Maude as a library: an efficient all-purpose programming interface

Rubén Rubio

Universidad Complutense de Madrid, Madrid, Spain
rubenrub@ucm.es

Abstract. We present a general and efficient programming interface to Maude from Python and other programming languages. All relevant Maude entities and operations are exposed in a documented object-oriented library to facilitate the integration of Maude into external programs and vice versa. This paper describes the design and implementation of the library, explains how to use it, and discusses some mature applications.

1 Introduction

Formal tools are more useful when they can cooperate and interact with the outside world through simple and well-defined interfaces. In addition to the traditional command-line interfaces, popular tools like the Z3 [27] and CVC4/5 [4,3] SMT solvers, the Storm [18] probabilistic model checker, or the Lean [28] theorem prover are offering programming interfaces to their functionality from languages like C++ and Python. Some are even conceived as libraries in the first place, like the Spot [14] platform for LTL and ω-automata. This laudable trend also reaches mainstream programming languages like C/C++, whose compiler Clang can be used as a library to inspect the abstract syntax tree of programs and control the different compilation phases.

Maude [8] is a high-performance logical and semantic framework based on rewriting logic [26]. Maude programs are collections of modules corresponding to specifications in this logic, where states are terms in an equational logic that are transformed by the nondeterministic application of rewrite rules. Rewriting logic is reflective and Maude provides a universal theory where terms, modules, and other related concepts are represented as data that can be manipulated within the language. Several tools for analyzing Maude specifications and application-specific interactive interfaces have been written using these metaprogramming features. However, interacting with external tools and visualization is not so easy within Maude, and the interpreter has occasionally been extended with custom ad hoc extensions. Examples are the Maude Formal Environment [13], which interacts with external termination provers and libraries, and several analysis and visualization tools of the ELP group at Universitat Politècnica de València [1,2].

Accepted author's manuscript of the article published in LNCS 13252
DOI: 10.1007/978-3-031-12441-9_14

Maude is being used behind the scenes by some tools like the Tamarin prover [25] for security protocol verification, the $\mathbb{K}$ semantic framework [29] (until its fifth version), and the heterogeneous tool set Hets [9], among others. All this software includes ad hoc code to run an instance of the Maude interpreter as a separate process, issue commands to its standard input stream, and parse their answers. The IMaude agent of the InterOperability Platform (IOP) [24] follows the same approach to communicate with Maude, but then provides an abstraction for other user-defined agents of this framework to interact with the language. IMaude is used by the Pathway Logic Workbench [36], Mobile Maude [7], and the graphical interface to Maude-NPA [34], among others.

We present here an intuitive programming interface for Python and other programming languages that exposes almost all functionality of the Maude interpreter and some useful extensions. Moreover, the connection in the opposite direction, from Maude to the external language, is also supported. Unlike previous tools, these language bindings are directly linked with the Maude implementation, so several new possibilities and better performance are expected from this approach. The library comes with detailed documentation and API reference, and it has already been used in some relevant projects (see Section 6).

Its implementation relies on the Simple Wrapper and Interface Generator (SWIG) [11], so bindings can be produced for any language supported by this tool. However, only Python has been extensively tested and enhanced with language-specific adaptations to provide a more natural interface. The Python module is available at the Python Package Index (PyPI) and can be installed with the command `pip install maude`. Currently, the binding for Java has also been tested to a lesser degree and the those for other languages must be compiled from source, for which instructions are available. In the following, we will focus on the Python flavor of the bindings for simplicity, although most information can be generalized to other languages.

This paper starts with a quick overview of the library in Section 2, which is further illustrated by a simple example in Section 3. Some advanced features are introduced in Section 4, and the implementation is described in Section 5. Finally, Sections 6 and 7 mention some applications and complete the discussion on related work in this introduction. More information can be found at `github.com/fadoss/maude-bindings` including documentation, examples, the API reference, and the source code of the library.

2 Overview of the library

In this section, we describe the design and overall organization of the language bindings, which coincide for all supported languages. However, we will stick to the Python instance for simplicity, as explained before.

