which should not be executed when the test case is being constructed.

In the following, we enumerate the most significant new features of Makina state machines.

Explicit state declaration In Makina state machines, the state is declared explicitly. A state consists of a set of attributes which can optionally be typed. Internally, a state is represented as a map with the attribute names as keys. In the example, the state declaration is on lines 8 and 9.

Explicit command declarations Next, the commands are enumerated. Borrowing from the PropCheck library, each command is defined using its own **`command`** construct.[9] A number of callback functions are defined in the body of a command, e.g., **`next`** and **`post`**. A difference compared to PropCheck is that commands have structure: a command has a fixed number of named parameters, which can be optionally typed. For example, the `reg` command defined between lines 13 and 18 has two parameters: `user` and `pass`. Note that the order of parameters in command declarations does not matter; they are distinguished only through their names.

Redirection of functions for executing the test In contrast with Fig. 3, no functions are defined to execute the test (e.g., executing the `reg`, `gen`, `rev` and `val` commands). Rather, we simply declare that the `Auth` provides such functions using the `implemented_by: Auth` keyword declaration. An alternative is to provide **`call`** callback functions in all commands; these functions will then be called when a command is to be executed in the system-under-test.

*The **`args`** callbacks create a keyword list of command parameter generators* Note the definition of the **`args`** function, such as **`args`** in `reg` on line 14. In standard EQC command-style models, the **`args`** function returns a generator that produces a list of command parameters. However, in Makina, a keyword list is used with command parameters as keys (in `reg`, this would be `user` and `pass`).

*The **`next`** callbacks return a keyword list of updates* In Makina models, the **`next`** functions should return a keyword list containing updates to the model state. Each element of the list contains a key with the name of the attribute to update and the corresponding value. Thus, **`next`** callback functions do not need to mention parts of the state (attributes) that are not changed. For example, the command `reg` in Fig. 5 on line 16 updates the state, adding the new user when it is not already registered.

Implicit parameters in callback functions Furthermore, note the absence of parameters in the callback functions, for example, there is no **`state`** parameter in the **`pre`** functions, nor are the command parameters present in the **`next`** function for `reg` (on line 16), even though these parameters *are* used in the body of the functions. During macro expansion, callback function headers are rewritten to make such implicit parameters explicit. The following implicit parameters or attributes are available:

- the state parameter (**`state`**)
- the state attributes
- the command arguments
- the result parameter (`result`)
- the result of the valid computation (**`valid`**)

Not having to explicitly mention parameters has two advantages: less code needed, and, more importantly, avoiding model bugs due to incorrect declarations of such parameters.[10]

A new callback: valid The **`valid`** callback is new in Makina. The intention with the callback is to identify the commands that the system-under-test should reject as being “invalid”, and thus normally do not change the state of the system-under-test. That is, a test containing only valid commands can be considered to correspond to a positive test in standard testing terminology, whereas a test with some invalid commands corresponds to a negative test which should cause the system-under-test to issue an error indication. There is a subtle difference with the **`pre`** and **`valid_args`** callbacks, which identify commands that should never be executed by the system-under-test (e.g. perhaps because issuing them would break some hardware part of the system-under-test). Consider, for instance, the definition of **`valid`** in the `reg` command on line 15, which identifies a `reg` command as being invalid if the user being registered has been previously registered. This information is used in the **`next`** callback function on line 16, which updates the state only if the `reg` command has been deemed valid.

Type checking Typing information provided in the state attribute declarations and command parameters is used to provide better diagnostics of type errors in Makina models, and the type information is preserved by the translation into a functional-style model. Such type information is used both by static type checkers like Dialyzer, by Makina itself during run-time, and thirdly can be used by dedicated run-time type checkers like Elixir TypeCheck. Consider, for instance, the definition of the **`next`** callback in the reg command on line 16. The code resulting from the translation of this callback function into the functional-style (i.e., combining various **`next`** callbacks into a single `next_state` callback) will have a type specification where the state is typed (due to types attached to attributes in the Makina model), and the command will be typed (due to the attribute types specified in the Makina model). Moreover, the type of the return value from the function is known as well. Thus, Dialyzer can check during compile time that, for example, the map operations are called with correctly typed parameters. On the other hand, the **`args`** callback in the `gen` command on lines 22 and 23 is harder for Dialyzer to check, as the generator constructs (`let`, `oneof`, etc.) are not typed. However, once, at run time, a keyword list has been generated, the Makina code can check that all command parameters are present as keys in the keyword list, and, moreover, the Elixir TypeCheck run time type-checker can check that values in the generated keyword list are

[9] In PropCheck the `defcommand` construct was used instead.

[10] Although perhaps surprising, such coding errors are not uncommon, and e.g. the Dialyzer type checker is not always capable of detecting these type of errors at compile time.

instances of their correct types. Note that the latter two types of run-time type errors can be clearly marked as *model errors*, indicating which combination of models, commands, and command callback are responsible for the type error.

Invariants Invariants express properties that should invariably hold in the model state. Consider, as an example, the invariant declaration on line 11, which expresses that no two users are associated with the same token; all issued tokens are unique. Invariants are checked after the execution of each command.

7. The semantics of the makina domain-specific language

This section formally describes the syntax and semantics of the Makina domain-specific language. The semantics of a Makina state machine model are obtained through its translation into a functional-style state machine. Note that translation of the Makina preserves the type information present in a Makina model. Given the present immature state of Elixir static type checking tools, type specifications are translated into runtime type checking assertions.

7.1. Translation to functional-style state machines

A Makina model is a program written in the Makina language. These models are written in modules in the Elixir host language, and they are composed of a state, one or more commands, and optionally invariants. The following EBNF grammar specifies the valid syntax at the module level of a Makina model:

$$
\begin{aligned}
\langle\text{model}\rangle ::= {} & \texttt{\textbf{defmodule}}\ \langle\text{module}\rangle\ \texttt{\textbf{do}} \\
& \quad \texttt{\textbf{use}}\ \texttt{Makina}\ \langle\text{options}\rangle \\
& \quad \langle\text{state}\rangle\ \langle\text{invariants}\rangle\ \langle\text{commands}\rangle \\
& \texttt{\textbf{end}} \\
\langle\text{module}\rangle ::= {} & \textit{Elixir module name}
\end{aligned}
$$

In these models, state definitions are global to the rest of the model, that is, the attributes defined in the state can be accessed by both commands and invariants.

In the Makina language, the keyword **`state`** followed by a list of attributes is used to define the model state. Each attribute in this list contains its name and an initial value, which can be any valid expression. Optionally, state attributes can be typed. If no type definition is provided, the type defaults to any valid term $T(\Sigma)$. In Makina, the keyword **`invariants`** followed by a list of invariants introduces invariant predicates about the state of the model (that should hold for all reachable states). An invariant is named, and its defining predicate expression may refer to the model state. Invariants are verified during the execution of the test, meaning that the model states are guaranteed to contain neither symbolic variables nor symbolic expressions. The following grammar rules specify the syntax of state attributes and invariants:

$$
\begin{aligned}
\langle\text{state}\rangle &::= \texttt{\textbf{state}}\,\langle\text{attributes}\rangle \mid \epsilon \\
\langle\text{attributes}\rangle &::= \langle\text{attribute}\rangle\ (\texttt{,}\langle\text{attribute}\rangle)^* \mid \epsilon \\
\langle\text{attribute}\rangle &::= \langle\text{atom}\rangle\ \langle\text{expression}\rangle \mid \langle\text{atom}\rangle\langle\text{expression}\rangle\texttt{::}\langle\text{type}\rangle \\
\langle\text{invariants}\rangle &::= \texttt{\textbf{invariants}}\ \langle\text{invariant}\rangle\ (\texttt{,}\langle\text{invariant}\rangle)^* \mid \epsilon \\
\langle\text{invariant}\rangle &::= \langle\text{atom}\rangle\ \langle\text{expression}\rangle \\
\langle\text{atom}\rangle &::= \textit{Elixir atom} \\
\langle\text{expression}\rangle &::= \textit{Elixir expression} \\
\langle\text{variable}\rangle &::= \textit{Elixir variable} \\
\langle\text{type}\rangle &::= \textit{Elixir type}
\end{aligned}
$$

Commands are written using the reserved word **`command`** followed by the command definition: a command name, its arguments, and a number of callback functions. The library provides a default implementation for every callback, except for the callbacks **`call`** and **`args`**.[11] Formally, the syntax of commands is defined through the grammar:

$$
\begin{aligned}
\langle\text{commands}\rangle &::= \langle\text{command}\rangle\ (\texttt{\\n}\langle\text{commands}\rangle)^* \mid \epsilon \\
\langle\text{command}\rangle &::= \texttt{\textbf{command}}\langle\text{atom}\rangle\texttt{(}\langle\text{arguments}\rangle\texttt{)}\ (\texttt{::}\langle\text{type}\rangle)^*\ \texttt{\textbf{do}}\ \langle\text{callbacks}\rangle\ \texttt{\textbf{end}}
\end{aligned}
$$

[11] The first one can be redirected by using `implemented_by:`, as shown in Sect. 6.

$$
\begin{aligned}
\langle\text{callbacks}\rangle &::= \langle\text{callback name}\rangle\ \langle\text{block}\rangle\ (\texttt{\textbackslash n}\ \langle\text{callback name}\rangle\ \langle\text{block}\rangle) \mid \epsilon \\
\langle\text{callback name}\rangle &::= \mathbf{pre} \mid \mathbf{args} \mid \mathbf{valid\_args} \mid \mathbf{call} \mid \mathbf{valid} \mid \mathbf{next} \mid \mathbf{post} \\
\langle\text{arguments}\rangle &::= \langle\text{argument}\rangle\ (\texttt{,}\langle\text{argument}\rangle)^* \mid \epsilon \\
\langle\text{argument}\rangle &::= \langle\text{variable}\rangle \mid \langle\text{variable}\rangle\texttt{::}\langle\text{argument type}\rangle \\
\langle\text{argument type}\rangle &::= \langle\text{variable}\rangle\texttt{()} \mid \langle\text{variable}\rangle\texttt{(}\langle\text{type}\rangle\ (\texttt{,}\langle\text{type}\rangle)^*\texttt{)} \mid \texttt{symbolic(}\langle\text{type}\rangle\texttt{)} \\
\langle\text{block}\rangle &::= \langle\text{expression}\rangle\ (\texttt{\textbackslash n}\langle\text{block}\rangle)^* \\
\langle\text{variable}\rangle &::= \textit{Elixir variable} \\
\langle\text{type}\rangle &::= \textit{Elixir type} \\
\langle\text{expression}\rangle &::= \textit{Elixir expression}
\end{aligned}
$$

Commands in models have unique names, define a number of command arguments, define a return type, and define a number of callback functions, all of which are optional.

Notation 7.1. Given a Makina model m, let *attributes*(m) refer to its attributes, *init*(m) to its initial state, *commands*(m) to its commands, and *invariants*(m) to its invariants.

Notation 7.2. Given a command c, let *arguments*(c) refer to the set of arguments in c and $\mathbf{pre}_c$, $\mathbf{args}_c$, $\mathbf{valid\_args}_c$, $\mathbf{call}_c$, $\mathbf{valid}_c$, $\mathbf{next}_c$ and $\mathbf{post}_c$ be its callbacks.

Definition 7.1. For a Makina model m the *translation* of m to a functional-style state machine is the generation of an Elixir module M that *behaves* as `StateM`, which means that it provides an implementation for each of its callbacks. The definition of each callback follows below:

- $\texttt{initial\_state()} \overset{\text{def}}{=} \textit{init}(m)$
- The generator of commands generates a command call α filtering all the commands that cannot be generated from the given state s, afterwords generates the arguments using **args**:

  $$\texttt{command}(s) \overset{\text{def}}{=} \{\{\texttt{:call}, M, c, a\} \mid c \in \textit{commands}(m) \wedge \mathbf{pre}_c(s) \wedge a \in \mathbf{args}_c(s)\}$$
- $\texttt{precondition}(s,\alpha) \iff \alpha = \{\texttt{:call}, M, c, a\} \ \wedge\ \mathbf{valid\_args}_c(s,a)$
- `next_state/3` passes the result of the **valid** predicate to the corresponding **next** and the result of the latter is used to compute the next state:

  $$\texttt{next\_state}(s, \{\texttt{:call}, M, c, a\}, \psi) \overset{\text{def}}{=} \texttt{update}(s, \mathbf{next}_c(s,a,\psi,\mathbf{valid}_c(s,a,\psi)))$$

  The function `update` applies each of the updates in the list of updates returned by $\mathbf{next}_c$ to the state s.
- In the same way as in `next_state/3`, in `postcondition` the result of the **valid** predicate is introduced. Then, the postcondition and the invariants are checked:
  $\texttt{postcondition}(s, \{\texttt{:call}, M, c, a\}, r) \iff \mathbf{post}_c(s,a,r,\mathbf{valid}_c(s,a,r)) \ \wedge\ \forall p \in \textit{invariants}(m)\ .\ p(s)$
  As seen here, invariants are global postconditions that are checked in all reachable states, in conjunction with the model specific post conditions.

Definition 7.2. Given a Makina model m, we say that it is ***non-terminating*** if the QuickCheck state machine obtained after applying translations in Definition 5.1 and Definition 7.1 is non-terminating.

$$
\begin{aligned}
&\forall s \in \mathbf{S}\ .\ \exists \alpha \in [\![\mathbf{command}(s)]\!]\ .\ [\![\texttt{precondition}(s,\alpha)]\!] \iff \\
&\forall s \in \mathbf{S}\ .\ \exists \alpha \in \{\ \{\texttt{:call}, M, c, a\}\ | c \in \textit{commands}(m) \wedge \mathbf{pre}_c(s) \wedge a \in \mathbf{args}_c(s)\}\ .\ \mathbf{valid\_args}_c(s,a) \iff \\
&\forall s \in \mathbf{S}\ .\ \exists c \in \textit{commands}(m)\ .\ \exists \alpha = \{\texttt{:call}, M, c, a\}\ .\ \mathbf{pre}_c(s) \wedge a \in \mathbf{args}_c(s) \wedge \mathbf{valid\_args}_c(s,a)
\end{aligned}
$$

This definition shows that a Makina model m is non-terminating if for every state s it can always generate a command call α from a command $c \in \textit{commands}(m)$, such that the precondition of that command $\mathbf{pre}_c$ holds for that state s and the generator of arguments for that command $\mathbf{args}_c$ can generate arguments from s that can be accepted by $\mathbf{valid\_args}_c$.

Definition 7.3. Given a Makina model m, we say it is ***non-inspective*** if the QuickCheck state machine obtained after applying translations in Definition 5.1 and Definition 7.1 is non-inspective.