The `maude` library exhibits all relevant Maude entities and operations as objects and methods of the target language. There are classes `Term` for terms, `Module` for modules, `Sort` for sorts, `Symbol` for symbols (or operators), `Equation` for equations, `Substitution` for substitutions, and so on. Most commands in

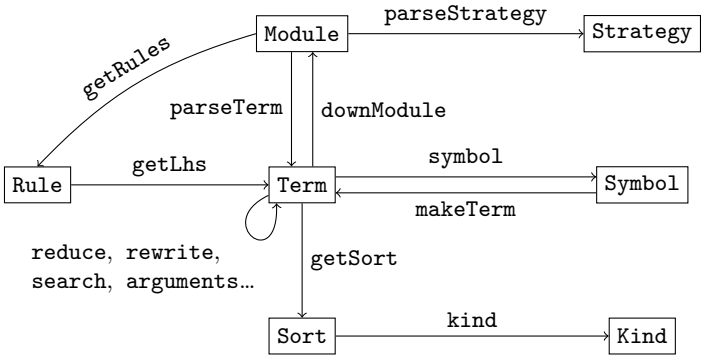

Fig. 1. Some relevant classes and methods in the library.

the Maude interpreter are gathered as methods of the `Term` class, like `reduce`, `rewrite`, `search`, `get_variants`, and `vu_narrow`. Some commands that are not applied to a singular term like `unify` are available through the `Module` class. Operations that are reserved to the metalevel in the Maude interpreter are also implemented as regular methods, like iterating over the arguments of a term with `arguments`, obtaining its least sort with `getSort`, its root symbol with `symbol`, or applying a substitution with `instantiate`, among others. Figure 1 shows a selection of the basic classes along with some methods that relate them.

A simple program that reduces the term `2 * 3` with the `maude` Python package and prints its result `6` to the terminal would look as follows

```
import maude
maude.init()
m = maude.getModule('NAT')
t = m.parseTerm('2 * 3')
t.reduce()
print(t)
```

The first two instructions load the `maude` package and initialize it with the `init` function. This must be called before anything else in the library since it sets up some required resources and loads the Maude prelude. Everything in Maude takes place within modules, so a `Module` object is needed to begin with, and it can be obtained with the `getModule` or `getCurrentModule` functions. Typically, we will then parse a term with the `parseTerm` method and apply some operations to it. The `Module` class also includes several methods for inspecting its contents.

While the library offers enough resources to manipulate terms without resorting to the metalevel, moving through different levels of reflection is natively supported with the `upTerm` and `downTerm` methods of `Module`. For expressions in the Maude strategy language, these methods are called `upStrategy` and `downStrategy`. Moreover, a `Module` object can be obtained from its metarepresentation using the `downModule` function, while the converse operation can be achieved by simply reducing an `upModule` term in the `META-LEVEL` module.

In the next section, we illustrate the possibilities of the library through an example, giving further details on how to use it. Other advanced features are described in Section 4, and more information is available on the home page of the language bindings.

3 How to use the library, illustrated by an example

In this section, a toy interactive rewriter is implemented using the `maude` library, as an excuse to illustrate its usage and possibilities. Most of this example can be programmed directly in Maude using reflection, probably in a more verbose and complex manner, but the same procedures can be used when actual interaction with the outside world is pursued.

Our interactive prototype will repeatedly read commands from the terminal and reply to them. Implementing this kind of interface in Python is easy thanks to the standard `cmd` module. We only need to subclass the `cmd.Cmd` class and provide a method `do_cmdname` to handle the command `cmdname`. Its full source code is available in the `inter.py` file of the bindings repository. As already explained, we should start by importing the library with `import maude` and initializing it with `maude.init()`. The `InteractiveRewriter` class holding the implementation of all commands in the interpreter can then be defined.[1]

```
import cmd
import maude

class InteractiveRewriter(cmd.Cmd):
  # A method will be added here for each command

if __name__ == '__main__':  # entry point
  maude.init()
  InteractiveRewriter().cmdloop()
```

For the moment, only two attributes are maintained, the current module and the term being rewritten, as specified in the class constructor.

```
def __init__(self):
  super().__init__()  # base class constructor
  self.module = None  # current module
  self.term = None    # current term
```