1. $\forall s,\alpha,\psi,\phi\ .\ s \in \mathbf{S}, \alpha \in\ \textit{commands}(m), \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d)\ .\ \phi([\![\texttt{next\_state}(s,\alpha,\psi)]\!]) = [\![\texttt{next\_state}(\phi(s),\phi(\alpha),\psi)]\!] \iff$

$$\forall s,\alpha,\psi,\phi \ \ . \ \ s \in \mathbf{S}, c \in \mathit{commands}(m), \alpha \in \ c, \alpha = \{\texttt{:call}, M, c, a\}, \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d) \ \ .$$
$$\phi(\texttt{update}(s, \mathbf{next}_c(s,a,\psi,\mathbf{valid}(s,a,\psi)))) = \texttt{update}(\phi(s), \mathbf{next}_c(\phi(s),\phi(a),\psi,\mathbf{valid}(\phi(s),\phi(a),\psi)))$$

As seen in Definition 5.3 the first part of the definition ensures that the `next_state` is non-inspective. By the translation in Definition 7.1, in Makina models this property applies to functions `update`, $\mathbf{next}_c$, and $\mathbf{valid}_c$.

2. $\forall s,\alpha,\psi,\phi \ \ . \ \ s \in \mathbf{S}, \alpha \in \mathit{commands}(m), \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d) \ \ . \ \ \alpha \in \{c \in [\![\mathbf{command}(s)]\!] \mid [\![\texttt{precondition}(s,c)]\!]\}$
$\iff \phi(\alpha) \in \{c \in [\![\mathbf{command}(\phi(s))]\!] \mid [\![\texttt{precondition}(\phi(s),\phi(c))]\!]\} \iff$
$\forall s,\alpha,\psi,\phi \ \ . \ \ s \in \mathbf{S}, c \in \mathit{commands}(m), \psi \in \mathbf{V}, \phi : \mathbf{V} \to T(\Sigma_d) \ \ .$
$\alpha \in \{ \ \{\texttt{:call}, M, c, a\} \mid \mathbf{pre}_c(s) \wedge a \in \mathbf{args}_c(s)\} \wedge \mathbf{valid\_args}_c(s,a)$
$\iff \phi(\alpha) \in \{ \ \{\texttt{:call}, M, c, \phi(a)\} \mid \mathbf{pre}_c(\phi(s)) \wedge \phi(a) \in \mathbf{args}_c(\phi(s))\} \wedge \mathbf{valid\_args}_c(\phi(s),\phi(a))$

The second predicate in Definition 5.3 applies non-inspection on functions that generate command calls. We see that after applying the translation in Definition 7.1 the property applies to $\mathbf{pre}_c$, $\mathbf{args}_c$ and $\mathbf{valid\_args}_c$.

7.2. Type checking for makina models

During translation into a functional-style machine, a number of sanity checks are applied to the Makina model. For example, the name spaces for commands, invariants, parameters, and state attributes must be distinct.

Moreover, types decorating state attributes and command arguments are carefully preserved by the translation to facilitate the detection and diagnosis of model errors. There have been multiple efforts to construct static type checking tools on top of the Elixir programming language [8,32,33] and also to add runtime type-checking [9,34]. Clearly, it would be desirable to use static type-checking tools to detect faulty models at compile-time, but at the time of writing, all such tools have major problems. A principal problem is that none of the tools understand the *dynamics* of state machines, e.g., that a state returned by the **next** callback will later be used as a parameter to the **pre** callback. Moreover, states are difficult to type, as their contents vary when used during test generation and when used during test execution. Considering e.g. the Dialyzer [8] it is a type checking tool for Erlang programs, which means that reported type specification and type errors can be hard to understand for an Elixir programmer.

The work reported in [33] on a new type system for Elixir is promising, but at the present time the implementation in Elixir (v1.18) is incomplete. Given the (current) drawbacks of such static type-checking tools, we have instead chosen to rely primarily on runtime type assertion checking to detect errors. In the following, we discuss how such runtime type assertions, of the form `conforms_to`(*value*, *type*), are generated during the translation to a functional-style state machine.

Symbolic type annotations One of the main sources of errors in models is related to Definition 5.3. That is, the transition function that computes the next state, or the generator that consumes the state, may violate the non-inspectiveness property. This is usually due to a misunderstanding of the different phases in which the model state is used during the testing process. That is, at test generation time the state may contain symbolic variables (such states will be referred to as *symbolic*) representing future results from the system-under-test, whereas at test execution time states do not contain symbolic variables (such states will be referred to as *dynamic*).

In order to be able to give a unique type for states, irrespective of whether they are static or dynamic, Makina introduces the `symbolic` annotation, i.e., `symbolic(`*type*`)`, where *type* can be either a simple Elixir type, such as `boolean()`, or a parametric type, like `List.t(integer())`. Such a symbolic annotation `symbolic(`*type*`)` is used when a part of a state either is a symbolic variable, or a value of type *type*. Thus, in a Makina model, a unique type can be provided for a model state. As we shall see in the following, during the automatic translation to a functional-style state machine, this unique type is translated into different runtime assertions depending on whether the state is used in a symbolic or dynamic context.

The syntax of the types admitted in Makina is then given by the following grammar:

$$\langle\text{symbolic}\rangle ::= \texttt{symbolic}(\langle\text{type}\rangle)$$

$$\langle\text{type}\rangle ::= \textit{Elixir type}$$

In a Makina model, state attribute types can contain such symbolic annotations, and types for command parameters can contain them too, as symbolic variables can migrate from state attributes to command parameters. The translation from a Makina model to a functional-style state machine automatically transforms a state attribute type *type* into a runtime assertion check `conforms_to`(*value*, *type*′) injected into any callback function that accepts a state as a parameter or returns a state, where *type*′ does not contain any symbolic annotation.

Note that the same type will be translated into different runtime assertions for different callback functions. Let us refer to a callback function (e.g., **args**) which is applied only during test generation as providing a static context, a callback function (e.g., **post**) which is applied only during test execution as providing a dynamic context, and a callback function (e.g., **next**) which is applied during both test generation and execution as providing a mixed context.

The context-dependent translation of types into runtime assertions uses two functions defined below, which transform a type with symbolic annotations into a type with no symbolic annotations. The function sym(*type*) strips the type inside the symbolic annotation, leaving only the symbolic type itself, e.g., `symbolic()`.[12] The function *dyn*(*type*) strips the symbolic annotation, leaving the type itself. For example, *sym*(`symbolic(integer())`) = `symbolic()` and *dyn*(`symbolic(integer())`) = `integer()`.

$$\begin{aligned} sym(\texttt{symbolic(\_)}) &= \texttt{symbolic()} & dyn(\texttt{symbolic}(t)) &= t \\ sym(n\,(t_1,\dots,t_n)) &= n\,(sym(t_1),\dots,sym(t_n)) & dyn(n\,(t_1,\dots,t_n)) &= n\,(dyn(t_1),\dots,dyn(t_n)) \\ sym(t) &= t \quad \text{otherwise} & dyn(t) &= t \quad \text{otherwise} \end{aligned}$$

Thus, in static contexts, the predicate `conforms_to`(*value*, *sym*(*type*)) is injected as a runtime assertion in the code. In a dynamic context, the predicate `conforms_to`(*value*, **dyn**(*type*)) is asserted, and in a mixed context, such as **`next`**, the predicate:

```
    ( conforms_to(result, symbolic())
      and conforms_to(state, sym(t_s))
      and conforms_to(arguments, sym(t_a)) )
or
    ( conforms_to(result, t_r)
        and conforms_to(state, dyn(t_s))
        and conforms_to(arguments, dyn(t_a)) )
```

is asserted, where t_r is the type of the result, t_s the type of the state, and t_a the type of the arguments. That is, either the result is a symbolic variable, in which case the variable parts of the state and arguments must be symbolic too, or the result is a normal Elixir value, and then the variable parts of the state and arguments are normal Elixir values too.

As an illustrative example, let us recall the example in Fig. 5 in Section 6. Lines 8-9 show the state attributes `users` and `tokens`. Values stored in `users` will be checked using the same runtime assertion in all callbacks, but the runtime assertion regarding `tokens` in e.g. the **`args`** callback will permit symbolic variables. If we forget to add the `symbolic` annotation in the type definition for `tokens`, the error will be detected during test generation when `tokens` in the **`next`** callback is assigned a list containing a symbolic variable on line 25.[13] As another example, if we forget to use a symbolic annotation in the type for the `token` parameter in the command definition in line 29, a runtime error would also likely result, as the generator in line 30 may choose one of the (symbolic) tokens stored in the state.

8. Makina operators

One of the major drawbacks of `eqc_statem` state machines is that they are monolithic. There are no built-in mechanisms for building a model from the composition of existing ones. Consider, for example, the model `Auth.Functional` in Fig. 3, which contains a test model for the authorization originally introduced in Sect. 3. Suppose we want to test a program that implements the document storage API (also defined in Sect. 3), and which uses the authentication API for authentication of destructive updates; what would the resulting test model look like? Using standard EQC or Proper, we would normally write a model that has as commands the union of the commands of these two APIs, and its state would include both the authentication API model state and the document API model state. Such a model has the significant disadvantage of becoming quite large, and moreover, suppose that we later implement a second API on top of the authentication API, and want to test this second API, then we would have to manually extract the authentication API commands from the combined model. In Makina, instead, the two models (authentication and documents) can be developed separately.

8.1. Extension, aliasing and hiding

The operation of extending a base machine `Mb` with another machine `Me` is accomplished by embedding a reference to `Mb` inside the `Me` machine, using the `extends` option:

```
defmodule Me do
  use Makina, extends: Mb
  ...
end
```

Let us illustrate the new model extension mechanism by implementing a new model `Docs.Makina`, depicted in Fig. 6, which implements `Docs`, the stateful API service for storing documents presented in Sect. 3. The model `Docs.Makina` furthermore extends the `Auth.Makina` model (from Fig. 5).

The `Docs.Makina` module is concerned only with commands and state attributes that are necessary to reason about the `Docs` API. That is, it does not explicitly refer to `tokens()` or `users()`. Instead, it focuses on dealing with storing, deleting, and retrieving documents.

Note first the state declaration in `Docs.Makina` (on line 10 in Fig. 6), which defines an attribute `docs`. Clearly, to check whether the implementation correctly implements the `Docs` API, the model needs to remember the documents stored in the system, and thus the attribute type is a map from `key()` to `doc()`.

[12] An Elixir type that recognizes symbolic variables.

[13] Recall that `symbolic(elem(result, 1))` is not a type annotation, but represents a call to `elem(result, 1)`, which is delayed until the `result` is not a symbolic variable (during test execution).

```
1   defmodule Docs.Makina do
2     use Makina, extends: Auth.Makina,
3                 implemented_by: Docs,
4                 where: [val: :put, val: :del],
5                 hiding: [:val]
6
7     @type key() :: Docs.key()
8     @type doc() :: Docs.doc()
9
10    state docs: %{} :: %{ key() => doc() }
11
12    command put(key :: key(), doc :: doc()) :: :ok | :error do
13      args key: integer(), doc: string()
14      next if valid, do: [docs: Map.put(docs, key, doc)]
15    end
16
17    command del(key :: key()) :: :ok | :error do
18      args key: oneof([integer()|Map.keys(docs)])
19      valid key in Map.keys(docs)
20      next if valid, do: [docs: Map.delete(docs, key)]
21    end
22
23    command get(key :: key()) :: {:ok, doc()} | :error do
24      args key: oneof([integer()|Map.keys(docs)])
25      post result == Map.fetch(docs, key)
26    end
27  end
```

Fig. 6. The `Docs` model implemented as a composite model.

The aliasing and the hiding operations can be utilized only within an extending operation. Suppose that we wish to create two copies of the command `cmdA`, named `cmd1`, and `cmd2`, and afterwards hide the command `cmdA`. This can be accomplished using the machine:

```
defmodule Me do
  use Makina,
    extends: Mb,
    where: [cmdA: :cmd1, cmdA: :cmd2],
    hiding: [:cmdA]
  ...
end
```

This is equivalent to first applying the aliasing operation, then applying the hiding operation, and finally extending the resulting machine with the (possibly empty) model `Me`.

Recalling our running example, `Docs.Makina` defines three commands (`put`, `del` and `get`). Two of these commands correspond to authenticated operations, which means that the caller must be registered and authenticated and that a valid `token` should be used when calling this command. In the extending model `Auth.Makina`, we have already defined an operation that requires a valid token `val`. In Makina, commands can be renamed in the extending model by using the expression:

```
where: [val: :put, val: :del]
```

The effect of the aliasing operation is that the base model contains two commands `put` and `del` that use the command `val` from the extending model as a base implementation. Next, the `Docs.Makina` model defines its own `put` and `del` commands (starting on lines 12 and 17). Thus, callbacks *not defined* in the commands `put` and `del` in `Docs.Makina`, but present in `Auth.Makina`, are automatically included in the `Docs.Makina` model. These are, thus, the **`post`** callback, and for `put` the **`valid`** callback. Note also that the new commands also introduce new arguments to the base command. For example, the resulting arguments for command `put` are the ones in its definition (`key` and `doc`) plus the arguments declared in the base command `val` (`token`). Next, note the definition of **`valid`** in `del`; this definition *complements* the earlier definition in `Auth.Makina` by, in addition to checking that the token exists, also checking that the generated key is present in the model state.

For example, as a result of the aliasing operation, the composite model (the result of extending `Auth.Makina` by `Docs.Makina`), can be considered to contain the following (composite) `del` command[14]:

```
command del(token :: token(), key :: key()) :: :ok | :error do
  args let args1 <- [token: oneof([pos_integer()|tokens])],
           args2 <- [key: oneof([integer()|Map.keys(docs)])],
       do: prio_merge([args2,args1])
  valid (token in tokens) and (key in Map.keys(docs))
```

[14] The function `prio_merge` used in **`args`** is explained in Sect. 9.

```
  next if valid, do: [docs: Map.delete(docs, key)]
  post if valid, do: result == :ok, else: result == :error
end
```

Note that the resulting model will check that both the basic functionality of the documentation API is correctly implemented by the system-under-test, and second that the destructive operations (`put` and `del`) are protected by a token credential acquired in the correct manner by the system-under-test using the authentication API. However, these two properties are nicely separated into two separate models, so that, e.g., the logic concerning authentication is not mixed up with the documentation handling logic in the same test model.

Note that the `extends` construct can be used to define cyclical dependencies between test models. This is a test model error that will be detected when the corresponding modules are compiled.

8.2. Composition

The composition of machines can only take place within an `extends` construct. Suppose that we want to compose the machines `M1`, `M2` and `M3` into a single machine, this must be specified as follows:

```
defmodule Me do
  use Makina, extends: [M1,M2,M3]
  ...
end
```

This is equivalent to first composing `M1`, `M2` and `M3` into a new machine M_m, and then extending M_m with `Me`. Note, however, that the machine `Me` can be empty, i.e., not define any attributes nor any commands. In such a case, its whole behaviour is defined by the composition of `M1`, `M2` and `M3`.

Now, let us take another approach to writing the model `Docs.Makina`. This time instead of a single model that contains the complete definition of `Docs.Makina`, we have decoupled the definition into four models: `Put`, `Del` and `Get` contain the information to generate and execute the `put`,`del` and `get` commands, respectively; and the model `Docs.Makina` combines them. The result, showing only the `Get` and `Docs.Makina` models, is depicted below.

```
defmodule Get do
  use Makina, extends: Auth.Makina, implemented_by: Docs

  state docs: %{} :: %{ Docs.key() => Docs.doc() }

  command get(key :: Docs.key()) :: {:ok, Docs.doc()} | :error do
    args key: oneof([integer()|Map.keys(docs)])
    post result == Map.fetch(docs, key)
  end
end
...
defmodule Docs.Makina do
  use Makina, extends: [Put, Del, Get]
end
```

The two modelling styles (monolithic, and separate commands in separate models) result in equivalent models, but the latter approach allows to easily experiment with variants such as only testing interactions between `put` and `get` (by omitting the mention of `Del` in the model `Docs.Makina`).

Another modelling style supported by the composition operator separates the generation of test data from the judgement whether an execution is correct. In addition, monolithic generators can be split. Combining these two ideas, let us reconsider the `Get` model. In the **`args`** callback, there are two generators merged together in a list: `integer()` that generates a random integer key, and `oneof(Map.keys(docs))` that takes one of the keys stored in the model state. We can split these generators into two models `RandomGet` and `ValidGet` that contain the generators described above, respectively:

```
defmodule RandomGet do      # Likely invalid get commands
  use Makina, extends: Auth.Makina, where: [val: :get]
  command get(key) do
    args key: integer()
  end
end

defmodule ValidGet do       # Valid (good key) get commands
  use Makina, extends: Auth.Makina, where: [val: :get]
  command get(key) do
    pre docs != %{}
    args key: oneof(Map.keys(docs))
  end
end

defmodule Get do
```

```
  use Makina, extends: [RandomGet, ValidGet], implemented_by: Docs
  command get() do
    post result == Map.fetch(docs, key)
  end
end
```

Note that the model `ValidGet` needs a new precondition that requires that there is at least one document stored, otherwise the **`args`** generator would fail.

9. The semantics of the makina operators

In this section, we define the syntax and semantics of the state machine operators for modifying state machines (models) and combining them.

More concretely, we introduce (i) an extension operation where a base state machine is composed with an extending state machine, resulting in a composite new state machine, (ii) an aliasing operation for commands (i.e., a command is imported into a state machine with a new name), and (iii) a hiding operation for commands (i.e., removing a command from a state machine). Finally, (iv) a composition operation is introduced whereby a set of state machines is combined into a single one. In a sense, our hiding and composition operators are similar to the classical hiding and parallel composition operators found in process algebras such as LOTOS [21] and ACP [22]. However, the command aliasing operation is slightly different, and the extension operation is more novel.

In the following, we present the syntax of an operator and describe how model(s) operated upon are translated to a single model without occurrences of the operator.

9.1. Extension

Given a Makina model m_b (the base model), this operator extends its functionality by expanding it with another model m_e (the extending model). A typical extension supported by the operator is that the model m_e adds new parameters to a command already present in m_b. As a consequence of the modified command, it is likely that a number of command callbacks in m_b need to be redefined as well, minimally the **`args`** callback (to generate the extra parameters). In the following, we define formally how the new composite machine is constructed from the machines m_b and m_e, detailing its resulting state, commands, invariants, and callback functions. The following grammar shows the valid syntax for an extension:

$$\langle\text{extends}\rangle ::= \texttt{extends:}\ \langle\text{module}\rangle$$

$$\langle\text{module}\rangle ::= \textit{Elixir module name}$$

In the following we use the notation $m_e \gg m_b$ to signify m_e extends m_b.

Definition 9.1. Given two models m_e and m_b, let the extended model m be the result of $m_e \gg m_b$ where:

- the initial state $\textit{init}(m)$ is the union of the states $\textit{init}(m_b) \cup \textit{init}(m_e)$,
- the set of invariants $\textit{invariants}(m)$ is the union of invariants $\textit{invariants}(m_b) \cup \textit{invariants}(m_e)$,
- the set of commands $\textit{commands}(m)$ is the union of commands only defined in one of the models
  $\{c \in \textit{commands}(m_b) \mid c \notin \textit{commands}(m_e)\} \cup \{c \in \textit{commands}(m_e) \mid c \notin \textit{commands}(m_b)\}$
  plus the set of extended commands which correspond to commands defined in both models
  $\{c_e \gg c_b \mid c_b \in \textit{commands}(m_b) \wedge c_e \in \textit{commands}(m_e)\}$
  where c_e and c_b refer to commands in m_e and m_b respectively that share the name c and $c_e \gg c_b$ results in a new extended command.

Definition 9.2. Given two commands e and b, let the extended command c be the result of $e \gg b$ where:

- the set of arguments $\textit{arguments}(m)$ is the union of the arguments $\textit{arguments}(c_b) \cup \textit{arguments}(c_e)$,
- predicates **`pre`**, **`valid_args`**, **`valid`** and **`post`** are conjunctive:

$$\begin{aligned}
\texttt{pre}(s) &\iff \texttt{pre}_b(s) \ \wedge\ \texttt{pre}_e(s)\\
\texttt{valid\_args}(s,a) &\iff \texttt{valid\_args}_b(s,a) \ \wedge\ \texttt{valid\_args}_e(s,a)\\
\texttt{valid}(s,a,\psi) &\iff \texttt{valid}_b(s,a,\psi) \ \wedge\ \texttt{valid}_e(s,a,\psi)\\
\texttt{post}(s,a,r,v) &\iff \texttt{post}_b(s,a,r,v) \ \wedge\ \texttt{post}_e(s,a,r,v)
\end{aligned}$$

- **`call`** is defined as $\texttt{call}_e$ if defined and $\texttt{call}_b$ otherwise,
- **`args`** is defined as the generator that results of merging the arguments generated by c_b and c_e, this ensures that every value generated by **`args`** contains a value for each parameter in the command. Finally, **`next`** merges the updates defined in both models:

$$\texttt{args}(s) \overset{\text{def}}{=} \{\text{prio\_merge}(\,[a_e, a_b]\,) \mid a_e \in \texttt{args}_e(s), a_b \in \texttt{args}_b(s)\}$$

$$\texttt{next}(s, a, \psi, v) \overset{\text{def}}{=} \text{prio\_merge}(\,[\texttt{next}_e(s, a, \psi, v), \texttt{next}_b(s, a, \psi, v)]\,)$$

The prio_merge($[l_1, \dots, l_n]$) function used above merges the keyword lists in $l_1, \dots, l_n$ with precedence (if a key is defined multiple times) to the first list $l \in l_1, \dots, l_n$. That is, in the above definition of e.g. **`args`**, parameter values generated from the model m_e have precedence over the values generated from m_b. We define:

$$\text{prio\_merge}(\,[l_1, \dots, l_n]\,) = \bigcup_{1 \le i < n} \{x : v \mid x : v \in l_i \;\wedge\; \neg\exists v' \,.\; x : v' \in \bigcup_{1 \le j < i} l_j\}$$

Definition 9.3. Given two non-terminating models m_b and m_e, the model $m = m_e \gg m_b$ is also ***non-terminating*** if the predicate in Definition 7.2 holds. That is, for every given state $s \in \mathbf{S}$, the model is able to generate a new command, thus there is a command whose **`pre`** holds and **`args`** is able to generate at least one set of arguments accepted by **`valid_args`**.

9.1.1. Refinement for the extension operator

The extension operator is powerful, but does not preserve some desirable properties of the base machine m_b. For example, if a test case tc can be generated by m_b there is no guarantee that a composite machine $m_e \gg m_b$ can also generate the same test case, or a "compatible" one (in case commands receive new parameters). However, it is possible to restrict extending models m_e in various ways, which permits us to conclude that a model m which is the result of extending m_b by m_e, is a *refinement* of the original model m_b.

Preliminaries Let $attributes(M)$ be the set of names of attributes of a model M and $commands(M)$ the set of its command names. For a command name c, let $params(c)$ be the set of names of its parameters. Let $writes(M)$ be the set of attributes that are written by a model M, i.e., that are mentioned in a **`next`** function in any of its commands. Let $name(c)$ refer to the name of the command c.

Given a test case t and a model m, we define the restriction operator $t\backslash m$ as follows:

$$\begin{aligned}
\epsilon\backslash m &\overset{\text{def}}{=} \epsilon \\
(\alpha \cdot t)\backslash m &\overset{\text{def}}{=} t\backslash m \text{ if } name(\alpha) \notin commands(m) \\
(\alpha \cdot t)\backslash m &\overset{\text{def}}{=} \alpha \cdot (t\backslash m) \text{ if } name(\alpha) \in commands(m)
\end{aligned}$$

That is, the restricted test case contains only commands defined in m. Similarly, we define the restriction of an execution r of a test t with respect to a model m ($r\backslash m$), as the sequence r' which removes from r all results corresponding to the execution of a command $n \notin commands(m)$ in t.

Moreover, given a model state s (a map from attribute names to values), let $s\backslash m$ be the map restricted to attribute names of m.

Now, let us relax the notion of refinement by permitting a refining model to define new commands, which do not interfere with the commands shared between the models:

Definition 9.4. Given two QuickCheck state machines m and m' we say that m' ***refines*** m when for all system-under-tests f:

$$\begin{aligned}
&\forall t \in \mathbf{G}_m \;\;.\;\; \exists t' \in \mathbf{G}_{m'} \,.\, t = t'\backslash m \\
&\forall t \in \mathbf{G}_{m'}, r \in f(t) \;\;.\;\; \mathbf{P}_{m'}(t, r) \wedge t\backslash m \in \mathbf{G}_m \implies \mathbf{P}_m(t\backslash m, r\backslash m))
\end{aligned}$$

That is, (i) if a test case t can be generated from m then there should exist a test case t' that can be generated by m' such that t' restricted by m is identical to t. Moreover, (ii), if the model m' deems that an execution r of a test case t generated by m' is correct, and moreover the test case t restricted by m can be generated by m, then the model m deems that the execution of r restricted by m is correct too.

Proving refinement Let us attempt to prove that $m_e \gg m$ refines m under the relaxed notion of refinement introduced in Definition 9.4, and under the constraints that the two models define separate commands, and that m_e does not write to parameters accessible to m.

Theorem. Given two models m_e and m such that $commands(m) \cap commands(m_e) = \emptyset$ and $writes(m_e) \cap attributes(m) = \emptyset$, then $m_e \gg m$ refines m according to the definition of refinement in Definition 9.4.

Proof. For brevity, we will refer to $m_e \gg m$ as m' in the following. First, observe that due to the definition of the extension operator it holds that for every command $n \in commands(m)$ its corresponding callbacks, e.g., **`pre`**/1, **`valid_args`**/2, **`args`**/1, **`post`**/3 and **`next`**/3, in m' are fully determined by the module m (as these commands are not defined by m_e). For example, $m'.n\_\texttt{pre}(s) = m.n\_\texttt{pre}(s)$. Moreover, for any state s of m', and all these functions, it holds that their behaviour is defined fully by the portion state visible to m. For example, $m'.n\_\texttt{pre}(s) = m'.n\_\texttt{pre}(s\backslash m) = m.n\_\texttt{pre}(s\backslash m)$.

Consider the first property, i.e.,

$$\forall t \in \mathbf{G}_m \ . \ \exists t' \in \mathbf{G}_{m'} \ . \ t = t' \backslash m$$

We assume the existence of a test case t of m and proceed to construct a new test case t' of m' which under restriction by m is identical to t. The proof will show that the state of m' can be constructed to be identical to the state of m when considering only the state attributes in m. Let $size(t)$ be the size of the test case, i.e., its number of commands. Furthermore, define $result(t, m, s)$ (where s is a state of m and t is a test of m) as the model state resulting from following the transition function of m:

$$\begin{array}{lcl} result(\epsilon, m, s) & \overset{\text{def}}{=} & s \\ result(\langle \alpha, \psi \rangle \cdot t', m, s) & \overset{\text{def}}{=} & result(t', m, s') \text{ if } s \xrightarrow[\alpha]{\psi} s' \end{array}$$

We prove by induction over the size of the test cases that: if a test case t can be generated by m in a state s of m, then, for every state s' such that $s' \backslash m = s$, t can also be generated by m' in s'. Moreover, $result(t, m', s') \backslash m = result(t, m, s)$.

- Base case $n = 0$: the empty test case can be generated by m in a state s, and trivially it can also be generated by m' in (every) state s'. Moreover, executing an empty test does not modify the model state.
- Induction case, size is $n + 1$: let $t = \langle \alpha_1, \psi_1 \rangle \cdots \langle \alpha_n, sv_n \rangle \cdot \langle \alpha_{n+1}, sv_{n+1} \rangle$. We may assume that if $t' = \langle \alpha_1, \psi_1 \rangle \cdots \langle \alpha_n, sv_n \rangle$ can be generated by m then, for every state s' of m' such that $s' \backslash m = s$, then so is it by m', and moreover, that $result(t', m', s') \backslash m = result(t', m, s)$. That the command α_{n+1} can be generated by m in state $s = result(t', m, s)$ means that $m.name(\alpha_{n+1})$`_pre`(s) evaluated to **`true`**, and so on (for **`args`**, and **`valid_args`**), but then $m'.name(\alpha_{n+1})$`_pre`(s) must also evaluate to **`true`**, and so on (for **`args`**, and **`valid_args`**).
  Moreover, the updates of the state attributes returned by **`next`** are the same for m' and m. Thus, the resulting states s'_n and s_n after a transition α_n, from states s' and s such that $s' \backslash m = s$, must still be equivalent, i.e., $s'_n \backslash m = s_n$.

Finally, we can deduce that if a test case t is derivable from m', it is also derivable from m since their initial states s_{m_0} and s'_{m_0} meet the requirement that $s_{m'_0} \backslash m = s_{m_0}$.

Consider next the second property, i.e.,

$$\forall t \in \mathbf{G}_{m'}, r \in f(t) \ . \ \mathbf{P}_{m'}(t, r) \wedge t \backslash m \in \mathbf{G}_m \implies \mathbf{P}_m(t \backslash m, r \backslash m))$$

We prove this property using induction on the size of $t \backslash m$, i.e., that if a test case t can be generated by m' in a state s, and $r \in f(t)$ for a system-under-test r, and t can also be generated by m in the state $s \backslash m$, then $P_{m'}(t, r)$ implies that $P_m(t \backslash m, r \backslash m)$, and $result(t, m', s) \backslash m = result(t \backslash m, m, s \backslash m)$.