In order to bring modules to our scope, we need a `load` command to read them from Maude source files. Thus, we implement a method `do_load` that essentially delegates on the `load` function of the library.

```
def do_load(self, path):
  maude.load(path)
```

[1] The official documentation of the `cmd` module and other Python features that may appear is available at `docs.python.org`.

```
  self.module = maude.getCurrentModule()
  print('The current module is', self.module)
```

In addition, we set the current module using the `getCurrentModule` function, which gives the `Module` object for the last module that has been entered or explicitly selected in the file. Its name is printed in the screen by printing the object itself. However, we may want to select another module, for what we also provide commands to list the available modules and to select one of them.

```
def do_list(self, _):
  for module in maude.getModules():
    print(module)

def do_select(self, name):
  self.module = maude.getModule(name)
```

For example, assume we have a file `foo.maude` with the following module.

```
mod FOO-MODULE is
  sorts Foo Bar .
  subsort Bar < Foo .

  ops a b c :          -> Bar [ctor] .
  op  f     : Foo Foo -> Foo [ctor] .
  op  g     : Foo     -> Foo [ctor] .

  vars X Y : Foo .

  rl [swap] : f(X, Y) => f(Y, X) .
  rl [next] : a => b .
endm
```

After running the `inter.py` script with Python, the following command prompt will appear, where we can input `foo.maude` using the `load` command.

```
     *** Interactive rewriter for Maude ***

IRew> load foo
The current module is FOO-MODULE
```

Term manipulation. At this point, we need to choose a term to start rewriting.

```
def do_start(self, text):
  self.term = self.module.parseTerm(text)
```

Issuing the command `start` t makes t the current term in this session. We are not taking care about errors, but `self.term` would be `None` and error messages would have been printed if `text` could not be parsed as a term. For printing the

syntax tree of this term, we can prepare a command `tree` by writing a method `do_tree` as before, which may simply call the following recursive function:

```
def print_tree(term, indent=''):
  print(f'{indent}{term.symbol()} : {term.getSort()}')

  for argument in term.arguments():
    print_tree(argument, indent + '  ')
```

The `print_tree` function starts by printing the top symbol of `term` and its sort with the appropriate indentation, and then proceeds recursively on the arguments via the `arguments` method. Notice that strings prefixed by `f` in Python are formatted by replacing the expressions between curly brackets with their values. For example, we can show the syntax tree of `f(g(a), b)` in `FOO-MODULE` by selecting this term with `start` and calling the `tree` command.

```
IRew> start f(g(a), b)
IRew> tree
f : Foo
  g : Foo
    a : Bar
  b : Bar
```

Standard commands. One of the most useful commands in Maude is `reduce`.

```
def do_reduce(self, _):
  nrew = self.term.reduce()
  print(f'Reduced to {self.term} in {nrew} rewrites.')
```

Methods like `reduce` and `rewrite` modify the term to which they are applied and return the number of rewrites instead. Since the original term is overwritten, if desired, it can be copied before with its `copy` method. Another command with a straightforward implementation is the strategy-rewriting command `srewrite`:

```
def do_srewrite(self, text):
  strategy = self.module.parseStrategy(text)
  for result, nrew in self.term.srewrite(strategy):
    print(f'{result} in {nrew} rewrites')
```

Methods like `srewrite`, `search`, and `vu_narrow` that may produce multiple solutions return an iterator and do not alter the original term. As an example, we apply the strategy `swap ; next` to the current term with this command:

```
IRew> srewrite swap ; next
f(b, g(b)) in 2 rewrites
```

Applying rules. For our interactive rewriter to honor its name, we should provide a command `step` to execute a single rewrite on the current term.

```
def do_step(self, label):
  results = []  # results of the rewriting step

  for k, (result, subs, ctx, rl) in enumerate(
           self.term.apply(label if label else None)):
    where = self.print_context(ctx, rl.getLhs())
    results.append(result)

    print(f'({k}) {result} by applying {rl} '
          f'on {where} with {subs}')

  self.select_one(results)
```