- The base case ($size(t' \backslash m) = 0$) is completely trivial (the postcondition predicate for an empty result is trivially true).
- A test of t' of m' such that $size(t' \backslash m) = n + 1$ has the shape

  $$\alpha_{m'}, \ldots, \alpha_{m'}, \alpha_{m_1}, \ldots, \alpha_{m'}, \ldots, \alpha_{m'}, \alpha_{m_2}, \ldots$$

  i.e., every action α_m of m may be preceded and followed by an arbitrary number of actions $\alpha_{m'}$ of m'. Since $t' \backslash m$ has size $n + 1$, then let us consider the prefix of t' that ends with the nth action of m: α_{m_n}. We can assume that the property holds for this test case prefix, and that the resulting states are equal for the state portion visible to m. It remains to prove that the suffix

  $$\alpha_{m'}, \ldots, \alpha_{m'}, \alpha_{m_{n+1}}, \alpha_{m'}, \ldots, \alpha_{m'}$$

  satisfies the postcondition predicate, and that the resulting states are the same (up to the attributes in m). Consider the first command $\alpha_{m'}$ after α_{m_n}. Since $writes(m_e) \cap attributes(m) = \emptyset$, we know that the state visible to m does not change after taking an arbitrary command $\alpha_{m'}$ not in m, so if the states visible to m were equal before taking the transition, they are still equal after taking such a transition. The same argument holds for all such commands $\alpha_{m'}$ before α_{m_n}. Consider now the trace terminating in α_{m_n} (with the states s' and s before taking that action such that $s' \backslash m = s$). That $\mathbf{P}_{m'}(t, r)$ holds for the trace terminating after α_{m_n} means that **`post`** must hold for α_{m_n} in state s'. But **`post`** for m' is defined in terms only of m and thus **`post`** for m in state s also. Moreover, the next states are equal (for the state portion visible to m), since **`next`** for m' is defined in terms of **`next`** for m. It remains to treat the suffix of commands $\alpha_{m'}, \ldots, \alpha_{m'}$ where $\alpha_{m'}$ is a command of m_e only. Using the same argument as before clearly taking such a command does not change the equality of the resulting states. Moreover, since $t \backslash m$ eliminates such commands, then if the postcondition predicate for m was true for the (restricted) prefix of t not including these actions, it must be true for the entire (restricted) test case t. $\square$

9.2. Aliasing

The aliasing operator permits renaming commands in a model. It is written as a list of pairs where in each pair the first element is the old name and the second one is the new name of the command. The following grammar specifies the allowed syntax when using command aliases:

$$\langle\text{extends}\rangle ::= \texttt{extends:}\ \langle\text{module}\rangle\ \texttt{, where: [}\langle\text{names}\rangle\texttt{]}$$
$$\langle\text{names}\rangle ::= \langle\text{atom}\rangle\langle\text{atom}\rangle\ (\texttt{,}\langle\text{atom}\rangle\langle\text{atom}\rangle)^*\ |\ \epsilon$$
$$\langle\text{atom}\rangle ::= \textit{Elixir atom}$$
$$\langle\text{module}\rangle ::= \textit{Elixir module name}$$

Definition 9.5. Given a model m_b and a substitution of command names ϕ, let m be the result of $\phi(m_b)$ where $\mathit{commands}(m) = \phi(\mathit{commands}(m_b))$ and the rest components of m_b remain unchanged.

9.3. Hiding

The hiding operator is used to remove a list of commands from a model. The following grammar specifies the syntax when hiding commands:

$$\langle\text{hiding}\rangle ::= \texttt{extends:}\ \langle\text{module}\rangle\texttt{, hiding: [}\langle\text{names}\rangle\texttt{]}$$
$$\langle\text{names}\rangle ::= \langle\text{atom}\rangle\ (\texttt{,}\langle\text{atom}\rangle)^*\ |\ \epsilon$$
$$\langle\text{atom}\rangle ::= \textit{Elixir atom}$$
$$\langle\text{module}\rangle ::= \textit{Elixir module name}$$

Definition 9.6. Given a model m_b and a set of command names *cmds*, let m be the result of $m_b \setminus \mathit{cmds}$ where $\mathit{commands}(m) = \{c \in \mathit{commands}(m_b) \mid c \notin \mathit{cmds}\}$ and the rest of the model remains unchanged.

9.4. Composition

Given a sequence of models $m_1, \dots, m_n$, the composition operation constructs a new model m that codifies the combined behaviour of the models. The following grammar shows the valid syntax:

$$\langle\text{composition}\rangle ::= \texttt{extends: [}\langle\text{modules}\rangle\texttt{]}$$
$$\langle\text{modules}\rangle ::= \langle\text{module}\rangle\ (\texttt{,}\langle\text{module}\rangle)^*\ |\ \epsilon$$
$$\langle\text{module}\rangle ::= \textit{Elixir module name}$$

In the following, we use the more intuitive syntax $m_1||m_2$ to express the composition of two (or more) models m_1 and m_2.

Definition 9.7. Given the models $m_1, \dots, m_n$, let the composed model m be the result of $m_1||\dots||m_n$ where:

- the initial state $\mathit{init}(m)$ is the union of the states in all models $\bigcup_{1\le i\le n} \mathit{init}(m_i)$,
- the set invariant of invariants $\mathit{invariants}(m)$ is the union of invariants in all models $\bigcup_{1\le i\le n} \mathit{invariants}(m_i)$,
- the set of commands $\mathit{commands}(m)$ is the union of the commands unique to each model
  $\{c \in \mathit{commands}(m_i) \mid 1 \le i \le n\ \wedge\ 1 \le j \le n \wedge i \ne j\ \wedge\ c \notin \mathit{commands}(m_j)\}$
  plus the set of the composed commands
  $\{c_1||\dots||c_n \mid c_1 \in \mathit{commands}(m_1) \wedge \dots \wedge c_n \in \mathit{commands}(m_n)\}$
  where $c_1, \dots, c_n$ refer to commands in $m_1, \dots, m_n$ that share the name c and $c_1||\dots||c_n$ results in a new composed command.

Definition 9.8. Given the commands $c_1, \dots, c_n$, let the composed command c be the result of $c_1||\dots||c_n$ where:

- The callback `pre` is disjunctive, i.e., if at least on of the commands has a true `pre`, then so does the composed model:

$$\mathbf{pre}(s) \iff \bigvee_{1\le i\le n} \mathbf{pre}_i(s)$$

- Generation of command arguments is the random choice of the enabled commands, by enabled we mean its `pre` holds. This is because the enabled models are those allowed to be executed by `pre` and all of them generate the same arguments:

$$\mathbf{args}(s) \overset{\text{def}}{=} \{a \in \mathbf{args}_i(s) \mid \mathbf{pre}_i(s)\}$$

- The `call` callback executes exactly one of the possible commands with an implementation of the `call` callback. Note that this is, in a sense, an arbitrary choice. We could instead have chosen to execute all executable commands with an implementation of the callback. It is largely a matter of modelling style preference which type of callback is preferred.

- The callback **`valid_args`** in c is conjunctive on the composed commands $c_1, \dots, c_n$ and thus **`valid`** and **`post`** are conjunctive too.

$$\begin{aligned}\texttt{valid\_args}(s,a) &\iff \bigwedge_{1\le i\le n} \texttt{valid\_args}_i(s,a)\\ \texttt{valid}(s,a,\psi) &\iff \bigwedge_{1\le i\le n} \texttt{valid}_i(s,a,\psi)\\ \texttt{post}(s,a,r,v) &\iff \bigwedge_{1\le i\le n} \texttt{post}_i(s,a,\psi,v)\end{aligned}$$

- Finally, the updates of a command c are the result of merging the updates in the commands $c_1, \dots, c_n$. It is required and checked that updates in commands $c_1, \dots, c_n$ are *compatible* updates. That is, if an attribute a is updated by multiple commands in $c_1, \dots, c_n$, then a must receive the same value from all models.

$$\texttt{next}(s,a,\psi,v) \overset{\text{def}}{=} \text{prio\_merge}(\,[\texttt{next}_1(s,a,\psi,\texttt{valid}_i(s,a,\psi)),\dots,\texttt{next}_n(s,a,\psi,\texttt{valid}_i(s,a,\psi))]\,)$$

  Note that this definition uses the callback **`valid`** that is local to each command c_i instead of the composed one in c. The reason for this is that a generated command can be valid for one of the individual models, and invalid for another one.
  As an example we could imagine a service API that charges for every syntactically correct order submitted to it, but does not execute an order that is not semantically correct (e.g., specifying a user that does not exist). In such a case we would like to write independent models for charging and fulfilling orders, so that for charging every syntactically order is valid, whereas for fulfilment far fewer orders are valid. As is the case for our composition operator that the validity of commands is specific to each model in the composition, the charging model would change its state, and the fulfilment model would not. If validity was interpreted to be a global concern, neither of the models would change their state.

Discussion It is clear that the composition operator defined in this section is just one of numerous such operators that can be defined, just as the choice of a particular parallel composition operator in process algebra is somewhat arbitrary. The present operator expresses the requirement that, when multiple machines implement the same command, for a command to be executable then all such machines must agree (the conjunction of the **`valid_args`** callbacks), and after the execution of the command they all move (execute the **`next`** callbacks) in unison. Thus, in a sense, a composition of machines sharing a command expresses a form of restriction on which test cases may be generated.

9.5. *Type checking for composed makina models*

Basic sanity checks are applied to composed Makina models, e.g., that modules being extended, or that occur in a parallel composition, exist. Moreover, runtime checks, as described in Section 7.2, are incorporated into the resulting functional-style state machine.

In addition to these checks, type compatibility across composed models is enforced during runtime. In Makina models, the concept of type compatibility is "conjunctive". If a model m_e extends a model m_b, and both models define a state attribute a, with two different type specifications t_{a_e} and t_{a_b}, then any value stored in attribute a must conform to both types t_{a_e} and t_{a_b}. The same principle applies to command arguments: if a command c is defined in both m_e and m_b, and has an argument p, then any value generated for that argument must conform to the type specifications for the argument p in the command c in both m_e and m_b.[15]

The same typing principle for submodels applies to attributes and command arguments for models in a parallel composition. For example, if a value v is stored in an attribute a defined in models m_1, m_2, and m_5 in a parallel composition $m_1 || \dots || m_n$, then v must conform to all three types t_{a_1}, t_{a_2}, and t_{a_5}.

10. Implementation

In this section, we describe how the translation of a Makina model into a functional-style state machine is implemented. The translation will, starting from an Elixir module M defining a Makina model, derive a module M`.StateM` that implements the state machine behaviour defined in Section 5. The translation is implemented using the Elixir macro facility. This is a challenging undertaking, as a Makina model is typically composed of different submodels using the model composition operators introduced in earlier sections. Each submodel is defined in its own Elixir module. Thus, in order to implement the checks described in earlier sections, a "whole-program" analysis of the set of Elixir modules comprising the Makina model is required. That is, the analysis of one module requires the analysis of its submodules to proceed first.

During translation, several additional Elixir modules will be created in addition to the M`.StateM` module. Each module implements some (sub)part of the model, e.g., one module encodes the state attributes, its invariants, and a third module contains the compiled code of its command callbacks and interfacing code.

In the following we describe how the Makina DSL uses Elixir's macro system in its implementation. Then, using the example in Fig. 5 from Section 6, the code produced by the compilation process is presented, illustrating how a Makina model is transformed into

[15] Recall that if no type specification is provided in a model the specification `any()`, which permits all values, is assumed.

a function-based state machine. The code example shows, for instance, how type information in a Makina model is translated into type annotations in the resulting state machine, where callback functions are provided with different type annotations depending on whether they are called during test case generation or execution.

10.1. Using the elixir macro facility to organize model translation

A macro in Elixir is a compile-time construct that receives an Elixir abstract syntax tree (AST) as an input and returns an Elixir AST as an output. Elixir macros are a versatile mechanism for executing code during compilation. The general use of Elixir macros is to extend the language by embedding DSLs.

After scanning and parsing, the Elixir compilation process can be divided into two main phases: macro expansion and translation. During macro expansion, calls to macros in the program being compiled are expanded (executed), and the resulting AST is injected into the program body. Note that macros themselves can return new macro calls, which in turn need to be expanded. The end result of macro expansion is that the resulting code contains user-written code (without calls to macros) and "special forms",[16] which constitute the main building blocks of Elixir. Once this point is reached, the second compilation phase starts, and the compiler translates the source code into BEAM-compatible object code that can be executed by the Erlang run-time system.

The Makina DSL uses the macro expansion mechanism to implement the translation of Makina models into EQC functional-style state machines as described earlier. The compilation of a Makina model is divided into three phases: (i) module configuration, (ii) module annotation, and (iii) code generation.

Module configuration The configuration is accomplished by the `__using__` macro defined in Makina, which is invoked by the *uses* directive (i.e., as in **`use`** `Makina`). The aim of this phase is to make the rest of Makina macros available in the invoking module, and to register the necessary module attributes that are required during the further macro expansion process. A module attribute is a compile-time register accessible by all macros that are being expanded in the same module. They are useful for storing and retrieving information during macro expansion. If the model that invokes **`use`** refers to other models using the `extends` option, the necessary information about these submodules is retrieved first. This is accomplished using reflection: modules containing Makina models implement a `__makina_info__` function, which returns information about the module type as described in Section 7. If any of these submodules have not yet been compiled, the compilation process of the current module is paused until the compilation of the submodules has been completed. The mechanisms used by the library to pause and resume the compilation of modules are the standard ones provided by the Elixir parallel compiler.

Module annotation The user accessible macros in Makina (e.g., the macros **`command`** and **`state`** which define commands and state attributes) have two roles: (i) checking that declarations of e.g. commands and attributes are syntactically correct; and (ii) annotating the module with the necessary information for the later code generation phase.

During this phase, if a syntax error is found, a compilation error is reported. The basic model validation checks described in Sections 7.2 and 9.5 are also performed. For example, if an extended model overrides an invariant of its base model, a compilation error is reported.

Code generation Makina defines a macro `__before_compile__`, which is called by Elixir immediately before compilation begins. This macro retrieves all the information stored in module attributes annotated by other macros. Then, using this information, the library proceeds to generate a functional-style EQC model, which is subsequently compiled by the normal Elixir compiler. The following subsection expands on the code generation phase.

10.2. Code generation

This Section demonstrates, using the `Auth.Makina` model in Fig. 5, how a Makina model is translated into a functional-style EQC model. In addition, the section shows how type annotations in the Makina model are translated into types and function type specifications. Such type specifications are subsequently used by the Corsa tool [34], which uses the generated type specifications to derive the concrete runtime type assertions introduced in Section 7.2.

Generating a module for the state The functions that implement the manipulation of Makina state attributes in the resulting EQC model are embedded in the module `Auth.Makina.State` generated by the translation as shown below (additional details of types and functions will be explained in the following running text):

[16] https://hexdocs.pm/elixir/Kernel.SpecialForms.html.

```
state users: %{} :: %{user() => pass()},
      tokens: [] :: [symbolic(token())]
```

→

```
1  defmodule Auth.Makina.State do
2    @type symbolic_state()
3    @type symbolic_update()
4    @type dynamic_state()
5    @type dynamic_update()
6    def users(), do: %{}
7    def tokens(), do: []
8    def initial() do
9      %{users: users(), tokens: tokens()}
10   end
11   def update(updates, state)
12   def validate(state)
13 end
```

For each attribute, a function with the same name returns the initial value (lines 6 and 7). These functions are used to define `initial()` (lines 8-10), the function that returns the initial state of the model. Moreover, type definitions `symbolic_state()` and `dynamic_state()` are synthesized from the state declaration using *sym* and *dyn* functions introduced in Section 7.2:

```
@type symbolic_state() :: %{users:  %{user() => pass()}, tokens: [symbolic()]}
@type dynamic_state()  :: %{users:  %{user() => pass()}, tokens: [token()]}
```

These are the two variants of the state type, which are used to type-check the rest of the generated code. As we have seen in Section 7, callbacks such as **`next`**, **`pre`**, or **`args`** use the variant of the state that may contain variables and symbolic values; the `symbolic_state()` type is appropriate for its function type specification. In contrast, invariants or the **`post`** callback are always called with ground states; thus, the `dynamic_state()` type is appropriate for its function type specification.

Actions that can be performed on the state include updates to its attributes. Updates can be applied to either the symbolic or dynamic state. The type definition for updates (lines 3 and 5) and the type specification for the update function (line 11) are as follows:

```
@type symbolic_update() :: {:users, %{user() => pass()}} | {:tokens, [symbolic()]}
@type dynamic_update() :: {:users, %{user() => pass()}} | {:tokens, [token()]}

@spec update([symbolic_update()], symbolic_state()) :: symbolic_state()
@spec update([dynamic_update()], dynamic_state()) :: dynamic_state()
```

Notice that the update function can be used with both the symbolic and the dynamic state. The only requirement imposed by the type specification is that when a symbolic state is updated, the result must remain symbolic, and the same applies to the dynamic state. The function `validate` (line 12) introduces run-time checks after updating or modifying the state. The Elixir code for the functions `update` and `validate` is omitted here.

Generating a module for commands The code generated from each command declaration is stored in a separate module. The generated code for the command `gen`, for example, is stored in the module `Auth.Makina.Command.Gen`:

```
1  command gen(user :: user(),
2              pass :: pass()) ::
3          {:ok, token()} | :error do
4    pre users != %{}
5    args let user <- oneof(Map.keys(users)),
6         do: [user: user,
7              pass: Map.get(users, user)]
8    valid {user, pass} in users
9    next if valid,
10        do: [tokens: [symbolic(elem(result, 1))
11                      |tokens]]
12   post if valid, do: match?({:ok, _}, result),
13                  else: result
14 end
```

→

```
1  defmodule Auth.Command.Gen do
2    @type symbolic_arguments()
3    @type dynamic_arguments()
4    @type result()
5    def pre(state)
6    def args(state)
7    def valid_args(state, arguments)
8    def valid(state, arguments, result)
9    def next(state, arguments, result)
10   def post(state, arguments, result)
11 end
```

Types in lines 1-3 store information about the arguments and the results of the commands. In this case, the arguments do not use symbolic information, as they are of the same type.

```
@type symbolic_arguments() :: %{ user: user(), pass: pass() }
@type dynamic_arguments()  :: %{ user: user(), pass: pass() }
@type result() :: {:ok, token()} | :error
```

Moreover, for each callback declared in the command, a function with the callback name is generated, copying the body from the callback. As explained in Section 7, callbacks **`pre`**, **`args`**, and **`valid_args`** only use symbolic information, so the generated function specification for these callbacks only refers to the symbolic types:

```
@spec pre(symbolic_state()) :: boolean()
def pre(%{users: users}) do users != %{} end
```

```
1   defmodule Auth.Makina.StateM do
2     def initial_state(), do: Auth.Makina.State.initial()
3
4     defp commands() do
5       [Auth.Commands.Reg, Auth.Commands.Gen, Auth.Commands.Rev, Auth.Commands.Val]
6     end
7
8     def commands(state) do
9       enabled = Enum.filter(commands(), fn cmd -> cmd.pre(state) end)
10      let [cmd <- oneof(enabled), args <- cmd.args(state)] do
11        {:call, cmd, :call, [Map.new(args)]}
12      end
13    end
14
15    def precondition(state, {:call, cmd, :call, [args]}) do
16      cmd.valid_args(state, args)
17    end
18
19    def next_state(state, result, {:call, cmd, :call, [args]}) do
20      valid = cmd.valid(state, args, result)
21      cmd.next(state, args, result, valid)
22      |> Auth.Makina.State.update(state)
23      |> Auth.Makina.State.validate()
24    end
25
26    def postcondition(state, {:call, cmd, :call, [args]}, result) do
27      valid = cmd.valid(state, args, result)
28      :ok = Auth.Makina.Invariants.check(state)
29      cmd.post(state, args, result)
30    end
31  end
```

Fig. 7. Generated function-style state machine for `Auth.Makina`.

```
@spec args(symbolic_state()) :: gen(symbolic_arguments())
def args(%{users: users}) do
  let user <- oneof(Map.keys(users)), do: [user: user, pass: Map.get(users, user)]
end
```

Callbacks **`call`** and **`post`** are never called with a symbolic state. Note that the call to the system-under-test in the **`call`** function preserves the order of the parameters from the command definition, line 20 in Fig. 5.

```
@spec call(dynamic_arguments()) :: result()
def call(%{user: user, pass: pass}) do Auth.gen(user, pass) end

@spec post(dynamic_state(), dynamic_arguments, result()) :: boolean()
def post(_state, _arguments, result) do
  if valid, do: match?({:ok, _}, result), else: result
end
```

Finally, **`valid`** and **`next`** are used in both contexts; therefore, we need to attach a type specification for both the symbolic and dynamic contexts.

```
@spec valid(symbolic_state(), symbolic_arguments(), symbolic_var()) :: boolean()
@spec valid(dynamic_state(), dynamic_arguments(), dynamic_state()) :: boolean()
def valid(%{users: users}, %{user: user, pass: pass}, result) do {user, pass} in users end

@spec next(symbolic_state(), symbolic_arguments(), symbolic_var(), boolean()) :: [symbolic_update()]
@spec next(dynamic_state(), dynamic_arguments(), result(), boolean()) :: [dynamic_update()]
def next(%{tokens: tokens}, _arguments, result, valid) do
  if valid, do: [tokens: [symbolic(elem(result, 1))|tokens]]
end
```

Note that if a callback function is missing (as is **`valid_args`** in our running example), the default implementation is introduced.

Putting all together: generating a functional-style state machine Fig. 7 illustrates the functional-style state machine generated for the `Auth.Makina` model. This module implements the `StateM` behaviour described in Section 5:

- The function `initial_state` (line 2) is redirected to the state module.
- The command generator (lines 8 to 12) filters out commands that cannot be generated from the current state, retaining only the enabled ones. It then creates a keyword list of arguments using **`args`** and converts this list into a map, enabling the extraction of arguments through pattern matching, as shown before.

```
1   defmodule Clock.Makina do
2     use Makina, implemented_by: Clock
3
4     state clocks: %{} :: %{optional(symbolic(pid())) => nil | symbolic(integer())}
5
6     command new() :: pid() do
7       next clocks: Map.put(clocks, result, nil)
8     end
9
10    command time(clock :: symbolic(pid())) :: integer() do
11      pre clocks != %{}
12      args clock: oneof(Map.keys(clocks))
13      next clocks: Map.put(clocks, clock, result)
14      post if not is_nil(clocks[clock]),
15             do: result == rem(clocks[clock], 12)
16    end
17
18    command tick(clock :: symbolic(pid())) :: :ok do
19      pre clocks != %{}
20      args clock: oneof(Map.keys(clocks))
21      next do
22        if not is_nil(clocks[clock]) do
23          value = symbolic(Kernel.+(clocks[clock], 1))
24          [clocks: Map.put(clocks, clock, value)]
25        end
26      end
27    end
28  end
```

Fig. 8. A Makina model for a trivial clock.

- The precondition (lines 15 to 17) is redirected to the **`valid_args`** function of the generated command.
- The transition function `next_state` (lines 19-24) evaluates the valid predicate (line 20), computes the next state (line 21), updates the state (line 22), and validates the new state (line 23).
- Similar to `next_state/3`, the `postcondition` (lines 26 to 30) introduces the result of the **`valid`** predicate, followed by a check of the postcondition and invariants.

11. Experiments

In this section, we evaluate the Makina specification formalism using a number of experiments. First, in Section 11.1, a trivial clock is modelled both using Makina and using a functional-style state machine. Next, mutations are manually introduced in both the system-under-test (the clock) and in the models, in order to compare the error messages produced when the system-under-test fails a test. The aim of this section is to empirically evaluate the quality of the error messages produced by the type analysis introduced in Section 7.2 through a comparison with the results produced by a number of comparable tools. In Section 11.2, Makina is applied to a larger example: testing that a Java data structure library is correctly implemented. This experiment heavily uses the Makina model composition operators, thus permitting us to examine whether the use of the operators contributes to readable models, and to what extent their use slows down model compilation and testing. Section 11.3 presents measurements on the time required to compile and execute a number of models.

The examples discussed in this section are available in the repository https://gitlab.com/babel-upm/makina/examples.git.

11.1. Trivial clocks

This section demonstrates the error detection mechanisms in Makina using an example based on Lamport's trivial clock introduced in [35]. In the example, a clock is implemented as an Elixir process, and users can interact with it through three functions: (i) `new()`, which creates a new clock with a random time (in 12-hour format) and returns the process identifier (PID) without revealing its time; (ii) the `time(clock)` operation, which retrieves the time of a clock; and (iii) `tick(clock)`, which increments the time of a clock.

Fig. 8 shows a Makina model to test the expected behaviour of a set of trivial clocks. Its state definition (line 4) contains the attribute `:clocks`, which maps clock PIDs to clock values. The clock values are encoded as either **`nil`** (the initial time of a clock that is unknown to the model) or an integer representing its known current time. Both clock PIDs and time values are annotated as symbolic values, as their initial values cannot be predicted during the generation phase due to the random nature of the clocks upon initialization. Lines 6-8 show the specification for `new`, which maps the created clock PID (accessible through `result`) to **`nil`**. Lines 10-16 define the specification for `time`: to generate calls to this command, at least one clock must be stored in the state (line 11). The clock passed as an argument is chosen from the clocks in the state (line 12), and after the call, the state is updated with the time given as a result (line 14). Lines 18-27 specify the behaviour of `tick`: precondition and argument generation are the same as in `time`; once the initial value of the clock is retrieved (line 21), the number of `tick` calls is tracked by storing a symbolic expression that represents the increment of the clock (lines 23-24).

```
defmodule Clock.Functional do
  use PropCheck

  def initial_state(), do: %{}

  def command(clocks) do
    if clocks == %{} do
      {:call, Clock, :new, []}
    else
      clock = oneof(Map.keys(clocks))

      oneof([
        {:call, Clock, :new, []},
        {:call, Clock, :time, [clock]},
        {:call, Clock, :tick, [clock]}
      ])
    end
  end

  def precondition(clocks, call) do
    case call do
      {:call, Clock, :new, []} -> true
      {:call, Clock, :time, [clock]} -> clock in Map.keys(clocks)
      {:call, Clock, :tick, [clock]} -> clock in Map.keys(clocks)
    end
  end

  def next_state(clocks, result, call) do
    case call do
      {:call, Clock, :new, []} ->
        Map.put(clocks, result, nil)

      {:call, Clock, :time, [clock]} ->
        Map.put(clocks, clock, result)

      {:call, Clock, :tick, [clock]} ->
        if not is_nil(clocks[clock]),
          do: Map.put(clocks, clock, {:call, Kernel, :+, [clocks[clock], 1]}),
          else: clocks
    end
  end

  def postcondition(clocks, call, result) do
    case call do
      {:call, Clock, :time, [clock]} ->
        if not is_nil(clocks[clock]), do: result == rem(clocks[clock], 12), else: true

      _ ->
        true
    end
  end
end
```

Fig. 9. A functional-style model for a trivial clock.

```
1  forall cmds <- commands(model) do
2    r = run_commands(model, cmds)
3    {_history, state, result} = r
4    for {clock, _} <- state.clocks, do: Clock.stop(clock)
5    (result == :ok) |> when_fail(print_report(r, cmds)) |> aggregate(command_names(cmds))
6  end
```

Fig. 10. A property to run trivial clocks.

Fig. 9 shows the same example implemented in functional style using the PropCheck library [7], which is an Elixir wrapper around the PropEr [15] tool. After introducing each mutation, the property depicted in Fig. 10 is used to test the system-under-test. Lines 1-3 show how commands are generated from the model and executed against it, line 4 stops all the clocks generated during the execution of a test, and line 5 shows how, in case of an error, the sequence of commands produced is pretty printed. Note that the property proved is identical for both types of models. Moreover, note that PropEr executes the `commands` function for generating a test case, and the `run_commands` function for executing a test case on the system-under-test (and checking postconditions) for both models.

Comparing test error reporting In the following, we compare the error reports generated for the Makina and PropCheck models.

We begin by introducing a bug into the system-under-test to compare the errors produced by each model. To do so, we proceed to modify the implementation of `time(clock)`. The faulty version first returns the current value of the clock, and after that, increments its value by 1. The introduced bug manifests when `time` is executed twice: the first call establishes the initial value of the clock, and after the second call, the postcondition fails due to a mismatch between the time computed by the model and the value returned by the implementation. Note that this test failure occurs regardless of the number of intermediate commands between the two calls to `time`. When running both models against the modified trivial clock, we obtain the following:

```
Postcondition failed.

Commands:
   var1 = New.call(%{}) -> #PID<0.352.0>
     Post state: %{clocks: %{#PID<0.352.0> => nil}}
   var2 = Time.call(%{clock: var1}) -> 8
     Post state: %{clocks: %{#PID<0.352.0> => 8}}
#! var3 = Time.call(%{clock: var1}) -> 9

Last state: %{clocks: %{#PID<0.352.0> => 8}}
```

```
Postcondition failed.

Commands:
   var1 = Clock.new() -> #PID<0.451.0>
     Post state: %{#PID<0.451.0> => nil}
   var5 = Clock.time(var1) -> 0
     Post state: %{#PID<0.451.0> => 0}
#! var7 = Clock.time(var1) -> 1

Last state: %{#PID<0.451.0> => 0}
```

We can see that the error reported by Makina (on the left) and the error reported by PropCheck (on the right) are virtually the same. Both of them have properly shrunk the minimum counterexample that consists of creating a clock and consulting its time twice. Given this result, we can consider for both models that clear error messages are reported when there is a mismatch between the model and the system-under-test (i.e., a postcondition error).

But what happens when the model itself is faulty? When there are bugs in the model, it is important that the model is blamed and not the system-under-test. If it does not do so, and thus the error is reported as a test error, this is a false positive, which is especially undesirable during testing. In the following, we evaluate how Makina and PropCheck detect and report model errors. To do this analysis, we introduce errors into the models, use them for testing, and analyse the resulting error messages.

Introducing errors into the values returned by model functions First, we introduce a bug in the preconditions, modifying the return value on line 11 in Fig. 8 to `:wrong` and adding the atom `:wrong` in line 24 in Fig. 9. For the functional-style model with this modification, we get the following error:

```
** (CaseClauseError) no case clause matching: :error
```

This, apart from not being very descriptive, does not blame the implementation of the precondition. As this precondition is consumed by a generator of commands, the reported stack trace only shows calls to the internal implementation of the generator of commands inside PropCheck. For the Makina model, we get this error instead:

```
** (Makina.Error) error in `Clock.Makina.time.pre` which returned `:error` but should return
   `boolean() | nil` where `nil` stands for `true`
```

This error does proper blaming, pointing out that the error is in the precondition of `Clock.time`. But in addition to that, it gives a reasonable explanation of why it is an error: the value returned by **`pre`** is not any of the values that the consumer of **`pre`** can use.