The `apply` method of `Term` calculates all possible rewrites with any rule labeled with the given string (or any rule at all if `None` is given instead). It provides an iterator over the rewritten terms (`result`), the matching substitutions (`subs`) and contexts (`ctx`), and the applied rules themselves (`rl`). Contexts designate a single position in a term, and we see them here as functions that fill that position with the given term. In other words, `ctx(subs.instantiate(rl.getLhs())` is the original term, and `ctx(subs.instantiate(rl.getRhs())` is `result`. In this case, we hide in the `print_context` method how the context is processed since we will come back to this soon. Every result is accumulated in a list that is later passed to another unspecified method `select_one` that lets the user choose the next term.

```
IRew> start f(f(b, c), a)
IRew> step swap
(0) f(a, f(b, c))
    by applying rl f(X, Y) => f(Y, X) [label swap] .
    on top with X=f(b, c), Y=a
(1) f(f(c, b), a)
    by applying rl f(X, Y) => f(Y, X) [label swap] .
    on f(@, a) with X=b, Y=c

Select one of the options (0-1):
```

Matching and substitutions. In addition to the rules in the module, the interactive rewriter may be interested in experimenting with new rules, for what we add a command `inline` to apply inline rules.

```
IRew> start f(g(a), b)
IRew> inline g(X) => c
(0) f(c, b) in f(@, b) with X=a

There is a single option, done.
```

This command can be implemented by manually matching the left-hand side of `=>` and replacing it with the right-hand side instantiated with the matching

substitution. The `match` method of `Term` is the appropriate resource for this. It takes a pattern as an argument.

```
def do_inline(self, text):
  lhs, rhs = text.split('=>', maxsplit=1)
  lhs = self.module.parseTerm(lhs)
  rhs = self.module.parseTerm(rhs)

  results = []  # results of inline rewriting

  for k, (subs, ctx) in enumerate(self.term.match(lhs,
                          maxDepth=maude.UNBOUNDED)):
    result = ctx(subs.instantiate(rhs))
    where = self.print_context(ctx, lhs)

    print(f'({k}) {result} in {where} with {subs}')
    results.append(result)

  self.select_one(results)
```

The first block in the method separates the left- and right-hand sides of the inline rule and parses them in the current module. Then, `lhs` is matched against the current term `self.term`, obtaining the matching substitution `subs` and context `ctx`. By default, matching is limited to the top symbol without extension, but `minDepth` and `maxDepth` can be set to fix maximum and minimum depths. The auxiliary method `print_context` can be defined as follows.

```
def print_context(self, ctx, lhs):
  var_name = f'<<PH>>:{lhs.getSort()}'
  var_term = self.module.parseTerm(var_name)

  ctx = ctx(var_term)

  return 'top' if ctx.isVariable() \
    else str(ctx).replace(var_name, '@')
```

The context is instantiated with a placeholder variable `<<PH>>`. If the result is a variable, matching has happened on top. Otherwise, we replace the placeholder by the `@` sign for aesthetic reasons.

Building terms and modules. Once convinced with the new rule, we may want to add it to the current module with a new command `add`. Since modules are immutable in Maude, the library does not provide any direct resource to modify them, but we can always draw on the metalevel. This requires a more complex processing that we will carefully explain. Given the command `add` l `=>` r, suppose both sides of the rule have been parsed into the variables `lhs` and `rhs`, like in the `inline` command. To modify the current module at the metalevel,

we should obtain its metarepresentation by evaluating the `upModule` operator of the `META-LEVEL` module at the beginning of our `do_add` method.

```
ml = maude.getModule('META-LEVEL')

if self.metamodule is None:
  self.metamodule =
    ml.parseTerm(f"upModule('{self.module}, false)")
  self.metamodule.reduce()
```

The module term is stored in the `metamodule` attribute of the interpreter for the next time. Remember that the metarepresentation of a module in Maude is an operator with a set-like argument for each type of declaration or statement in it. Hence, we will construct the metarepresentation of the new rule and insert it in the slot of rule statements. The first ingredient is the operator

```
op rl_=>_[_]. : Term Term AttrSet -> Rule [ctor] .
```

for unconditional rules in the universal theory of `META-LEVEL`. The `findSymbol` method of `Module` allows finding operators in the module by their names and signatures, given as a sequence of domain kinds and a range kind. These kinds should be obtained first with the `findSort` and `kind` methods.

```
term_kind = ml.findSort('Term').kind()
rule_kind = ml.findSort('Rule').kind()
attr_kind = ml.findSort('Attr').kind()

rl_symb = ml.findSymbol('rl_=>_[_].', (term_kind,
               term_kind, attr_kind), rule_kind)
```

Now, we only have to fill the gaps with the metarepresentations of `lhs` and `rhs`, and with the constant `none` for the attribute part of the statement. We parse this latter constant with the `parseTerm` as usual, but providing the additional argument `attr_kind` to restrict parsing to this kind and avoid ambiguities.

```
none_attr = ml.parseTerm('none', attr_kind)
```

Finally, `Symbol`'s `makeTerm` constructs a term with a given sequence of arguments.

```
rl_term = rl_symb.makeTerm((ml.upTerm(lhs),
                            ml.upTerm(rhs), none_attr))
```

Syntactic sugar is provided for invoking the `makeTerm` method when a `Symbol` object is applied as a function, so the previous is equivalent to

```
rl_term =
  rl_symb(ml.upTerm(lhs), ml.upTerm(rhs), none_attr)
```

Now, `rl_term` must be inserted into the seventh argument of the metamodule, which holds the set of rules in system and strategy modules. For simplicity, we assume that the module is not a functional one. In order to add the rule to this

set, we must build a new term with the union operator `__` of `RuleSet`. The list of arguments of the metamodule is obtained into the `mm_args` variable.

```
rls_symb = ml.findSymbol('__', (rule_kind, rule_kind),
                         rule_kind)

mm_args = list(self.metamodule.arguments())
mm_args[7] = rls_symb(mm_args[7], rl_term)
```

Finally, the module is reassembled with the `makeTerm` method.

```
self.metamodule = self.metamodule.symbol()
                      .makeTerm(mm_args)
```

This new metamodule is converted to a `Module` object with the `downModule` function, then assigned to the `module` attribute of the rewriter.

```
self.module = maude.downModule(self.metamodule)
```

Term objects in the library belong to a fixed module and they cannot operate with entities from other modules, even if related by inclusion. Hence, if a term was already set, we must reparse it in the new module.

```
if self.term:
  self.term = self.module.parseTerm(str(self.term))
```

We can check that the new command works by executing the interpreter.

```
IRew> start a
IRew> add a => c
The rule has been inserted.
IRew> step
(0) b by applying rl a => b [label next] .
      on top with empty
(1) c by applying rl a => c . on top with empty

Select one of the options (0-1): 1
```

Interoperability. To conclude and connect with the interoperability goals of the library, we will implement a command `trs` that exports the rules in the module into the standard TRS format, used by multiple verification tools.

```
IRew> load foo
IRew> trs
(VAR X:Foo Y:Foo)
(RULES
  f(X:Foo, Y:Foo) -> f(Y:Foo, X:Foo)
  a -> b
)
```

Since the format includes a `VAR` entry specifying the set of variables in the rules, we must calculate this set with the following straightforward recursive function.

```
def find_vars(term, varset):
  if term.isVariable():
    varset.add(term)
  else:
    for argument in term.arguments():
      find_vars(argument, varset)
```

This `find_vars` function explores a term recursively accumulating its variables into the set `varset` of terms. Terms and most objects in the library can be safely used in dictionaries, sets, and other data structures since they support equality comparison and hashing. The implementation of the `trs` command simply iterates over the rules printing them. Instead of the default conversion of terms into strings, we use the `prettyPrint` method that permits finer control on the printing format. In particular, a zero argument causes terms to be printed in prefix form as required by the TRS format. Variables are also printed with an explicit type annotation.

```
def do_trs(self, _):
  varset = set()  # variables in the rules

  for rl in self.module.getRules():
    find_vars(rl.getLhs(), varset)
    find_vars(rl.getRhs(), varset)

  pv = lambda v: f'{v.getVarName()}:{v.getSort()}'
  print('(VARS', ' '.join(map(pv, varset)), ')')
  print('(RULES')

  for rl in self.module.getRules():
    lhs, rhs = rl.getLhs(), rl.getRhs()
    print(f'\t{lhs.prettyPrint(0)} -> '
          f'{rhs.prettyPrint(0)}')

  print(')')
```

In the general case, we should also ensure that identifiers respect the grammar of the TRS format and consider equations and structural axioms. The complete version of this example includes two more commands `termination` and `confluence` that automatically check these properties on the rules using the AProVE [17] and CSI [37] tools, with the generated TRS specification as input.