Now let us modify the generator of arguments and commands, respectively. In the Makina model we change the value returned by **`args`** (line 12 in Fig. 8) to the atom `:wrong`, and in **`command`** of the functional-style state machine we change the return value by the same atom (line 6). For the PropCheck model, we get the following error:

```
** (CaseClauseError) no case clause matching: :wrong
```

Again, this is not a very descriptive error, but apart from that, if we have a look at the reported stack-trace, it points to `precondition`, as it is this function that consumes the arguments generated by **`command`**. This means that to debug this error, the user needs to know how these functions are internally called by PropCheck. In contrast to this, the following error is reported for the Makina model:

```
** (Makina.Error) error in `Clock.Makina.time.args`, returned `:wrong` but type
   `generator([{:clock, generator(symbolic_expr())}])` was expected
```

Once again, this error correctly blames `Clock.Makina.time.`**`args`**, which is the function that does not return a valid value. Notice that the type that it was expecting is a generator of lists of tuples containing the atom `:clock` and symbolic expressions; this type is derived from the type annotation on line 10 that says that the arguments produced by `time` must be symbolic values.

Finally, let us modify **`next`** (line 12 in Fig. 8) and `next_state` (line 28 in Fig. 9), by adding the value `:wrong`. For the functional model, we obtain:

```
** (BadMapError) expected a map, got: :wrong
```

This error blames line 10 in Fig. 9, which attempts to access the keys of the `clocks` map, but at this point this variable contains the atom `:wrong` instead. This is because `next_state` (wrongly) updates the state and this new state is passed to **`command`**. This once again forces the user to know the internal machinery of PropCheck. For the Makina model, we got:

```
** (Makina.Error) error in updates returned by `Clock.Makina.time.next`:
    - should return a list of updates, but instead returned: `:wrong`
```

Once again, this error identifies the correct faulty function (**`next`**) and provides an informative message about the issue in the result of `Clock.time.`**`next`**.

Introducing state errors Regarding the state, one of the significant differences between PropCheck and Makina is that the latter provides a structured approach to state declaration and manipulation. Let us introduce an error on the initial state by adding the atom `:wrong` on line 4 in Fig. 8 and line 4 in Fig. 9. For the functional-style model:

```
** (BadMapError) expected a map, got: :error
```

That error points to the generator of commands in line 11 (as we have seen in previous examples). In contrast, the error reported for the Makina model suggests that the error is in the definition of the initial state:

```
** (Makina.Error) errors on initial state in model `Clock.Makina`:
    - `clocks` has value `:error` but type `%{optional(symbolic_expr()) => nil | symbolic_expr()}`
      was expected
```

Since PropCheck does not restrict the state in the functional-style models, it becomes hard to diagnose and debug errors related to incorrect state definitions. In comparison, the approach taken by Makina allows for a thorough analysis of the state, aiding in the diagnosis of errors. For instance, if we add an extra attribute `:attr` in line 13 we get this error message:

```
** (Makina.Error) error in updates returned by `Clock.Makina.time.next`:
    - updates to unknown attributes: `:attr`
```

Another example is when we mistakenly update a value declared as symbolic with a non-symbolic value. If we modify `value` on line 25 and change it to the integer `1`, we get the following error:

```
** (Makina.Error) error in updates returned by `Clock.Makina.tick.next`:
    - update for attribute `:clocks` is `%{{:var, 1} => 1}` but type
      `%{optional(symbolic_expr()) => nil | symbolic_expr()}`
      was expected
```

This error shows that the state after **`next`** contains an integer as a key, but it should contain a **`nil`** or a `symbolic_expr()`. If we introduce the same error in the functional model (modifying line 38 in Fig. 9) and execute the functional model, we get a false positive, an error in the model that manifests as a test error:

```
Postcondition failed.

Commands:
   var1 = Clock.new() -> #PID<0.722.0>    Post state: %{#PID<0.722.0> => nil}
   var3 = Clock.time(var1) -> 1           Post state: %{#PID<0.722.0> => 1}
   var4 = Clock.tick(var1) -> :ok         Post state: %{#PID<0.722.0> => 1}
#! var5 = Clock.time(var1) -> 2           Last state: %{#PID<0.722.0> => 1}
```

But it is crucial not only to catch errors in symbolic state updates, but also to ensure that symbolic expressions are evaluated consistently when tests are executed in a dynamic context. For example, let us modify again line 25 in Fig. 9 and line 38 in Fig. 8 and introduce the symbolic expression `{:call,Kernel,:elem,[{`**`true`**`},0]}` which during the execution of the test is evaluated to **`true`**. For the functional model, we get:

```
Postcondition crashed:
** (ArithmeticError) bad argument in arithmetic expression :erlang.rem(true, 12)

Commands:
   var1 = Clock.new() -> #PID<0.750.0>    Post state: %{#PID<0.750.0> => nil}
   var3 = Clock.time(var1) -> 11          Post state: %{#PID<0.750.0> => 11}
   var4 = Clock.tick(var1) -> :ok         Post state: %{#PID<0.750.0> => true}
#! var5 = Clock.time(var1) -> 0           Last state: %{#PID<0.750.0> => true}
```

This error indicates that there is an issue with the model (as indicated by the fact that it states the postcondition crashed rather than failed), so this is not a false positive. But the error blames the computation of the reminder in `time`, which is not very accurate. For the Makina model, we get the following error:

```
Postcondition crashed:
** (Makina.Error) error in dynamic state in `Clock.Makina.tick.post`, revise `next`:
    - `clocks` has value `%{#PID<0.355.0> => true}` but type
      `%{optional(pid()) => nil | integer()}` was expected

Commands:
   var1 = New.call(%{}) -> #PID<0.355.0>      Post state: %{clocks: %{#PID<0.355.0> => nil}}
   var2 = Time.call(%{clock: var1}) -> 2      Post state: %{clocks: %{#PID<0.355.0> => 2}}
   var3 = Tick.call(%{clock: var1}) -> :ok    Post state: %{clocks: %{#PID<0.355.0> => true}}
#! var4 = Tick.call(%{clock: var1}) -> :ok    Last state: %{clocks: %{#PID<0.355.0> => true}}
```

This error indicates that the issue results from an incorrect update in **`next`** and shows the expected type when evaluating the post-condition (in the dynamic context).

Introducing errors in argument generators Writing generators of arguments is another particularly error-prone aspect of model development. We will ignore the fact that command generators in functional-style state machines can generate calls to commands that do not exist, whereas this is impossible for Makina models.[17]

First, let us consider what happens when some arguments are missing in the generated call. Let us modify the generators `time` arguments in both models: in Fig. 9 we change `[clock]` by `[]` in line 14; in Fig. 8 we just remove line 12. For the resulting PropCheck model, the following error is reported:

```
** (CaseClauseError) no case clause matching: {:call, Clock, :time, []}
```

This error blames line 21, as there is no match for the given tuple in that case pattern. The problem is once again that the error does not blame the command generator, which is where the real error is. In contrast, for the Makina model, the error suggests that the problem is that **`args`** is not providing a generator for the argument `clock`:

```
** (Makina.Error) error in `Clock.Makina.time.args`, missing arguments: `:clock`
```

The same happens when additional parameters are provided; let us add an extra parameter to line 15 of the functional model with the symbolic call `{:call,Clock,:time,[clock,1]}`. This produces the following error for the PropCheck model:

```
** (CaseClauseError) no case clause matching: {:call, Clock, :time, [{:var, 1}, 1]}
```

If we add the same argument to the Makina model (by adding `arg: 1` in line 12) we get an error that says that there is an extra argument returned by **`args`**:

```
** (Makina.Error) error in `Clock.Makina.time.args`, returned extra arguments: `:arg`
```

Finally, as for states, the arguments may have symbolic values. In the Makina model, all argument clocks generated in `time` and `tick` are marked as symbolic PIDs. If we change the generator so that it produces a concrete PID (for example, by modifying line 12 and returning `clock: self()`), we get:

```
** (Makina.Error) error on the generator returned by `Clock.Makina.time.args`:
    - `clock` has value `#PID<0.297.0>` that does not match type `generator(symbolic(pid()))`
```

But there is another problem with the values returned by **`args`**: these values can be wrapped inside generators, and thus, we cannot inspect their value in the result of **`args`**, we can only inspect once the value is generated. How does Makina deal with this? Let us modify the generator of `clock` in line 12 by `oneof(Process.list())` that returns a generator of PIDs. Notice that this is wrong, as the value produced by that generator is a PID and not a symbolic PID. For the Makina model, after this modification, we get:

```
** (Makina.Error) type error in arguments in `Clock.Makina.time.next`, please revise `args`:
  - `clock` has value `#PID<0.66.0>` but type `symbolic_expr()` was expected
```

This states that the value clock produced by the generator returned by **`args`**, when used in next, is not a symbolic expression. Note that we have not compared these last two examples with PropCheck, as when such errors are introduced in the PropCheck model, it strangely can generate tests using the property, but only calls to `new` are present in the tests. This is a completely undesired behaviour, and we may classify this as a false positive, as the implementation is successfully tested using the property, even though model is buggy.

Other errors detected by makina We devote the last part of this section to analyse other types of error that cannot be directly represented in functional-state machines since they concern Makina operators or DSL constructs. In Makina models, names of commands, command arguments, and invariant names must be unique. If there are repeated occurrences of any of these, the following compile-time error is reported:

```
** (Makina.Error) commands must have different names, repeated command `new` in `Clock.Makina`
** (Makina.Error) arguments must have different names, repeated name `clock` in `Clock.Makina.time`
```

It is also forbidden that command argument names override state attribute names. The following compile-time error is reported:

```
** (Makina.Error) avoid using attribute names as command arguments:
    - `:clocks` in `Clock.Makina.time` overlap(s) with its state attributes
```

During compilation, it is also checked that commands contain definitions for all callbacks. The main problem has to do with **`call`** as it does not provide a default implementation. If this callback is missing, a compile-time warning is thrown. If we remove the `:implemented_by` option in our running example, we get these compile-time warnings:

```
warning: callback `call` undefined in `Clock.Makina.new`
warning: callback `call` undefined in `Clock.Makina.time`
warning: callback `call` undefined in `Clock.Makina.tick`
```

[17] PropCheck [7] and EQC [36] provide DSLs that also fix this issue.

But sometimes it is useful to define intermediate models that are not executable. In this case, we can use the option `abstract: true`. The effect of this option is twofold: (i) it removes these compilation warnings; and (ii) adds a runtime check that warns if an abstract model is executed. For example, if the `:implemented_by` option is removed from our running example, and we convert it into an abstract model by adding `abstract: true`, this following error message is produced:

```
** (Makina.Error) call to abstract command 'ClockModel.new'.
```

Notice that abstract models cannot be executed, but can be used to generate tests.

Makina operators can be composed; for instance, we could hide and rename commands. If we try to hide, and after that, rename the command `new` in our running example, we would get the following compile-time error message:

```
** (Makina.Error) ':where' error, missing command 'new' in model 'Clock.Makina hiding: :new'
```

Notice that if we apply these operators in reverse order, first renaming and after that hiding, we would produce a valid model.

Extended models can override both commands and state attributes, and runtime checks are introduced to ensure type compatibility between the models. For example, suppose that we define the following model that (wrongly) overrides the `clocks` attribute of our running example:

```
defmodule ExtendedClock do
  use Makina, extends: Clock.Makina
  state clocks: []
end
```

If we test the system-under-test using the model `ExtendedClock`, an error that informs that the initial state in `ExtendedClock` is not compatible with the base model `Clock.Makina` is produced:

```
** (Makina.Error) errors on initial state in model 'ExtendedClock':
    - 'clocks' has value '[]' that is not compatible with 'clocks' in 'Clock.Makina'
      that expected '%{optional(symbolic_expr()) => nil | symbolic_expr()}'
```

Type compatibility across Makina models is conjunctive, that is, given a value, it should be type compatible with the model where it has been defined and also with all the types declared in extending models.

11.2. A makina model for the AED library

The AED library [37] (`aedlib`), which is used in an undergraduate course on Algorithms and Data Structures at the Escuela Técnica Superior de Ingenieros Informáticos at the Universidad Politécnica de Madrid, implements several data structures using Java. The library is inspired by the previous work by Goodrich et al. in the `net.datastructures` library [38]. The design of the `aedlib` library is focused around teaching requirements, choosing simplicity of design and teachability rather than efficiency. For instance, the library provides several implementations of data structures which are not efficient but useful to demonstrate a particular implementation technique.

In this section, we develop a Makina model for a substantial part of the library,[18] illustrating how the use of the Makina compositional operators enables the separation of different model concerns into different submodels. Moreover, since an EQC testing model for the library also exists, the EQC and Makina models can be compared with regard to metrics such as the number of lines of model code for the different PBT models.

A summary of the Java package, class, and interface hierarchy for the tested part of the library is depicted in Fig. 11. As an example, the box labelled *lifo* (a stack) corresponds to a Java package comprising two classes, `LIFOList` and `LIFOArray`, and the `LIFO` interface which both classes implement. Moreover, the `LIFO` interface extends the `Iterable` interface.

The different elements of the Java hierarchy in the figure can be replicated as Makina models. However, additional Makina models are used to focus on particular behavioural aspects of the `aedlib` library. Fig. 12 shows the Makina models that depend on `AED.LIFOList`, the principal Makina model to test the Java class `LIFOList`. An arrow from a model to another one indicates that the second model *extends* the first one. For example, `AED.Node` extends `AED.Structure`. The figure also shows which commands are introduced by each model and whether the commands are "hidden". For example, the `AED.LIFO` model defines the commands `push`, `pop` and `top`, while hiding the commands `mcall` and `mmcall`.

In the following, the Makina models depicted in Fig. 12 will be briefly discussed. The figure shows the hierarchy of Makina models on which the `AED.LIFOList` model transitively depends. If a model depends directly on just a single model (e.g., `AED.LIFOList` depends only on `AED.LIFO`), the relationship is implemented using the extension operator. When a model depends directly on multiple other models (such as `AED.OrderedDataStructure`), the relationship is realized using the composition operator. This example highlights two useful modelling styles: (i) a model expressing a general behaviour can be refined into a more specialized behavioural model (e.g., the array behaviour in `AED.Arrays` is used to derive the `AED.OrderedArrays` model); and (ii) independent properties, such as order, equality, and emptiness, can be defined in separate models, and later combined using the composition operator into a composite new model.

[18] Omitting graph based and priority queue data structures. The EQC library model does check these data structures; we foresee no difficulties adding support for testing these data structures to the Makina library model.

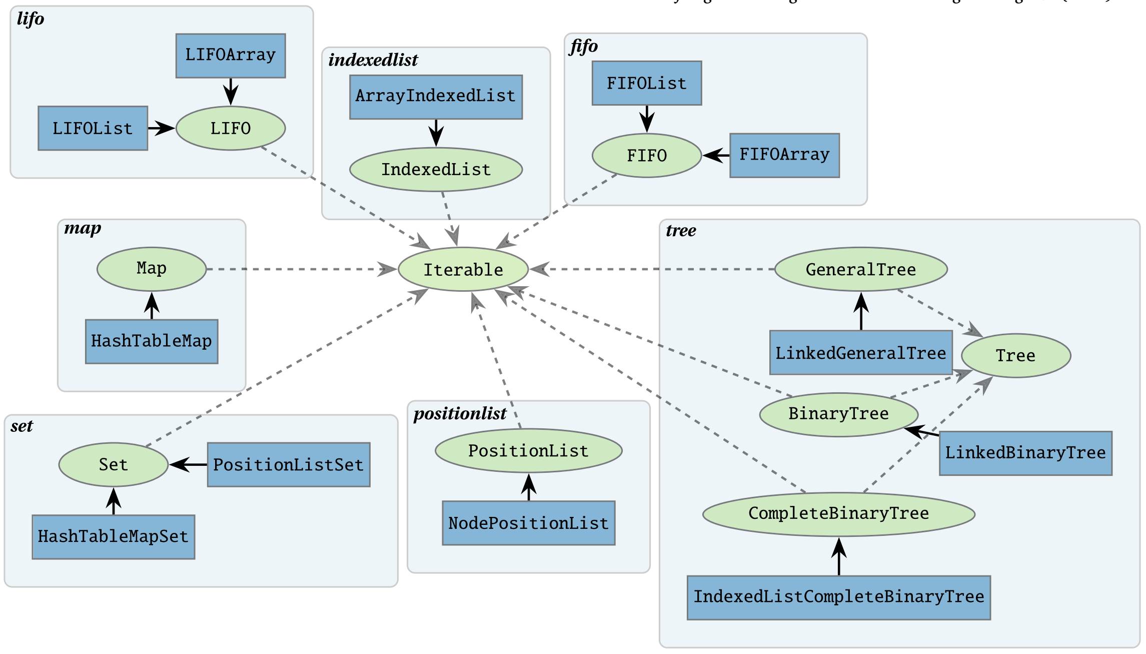


Fig. 11. The `aedlib` package/class/interface hierarchy.

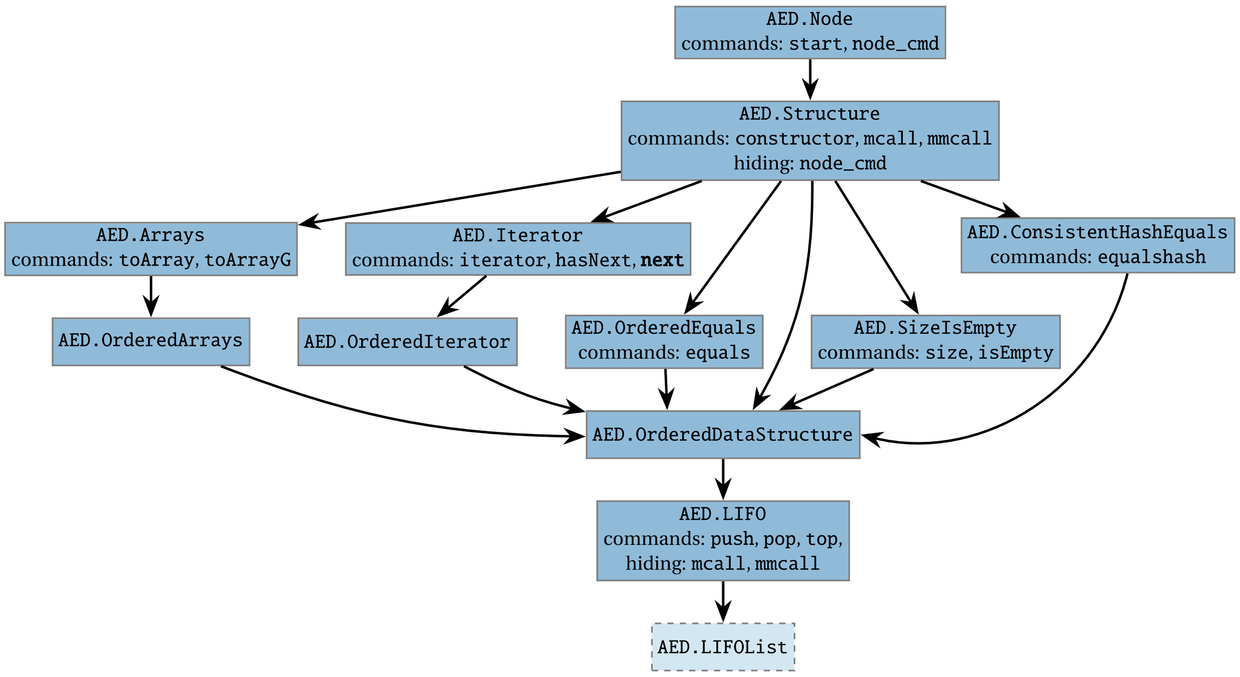


Fig. 12. Model dependencies for the `AED.LIFOList` Makina model.

We begin with the base model `AED.Node`, which does not extend any model; its model code is shown in Fig. 13. To test the Java library, a connection is established between a Java runtime, that runs the library code, and the Elixir runtime that executes the model tests using the JavaErlang library [5]. The `AED.Node` model defines an attribute `node_id`, which stores an identifier for the connection between the Elixir and Java runtimes, initially `:undefined`. Moreover, two commands are defined: `start()` and `node_cmd`. The `node_cmd` command has no behaviour, but requires the `start` command to execute first. The `start` command sets up the connection between the Java runtime and the Elixir runtime through the call to the `Java.start_node()` function. As all commands in other models will be derived from (extend) the `node_cmd` command in this way, it is ensured that the first command executed in any test is the `start` command.

```
defmodule AED.Node do
  use Makina
  state node_id: :undefined :: :undefined | symbolic(AED.Java.node_id())

  command start() :: {:ok, AED.Java.node_id()} do
    pre node_id == :undefined
    call Java.start_node([])
    next node_id: symbolic(Node.extract_node(result))
  end

  command node_cmd() do
    pre node_id != :undefined
  end

  def extract_node({:ok, node}), do: node
end
```

Fig. 13. The `AED.Node` model.

```
defmodule AED.Structure do
  use Makina, extends: AED.Node,
              where: [node_cmd: :constructor, node_cmd: :mcall, node_cmd: :mmcall],
              hiding: [:node_cmd]

  state implementingClass: :"" :: atom(),
        structures: %{} :: %{optional(symbolic(AED.Java.ref())) => any()},
        get_elements: fn _ref, _state -> raise "not␣defined" end,
        non_modifiable_references: []

  command constructor(node, class) do
    args node: node_id, class: implementingClass
    call Java.new(node, class, [])
  end

  command mcall(ref) do
    pre map_size(structures) > 0
    args ref: oneof(references(state))
    valid_args Enum.member?(references(state), ref)
  end
  ...
```

Fig. 14. The `AED.Structure` model.

Next, consider the `AED.Structure` model in Fig. 14 which extends `AED.Node`. The model defines four state attributes: `implementingClass` will, in extending models, be set to the name of the Java class being tested. The `structures` state attribute has a central role, associating a Java object reference (to a data structure) with the elements that the data structure should contain according to the Makina model. Note that a type is specified for the state attribute: Java references are mapped to arbitrary values. As such Java references are returned when the test case is executed, the `symbolic(AED.Java.ref())` annotation is required, as during test case generation, the attribute will instead contain a symbolic test variable representing the future Java reference. As explained in earlier sections, such types will cause the callbacks of the EQC model (to which the Makina model is translated) to include function type specifications. These type specifications may be checked by static type checking tools such as Dialyzer [8], and at runtime by dynamic type checking tools like Corsa [34]. The attribute `get_elements` should define a function which returns, given a reference, the list of elements in the model state,[19] and the attribute `non_modifiable_references` enumerates the Java references for which an iterator has been created.

The model provides three commands, `constructor` (a command for constructing a new Java object), `mcall` (a Java method call without side effects) and `mmcall` (a Java method call with side effects), all derived from the `node_cmd` command in the `AED.Node` model. The separation between method calls with/without side effects is made in order to correctly model the behaviour of the data structure when iterators and method calls with side effects are interleaved, i.e., the Java exception `ConcurrentModificationException` may be raised.

The constructor command `constructor(node,class)` creates a new Java instance of class `class` by calling the `Java.new` method, which requires the `node_id` of the connection between the Elixir and Java runtimes as a parameter. Since the command is derived from the `node_cmd` command, its precondition also includes the requirement that the `start` command in `AED.Node` must already have been executed, and thus, a node identifier is available. Note that the command does not modify the model state (there

[19] This function is used for e.g. testing iterators.

```
defmodule AED.LIFO do
  use Makina, extends: AED.OrderedDatastructure,
    where: [mcall: :top, mmcall: :pop, mmcall: :push],
    hiding: [:mcall, :mmcall]

  state structures: super() :: %{optional(symbolic(AED.Java.ref())) => list(integer()|:null)},
        get_elements: &stack/2

  command constructor(node, class) do
    next set_stack(result, [], state)
    post not(Java.is_exception(result))
  end

  command push(ref, value) do
    args value: oneof([integer(), :null])
    call Java.call(ref, :push, [value])
    next set_stack(ref, [value | stack(ref, state)], state)
  end
  ...
```

Fig. 15. The `AED.LIFO` model.

is no **`next`** callback). The models that extend `AED.Structure` are responsible for modifying the model state. The `mcall(ref)` command has as a precondition (**`pre`**) that a Java reference exists, and the **`args`** callback randomly chooses a reference.

The `AED.Structure` is extended by many other models, as it provides a "blueprint" for other commands, i.e., the requirement that a Java object reference exists, and choosing randomly an object reference as a command parameter.

Inspecting the interfaces in Fig. 11, all data structures must implement the methods `size()` and `isEmpty()`, which reports on the number of elements stored in the data structure. Thus, to permit model reuse for different types of data structures, the Makina model `AED.SizeIsEmpty` is provided, which defines the two commands `isEmpty(ref)` and `size(ref)`. Both commands are parametric on a Java object reference, and thus naturally extend the `AED.Structure` model and commands. Similarly, the model `AED.ConsistentHashEquals` checks that the implementation of the Java `equals(Object obj)` and `hashCode()` methods are consistent.[20]

Another opportunity for model reuse is provided by the `Iterable` interface, which is extended by all data structure interfaces, providing a mechanism to access the elements of a data structure. However, as the model checks *observable behaviour* rather than just the presence of correctly named and typed methods in Java classes, we provide two iterator models: `AED.OrderedIterator` for when elements are returned in a well-defined order (e.g., for `LIFOList`), and `AED.UnorderedIterator` for when they are not (e.g., for `HashTableMap`). Both iterator models extend the base `AED.Iterator` model.

Similarly, most interfaces also provide the methods `toArray()` and `toArray(E[] arr)` to return the elements of the data structure in an array, either in a well-specified order or not, corresponding to the two array models: `OrderedArrays` and `UnorderedArrays`.

To capture these intuitions of ordered and unordered data structures, we provide two Makina models, `AED.OrderedDatastructure` and `AED.UnorderedDatastructure`. The first model reads:

```
defmodule AED.OrderedDatastructure do
  use Makina, extends: [AED.SizeIsEmpty, AED.ConsistentHashEquals, AED.OrderedIterator,
                        AED.OrderedArrays, AED.OrderedEquals, AED.Structure]
end
```

That is, any ordered data structure must provide the functionality tested by the extended models.

The `AED.LIFO` model (corresponding to the `LIFO` Java interface) depicted in Fig. 15 extends the ordered data structure, and in addition the model provides commands for the methods unique to LIFO's (stacks): `push(E e)`, `pop()` and `top()`, and redefines the `constructor(node,class)` command introduced in `AED.Structure`.

In the `push(ref,value)` command, a value (either an integer or the `:null` atom) is pushed on top of the stack. Note that the **`args`** callback only generates the value parameter; the reference parameter `ref` is generated by the `mcall(ref)` command which is extended by the `push(ref,value)` command. The model state is updated in the **`next`** callback: the `set_stack(ref,`*list*`)` function updates the map in the `structure` attribute with a new list where `value` is concatenated to the list in the previous model state. The constructor command simply updates the model state, associating the result (a Java reference to a new object) with the empty list (the LIFO is initially empty).

Note also the type declaration for the `structures` state attribute: a map associating a Java reference with a *list containing integers or the* `:null` *reference*. Recall that, in the definition of the `AED.Structure` the type for the `structures` attribute was "weaker": a map associating a Java reference with any value (line 7 in Fig. 14). As the `AED.Structure` is extended by multiple data structure models, e.g., both `AED.LIFO` and `AED.Map`, the `AED.Structure` type specification for `structures` has to permit different types

[20] That is, equal objects must have the same hash code.

of model states, whereas in `AED.LIFO` it is known that references will always be associated with lists of integers or the atom `:null`. Makina assumes both type specifications to be valid, i.e., at runtime it is checked that the `structures` attribute conforms to *both* type specifications.

Finally, the model `LIFOList` details which Java class is tested by redefining the `implementingClass` attribute, which was originally defined in `AED.Structure`:

```
defmodule AED.LIFOList do
  use Makina, extends: AED.LIFO
  state implementingClass: :'es.upm.aedlib.lifo.LIFOList'
end
```

Metrics and style comparison with the EQC model The Makina model of the `aedlib` library is composed of 38 separate Makina models, for a total of around 1600 lines of code compared with the around 2670 lines of code for the Erlang EQC model that tests the same `aedlib` functionality.[21]

The principal difference between the Erlang EQC models and the Makina models is that the EQC models are monolithic; no easily used EQC based mechanism to subdivide a model into submodels exists. To permit a limited form of model reuse the `aedlib` model instead uses the Erlang pre-processor to inject Erlang source code into another piece of source code at compile time.

Thus, for instance, in the EQC model, the commands for checking iterators are stored in a file, which is included in models that test data structures that implement iterators. However, to permit testing a data structure which implements multiple iterators, the model state (and the iterator code) becomes more complex, whereas in the Makina model it is a simple task to extend a more generic iterator model into multiple specialized iterator models. Moreover, as in EQC the same commands are used for all iterators, it becomes much harder to analyse which iterator revealed an error.

As we have seen, the Makina model routinely redefines (or extends) commands defined in a submodel in an extending model. Well-formedness checks applied (at compile-time), when a Makina model is translated into an EQC model, ensure that such command redefinitions are safe. In EQC, commands cannot be redefined. Thus, in the EQC model for, e.g. `LIFO` the code to choose a Java reference must be replicated in the **`args`** callback for all commands (`top`, …), while in the Makina model this is managed by the command `mcall` (defined in `AED.Structure`) which is extended by, e.g. `top`. This permits the choice of a reference to be localized to one code location.

Third, the EQC model does not include type specifications for states or commands, as assertions stating that callback functions respect such type specifications would have to be added by hand for all the relevant callbacks, which is a tedious task. Moreover, the task is complicated by the fact that such type specifications must change depending on whether the callback is executed in a dynamic (execution time) context, or symbolic (test generation time) context, or both. In Makina, on the contrary, such type specifications are compiled into an EQC model where type assertions have been automatically added to the relevant locations, thus allowing dynamic type checking tools to accurately report type errors as shown in Sect. 11.1.