4 Advanced features

This section introduces two features of the library with useful applications and no direct correspondence in the Maude interpreter.

4.1 Rewrite graphs and model checking

Exploring the graph of all reachable states and transitions from a given initial term is useful for debugging, visualizing, and model checking Maude specifications. We can recursively build this graph in the library using the `apply` method or in Maude itself using the descent functions `metaSearch` or `metaXapply`, but this does not work for strategy-controlled models and such a common operation deserves to be a builtin feature. The language bindings offer two classes `RewriteGraph` and `StrategyRewriteGraph` to explore the rewrite graph of standard and strategy-controlled models, respectively. States are indexed by natural numbers starting from zero, the state's term can be obtained with `getStateTerm`, its successors can be enumerated with `getNextState`, and other methods can be used to obtain the rule applied in each transition. This makes it easy to program a search or any other algorithm in Python that directly operates with the graph produced by Maude.

Moreover, a high-level interface to the Maude LTL model checker [16] and its extension for strategy-controlled systems [30] is provided through these graphs. This is more convenient than reducing, as usual, the `modelCheck` operator of the `MODEL-CHECKER` module.[2] The `modelCheck` method of both graphs receives a term of sort `Formula` and returns a record indicating whether the formula holds and a counterexample that refutes if it does not. Counterexamples are described by a cycle and a path to it from the initial state, both given as lists of indices in the rewrite graph. One of the advantages of this approach is that the same graph can be used to model check multiple properties, hence saving the work required for the generation of the model in successive executions. Moreover, we can further process the graph or the counterexample when model checking has finished.

4.2 Custom special operators

Having overly shown that the `maude` module lets Python programmers evaluate Maude code in their programs, the interaction in the opposite direction, calling Python code from Maude, has not been explored yet.

User-defined and many predefined functions in Maude are specified with equations, but the prelude also includes some *special* operators whose behavior is internally defined in the C++ code of the interpreter. Most operations on the builtin types `Nat`, `Float`, `Qid`, and `String`, some polymorphic operators like equality `==`, and most descent functions in the `META-LEVEL` module are examples of special operators. Moreover, the Maude implementation has occasionally been extended ad hoc with new special operators, like in the Maude Formal Environment [13].

[2] Even though the strategy language is part of the official releases of Maude [12], the strategy-aware model checker [30] is not yet, but we have included it in the Maude build used for this library.

The language bindings allow declaring custom special operators whose behavior against equational reduction and/or rule rewriting is defined in the target language. In the Maude side, the operator should be declared first with the `special` attribute and its `id-hook SpecialHubSymbol` option. For instance, the gamma function that extends the factorial to real (and complex) numbers can be declared as the following `gamma` operator within a module.

```
op gamma : Float -> Float [special (
  id-hook SpecialHubSymbol
)] .
```

On the Python side, we have to define and register the callback that is invoked when a term with `gamma` on top is reduced or rewritten. This is done by subclassing the `maude.Hook` class and implementing its `run` method, and then calling the functions `connectEqHook` and/or `connectRlHook` to register an object of the class as the handler for the special operator.

```
class GammaHook(maude.Hook):
  def run(self, term, data):
    module = term.symbol().getModule()
    argument, = term.arguments()

    value = math.gamma(float(argument))
    return module.parseTerm(str(value))
```

The `run` method receives the `term` that it should return reduced or rewritten. The implementation of `gamma` is directly provided by the `math` module of the Python standard library, so in this case we only need to convert the argument and result from a Maude term to a Python floating-point value and the other way around. Finally, we install the hook for equational reduction with the `connectEqHook` function.

```
hook = GammaHook()
maude.connectEqHook('gamma', hook)
```