11.3. Benchmarks

Table 1 shows the average compilation times, the average test generation times, and the average test execution times for the functional models presented in this article. Table 2 depicts the compilation times (of just the chosen model or the entire project), the average generation times (with type checking assertions enabled and disabled), and the average execution times (again, with type checking assertions enabled and disabled) of the models introduced in this paper, together with additional examples from the repository https://gitlab.com/babel-upm/makina/examples.git. This repository houses a collection of examples that show the practical application of the Makina library. Examples included in the repository also explore different modelling styles offered by the library, e.g., separating generation and checking concerns in different models. The test generation times and test execution times represent the average duration for generating and executing a single sequence of 1,000 commands. All benchmarks[22] were run on an Intel® Core™ i5-8265U CPU at 1.60GHz, with 8 cores, 15.31GB of RAM, and running Elixir 1.18.3 and Erlang 27.3 under Linux (kernel 6.14.0).

As an example, the compilation of the Makina authentication model is, on average, 12 times slower than the functional-style model. Similarly, for the trivial clock example, the compilation time of the Makina model is 8 times slower. This is expected as Makina generates multiple modules and introduces type definitions and type assertions throughout the generated code. Among the examples, the `AED.LIFOList` model takes the longest to compile (6 seconds), which is not surprising, as it extends several other Makina models; the total time for compiling all required models is around 28 seconds.

To assess the cost of test case generation, we compare the times spent for generating test cases for the three functional models depicted in Table 1 with the time spent by equivalent Makina models in Table 2, when assertions are disabled. For the authentication models, the functional version (`Auth.Functional`) used 118.25 ms whereas the Makina model (`Clock.Makina`) used 82.39 ms. Regarding the clock models, 112.79 ms were required to generate test cases using `Clock.Functional`, versus 132.78 ms for `Clock.Makina`. For the store models, the functional version used 196.96 ms, while the Makina models used 78.63 ms

[21] Measured by the Cloc tool (https://github.com/AlDanial/cloc).

[22] These benchmarks are available at https://gitlab.com/babel-upm/makina/benchmarks.

Table 1
Compilation, test generation and test execution of functional-style state machines.

model	avg. compile time (ms)	avg. gen. time (ms)	avg. exec. time (ms)
`Auth.Functional`	29	118.25	41.98
`Clock.Functional`	44	112.79	52.34
`Store.Functional`	21	196.96	42.94

Table 2
Compilation, test generation and test execution of Makina models.

model	avg. compile time (ms)		avg. gen. time (ms)		avg. exec. time (ms)	
	chosen model	full project	no assertions	assertions	no assertions	assertions
`Auth.Makina`	373	401	82.39	330.85	109.00	981.73
`Clock.Makina`	373	409	132.78	1176.79	80.07	1262.87
`Docs.Makina`	1561	1934	140.72	334.23	84.57	1171.86
`Docs.Comp.Makina`	1271	3208	100.87	347.45	88.56	118.55
`AED.LIFOList`	6592	28857	299.29	330.54	1124.93	2533.96
`Counter`	312	322	37.50	52.34	19.76	29.84
`Counters`	348	703	125.01	1322.96	92.71	165.87
`FizzBuzz`	315	332	28.99	72.97	51.16	80.89
`OneTimePassword`	384	408	143.07	220.33	61.03	317.39
`Secret`	541	577	789.60	4179.72	185.85	2027.70
`Store.Makina`	521	532	146.61	736.43	78.63	636.26
`StoreModelExt`	401	915	138.55	1271.50	94.30	1732.65
`StoreModelComp`	701	1106	133.72	1179.50	90.69	1703.56

(`Store.Makina`), 94.30 ms (`Store.Makina`), and 90.69 ms (`StoreModelComp`). As a conclusion, benchmark data do not show a consistent overhead when Makina models are used for test case generation, if assertions are disabled.

In contrast, the time spent during test execution (and correctness checking) of comparable Makina and non Makina authentication and clock models show an execution time slowdown of 67.02 ms and 27.73 ms for the Makina models. Similarly, a comparison of the store models shows slowdowns of 35.69 ms, 51.35 ms, and 48.02 ms for the Makina models. These results indicate a consistent slowdown, yet within acceptable margins.

As a summary, while compiling Makina models is substantially slower than comparable functional-style models, generating tests from them has no significant overhead, and test execution is not substantially slowed down. However, a significant cost is associated with the type assertions optionally introduced by Makina during the compilation. This is not surprising, as checking them entails inspecting always the “deep” structure of instrumented function call parameters and results (e.g., a deep, recursive, inspection of all elements of a list), unless the type assertion fails.

12. Conclusions and further work

Although the use of property-based testing (PBT) for program testing has become quite popular, it is fair to say that the use of state machine based PBT technologies for testing stateful systems is less widespread. This is in part unavoidable. Writing properties for testing complex stateful library APIs is a far more challenging task than generalizing unit test cases that test isolated, non-stateful, functions. However, part of the lack of acceptance of state machine based PBT technologies lies, we believe, in that they are from an engineering point of view unnecessarily difficult to work with. The EQC state machine library `eqc_statem`, for instance, provides little help to the user in writing correct state machine models. It is only too easy to forget needed command parameters in a callback, or returning an incorrect model state. Such problems stem, at least in part, from the design of the `eqc_statem` library API which makes it quite hard to do effective type checking. Moreover, even when a test model is correct, it can be a daunting task for non-experts to maintain the model. This leads to test models which are sadly abandoned, when the underlying system-under-test unavoidably changes, as the cost of modifying the test model is too high.

In this paper, we have presented Makina, a library for writing EQC state machines in Elixir, to address, in a small part, the underlying problems of correct model design and model reuse. Makina enables strong typing of state machine test models, and allows test models to be constructed from other test models, in a natural and compositional way. The internal design of the library makes use of the Elixir macro facility in an interesting way, as input test models are transformed, while preserving and distributing type information to new parts of the rewritten test model. We can do this effectively as we are aware of the underlying semantics of `eqc_statem` state machines, i.e., with which parameters the `eqc_statem` library code calls these callback functions. In a sense, we have used the macro facility to write a compiler for state machines, going far beyond the standard use of macros as simple vehicles for straightforward syntactical transformations.

One of the principal motivations for our work on the new library is a currently ongoing effort to test smart contracts that operate on the Ethereum blockchain. Testing contracts on a simulated blockchain, i.e., a chain running real blockchain code but without real users, requires us to model many aspects of the blockchain, apart from the smart contract code itself. For example, to be able to predict accurately the outcome of a smart contract call, we have to consider not only the parameters of the call, and the state of the smart contract, etc., but also the "machinery" needed to put such call transaction on the blockchain, e.g., the correct use of gas, etc. Thus, a natural EQC model for smart contracts is a layered one: in one layer we model the core transaction machinery of Ethereum, and in another layer we model the actual smart contract being tested. Such layered descriptions could have been developed, for example, EQC, using various libraries. However, our desire to write models in Elixir, the fact that support for parse transforms is deprecated in Elixir, and a desire to experiment with slightly more "light-weight" layering constructs in state models led us to design the new Makina library.

As this is still ongoing work, there are several issues that need to be addressed in future work. For example, as we have seen in Section 11, the error diagnostics produced by Proper/EQC when testing a system using a Makina model report the model state as a single (map) value, rather than printing a state as a collection of attributes. Considering the expressive power of the model composition operators, one desirable extension would be to permit state attributes to be hidden and relabelled similar to the hiding and relabelling of commands, in order to permit private model state attributes. Moreover, we would like to shorten the time required for translating Makina models into functional-style EQC state machines. One possibility is to change the translation strategy, e.g., by not emitting state machines for all model composition artefacts.

A minor issue concerns the choice of the right callbacks to generate commands. Currently, Makina, as well as EQC, uses a combination of the **pre** and **args** callbacks to generate commands and command parameters. This is an efficient solution, as the **args** callback is only invoked when the command to generate has already been decided on using **pre**, but for a novice model developer the relation between the two callbacks is not very intuitive. We would like to explore alternative generative callbacks. An option is to combine the two callbacks in a single `generate` callback, as exemplified by a generator for the `put` command in our running example:

```
generate when map_size(stores) > 0 do
  [ store: oneof(Map.keys(stores)),
    value: integer() ]
end
```

The `generate` construct would permit arbitrary Elixir code inside the **when** guard, and the macro expansion rules would ensure that such a combined callback can be efficiently separated into the usual **args** and **pre** callbacks.

CRediT authorship contribution statement

Luis Eduardo Bueso de Barrio: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Lars-Åke Fredlund:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Ángel Herranz:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Clara Benac-Earle:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Julio Mariño:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been partly funded by EIE Funds of the European Union and by the Spanish MCI/AEI under grant ref. PID2019-104735RB-C44.

References

[1] K. Claessen, J. Hughes, Quickcheck: a lightweight tool for random testing of Haskell programs, in: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, ACM, New York, NY, USA, 2000, pp. 268–279, https://doi.org/10.1145/351240.351266.
[2] T. Arts, J. Hughes, J. Johansson, U.T. Wiger, Testing telecoms software with Quviq QuickCheck, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, 2006, pp. 2–10, https://doi.org/10.1145/1159789.1159792.
[3] J.M. Hughes, H. Bolinder, Testing a database for race conditions with QuickCheck, in: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 72–77, https://doi.org/10.1145/2034654.2034667.
[4] J. Hughes, B.C. Pierce, T. Arts, U. Norell, Mysteries of DropBox: property-based testing of a distributed synchronization service, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 135–145, https://doi.org/10.1109/ICST.2016.37.
[5] C.B. Earle, L.-Å. Fredlund, Functional testing of Java programs, in: M. Serrano, J. Hage (Eds.), Trends in Functional Programming - 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 9547, Springer, 2015, pp. 40–59, https://doi.org/10.1007/978-3-319-39110-63.
[6] J. Valim, https://elixir-lang.org, 2012. (Accessed 26 January 2023).

[7] K. Alfert, https://github.com/alfert/propcheck, 2015. (Accessed 6 May 2021).

[8] T. Lindahl, K. Sagonas, Practical type inference based on success typings, in: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press, New York, NY, USA, 2006, pp. 167–178, https://doi.org/10.1145/1140335.1140356.

[9] M. Wijnja, https://github.com/Qqwy/elixir-type_check, 2020. (Accessed 26 January 2023).

[10] L.E. Bueso de Barrio, L.-Å. Fredlund, A. Herranz, C.B. Earle, J. Mariño, Makina: a new QuickCheck state machine library, in: Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Erlang 2021, Association for Computing Machinery, New York, NY, USA, 2021, pp. 41–53, https://doi.org/10.1145/3471871.3472964.

[11] R. Nilsson, ScalaCheck: The Definitive Guide, Artima, 2014.

[12] D.R. MacIver, Z. Hatfield-Dodds, many others, Hypothesis: A new approach to property-based testing, J. Open Source Softw. 4 (43) (2019), https://doi.org/10.21105/joss.01891.

[13] J. Stanley, https://hedgehog.qa, 2021. (Accessed 6 May 2021).

[14] Quviq, https://github.com/Quviq/eqc_ex, 2014. (Accessed 6 May 2021).

[15] M. Papadakis, K. Sagonas, A PropEr integration of types and function specifications with property-based testing, in: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang, Erlang '11, Association for Computing Machinery, New York, NY, USA, 2011, pp. 39–50, https://doi.org/10.1145/2034654.2034663.

[16] A. Löscher, K. Sagonas, T. Voigt, Property-based testing of sensor networks, in: Sensing, Communication, and Networking, 12th Annual IEEE International Conference on, IEEE, 2015, pp. 100–108, https://doi.org/10.1109/SAHCN.2015.7338296.

[17] H. Li, S.J. Thompson, P.L. Seijas, M.A. Francisco, Automating property-based testing of evolving web services, in: W. Chin, J. Hage (Eds.), Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA, ACM, 2014, pp. 169–180, https://doi.org/10.1145/2543728.2543741.

[18] Y. Gurevich, Evolving Algebras 1993: Lipari Guide.

[19] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[20] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (3) (1987) 231–274, https://doi.org/10.1016/0167-6423(87)90035-9.

[21] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS, Comput. Netw. ISDN Syst. 14 (1) (1987) 25–59, https://doi.org/10.1016/0169-7552(87)90085-7.

[22] J.A. Bergstra, J.W. Klop, Process algebra for synchronous communication, Inf. Control 60 (1–3) (1984) 109–137, https://doi.org/10.1016/S0019-9958(84)80025-X.

[23] E.A. Lee, S.A. Seshia, Introduction to Embedded Systems: A Cyber-Physical Systems Approach, 2nd edition, The MIT Press, 2016.

[24] A. Sane, R.H. Campbell, Object-oriented state machines: subclassing, composition, delegation and genericity, in: R. Wirfs-Brock (Ed.), Proceedings of the 10th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, 1995, pp. 17–32.

[25] U. Norell, H. Svensson, T. Arts, Testing blocking operations with QuickCheck's component library, in: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang, Erlang '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 87–92, https://doi.org/10.1145/2505305.2505310.

[26] B.H. Liskov, J.M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (1994) 1811–1841.

[27] G. Gay, M. Staats, M.W. Whalen, M.P.E. Heimdahl, The risks of coverage-directed test case generation, IEEE Trans. Software Eng.

[28] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[29] T. Johnsson, Lambda lifting: transforming programs to recursive equations, in: J.-P. Jouannaud (Ed.), Functional Programming Languages and Computer Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, pp. 190–203.

[30] L.-Å. Fredlund, A framework for reasoning about Erlang code, Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden, 2001, https://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-22629.

[31] N. Nishida, A. Palacios, G. Vidal, A reversible semantics for Erlang, in: M.V. Hermenegildo, P. Lopez-Garcia (Eds.), Logic-Based Program Synthesis and Transformation, Springer International Publishing, Cham, 2017, pp. 259–274.

[32] M. Cassola, A. Talagorria, A. Pardo, M. Viera, A gradual type system for Elixir, J. Comput. Lang. 68 (2022) 101077, https://doi.org/10.1016/j.cola.2021.101077, https://www.sciencedirect.com/science/article/pii/S2590118421000551.

[33] G. Castagna, G. Duboc, J. Valim, The design principles of the elixir type system, Art Sci. Eng. Program. 8 (2) (2023), https://doi.org/10.22152/programming-journal.org/2024/8/4.

[34] L.E. Bueso de Barrio, L.-Å. Fredlund, Á. Herranz, J. Mariño, C. Benac Earle, Executable contracts for Elixir, J. Log. Algebr. Methods Program. 142 (2025) 101019, https://doi.org/10.1016/j.jlamp.2024.101019, https://www.sciencedirect.com/science/article/pii/S2352220824000737.

[35] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, Boston, Mass, 2003.

[36] Quviq, https://www.quviq.com/documentation/eqc/index.html. (Accessed 8 April 2025).

[37] G. Román, L.-Å. Fredlund, P. Nogueira, https://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/, 2020. (Accessed 20 March 2025).

[38] M.T. Goodrich, R. Tamassia, Data Structures and Algorithms in Java, 5th edition, John Wiley & Sons, Inc., USA, 2010.