After that, when we explicitly or implicitly reduce terms containing `gamma` in the library, `hook`'s `run` would be executed and we would obtain the desired number. For instance, if we program and run a REPL that parses and reduces every line from standard input, we can obtain the following:

```
Gamma> 1.2 + gamma(6.5)
2.8908527781504438e+2
```

In the signature of the `run` method, there is another argument `data` giving access to the `op-hook` and `term-hook` attributes of the special operator. Suppose we want to implement a custom predicate that tells whether a number is prime.

```
op isPrime : Nat -> Bool [special (
  term-hook trueTerm (true)
  term-hook falseTerm (false)
)] .
```

Using the above term hooks for the Boolean constants, we can define its `run` method by the expression

```
data.getTerm('trueTerm' if test_prime(argument)
             else 'falseTerm')
```

for some `test_prime` Python function. While the same can be achieved by parsing the constants with `parseTerm`, the advantage of hooks is that keep working even if truth values are renamed, for example to `tt` and `ff`, in a module importation within Maude. Further details are explained in the documentation.

5 Implementation

The language bindings for Maude are implemented on top of the official implementation of Maude using some additional C++ code and the Simple Wrapper and Interface Generator (SWIG) [11], as illustrated in Figure 2. The desired programming interface is specified by selecting the classes, functions, and methods of the Maude implementation and the additional helper code that want to be exposed in the target language. Several languages like Python, Java, Lua, C#, Scheme, PHP, and JavaScript are supported, but only Python has been extensively tested and used in our case. From this specification, SWIG generates glue code in the selected language and in C, and this latter is then compiled into a binary module for the target language interpreter. This module is linked to the Maude implementation, which we have compiled as a shared library by adapting the build process. Indeed, we already did it to integrate Maude as a plugin for the language-independent model checker LTSmin [31]. Notice that Maude does not provide an official stable interface and the bindings are using its internal classes, so the implementation should be adapted on every new release of Maude. Moreover, instead of using the official Maude implementation as is, the language bindings are linked with our extension including a model checker for systems controlled by strategies [30], which does not alter any other aspect of the Maude implementation.

A large part of the classes and methods of the interface are direct wrappers to the homonym classes and methods of the Maude implementation, but some are implemented on purpose to facilitate the interaction. For example, terms are represented in Maude sometimes as trees and sometimes as nodes in a directed acyclic graph, but this particularity is hidden to the library user in the uniform `Term` class. This type is backed by an auxiliary C++ class `EasyTerm` that chooses the appropriate representation and manage the conversion between them. Custom special operators in Section 4.2 are supported by a `SpecialHubSymbol` subclass of the `Symbol` type of the Maude implementation written on purpose to allow registering C functions as callbacks for the equational reduction and rule rewrite handling methods of the symbol. The connection with the target language is based on the *directors* feature of SWIG and the `maude.Hook` class, whose `run` method implemented in the target language can be called from the registered callbacks of the special operator.

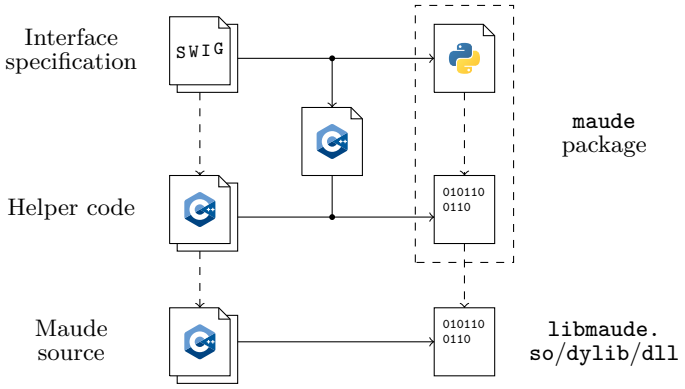

Fig. 2. Implementation structure.

When the Python interpreter executes the `import maude` statement, it loads the Python script generated by SWIG with the definition of all the classes and functions of the interface. This Python code loads the binary module that has been built from the SWIG-produced C code and the helper classes in the middle part of Figure 2. This module is linked with the dynamic library `libmaude.so` (`.dylib` in macOS or `.dll` in Windows) that contains the Maude implementation. Every object of the library in the target language holds a pointer to an object living in the Maude implementation, whose methods are invoked when the equivalent methods of the library are called. However, arguments may need to be translated in the process, for example, from a Python list to a C++ vector. This is done by the glue code generated by the interface generator.

5.1 Performance considerations

Since the language bindings replace text-based interprocess communication by direct procedure calls and despite the cost of the translations mentioned in the last paragraph, this approach is expectedly much more efficient than the classical interaction through the interpreter, especially when the results are frequently reused. We have executed some small experiments to compare the performance of reduction using (1) the `maude` Python library, (2) an I/O interaction that inputs `reduce` commands on a running Maude interpreter process and parse their results, and (3) a socket-based approach that communicates with a Maude-implemented TCP server that replies with the reduced forms of the terms it receives line by line. Reducing the constant `0` in the predefined module `CONVERSION` takes respectively (1) 3.21 μs, (2) 11.27 μs, and (3) 48.31 μs, so the best results are obtained with the `maude` Python library. Moreover, the last two options have been implemented in the simplest way possible and assuming unrealistic constraints, so production-ready implementations would likely be more costly.

Performance improvements are more noticeable when reusing the output of previous operations. For example, consider a toy Maude function `fibonacci`

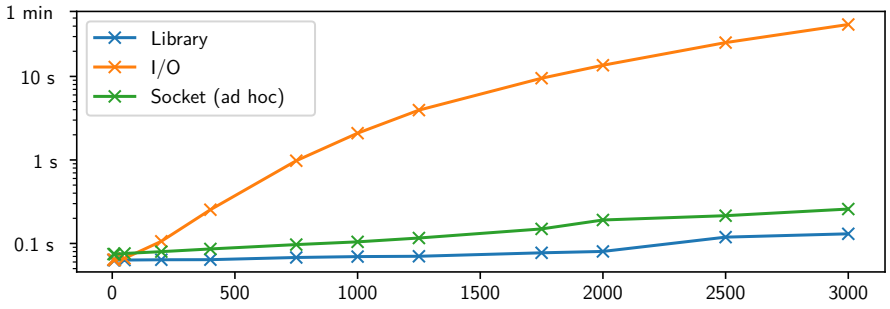

Fig. 3. Time spent in the iterative reduction of `fibonnaci` by number of iterations.

that expands a given list of integers by appending the sum of two leftmost numbers to the left. Repeatedly calling this function on the result of the previous call takes the amount of time depicted in Figure 3 (in logarithmic scale) for an increasing number of iterations. In this experiment, the socket alternative has been improved to store and reuse the result of the previous call, which is already done by the bindings out of the box. While the language bindings and the socket approach show almost a constant execution time per iteration, the I/O alternative requires Maude to parse the list of integers again and again with a much higher cost. All these benchmarks are available at the bindings repository.

6 Some applications

Since the first version of the library was released, almost two years ago, it has been applied from small quick scripts to more relevant projects. Examples of the latter are the integration of Maude into a robotic environment and a unified interface to several external verification tools.

6.1 Integration of Maude into the Robot Operating System

The Robot Operating System (ROS) [10] is a collaborative robotic framework organized as a collection of nodes that deal with the different robotic tasks and communicate with each other by message passing. One of its most prominent components is the navigation module. The officially supported languages for programming ROS nodes are C++ and Python, but in a recent work [23] Maude has been used for programming an alternative path-planning node and experimenting with the inclusion of declarative languages in this context. The `maude` Python library provides the required connection between the communication infrastructure of ROS and the actual path-planning algorithm. Even though random access to the map is enabled by a custom special operator (see Section 4.2), the efficiency of the Maude-based planner is not comparable to the

existing optimized C++ implementation, but the integration has been used for the formal verification of the latter. The more abstract Maude implementation of the navigation algorithm has been formally verified via model checking and SMT solving, and the correspondence with the official C++ planner has been established by differential testing with a huge collection of maps and paths.

In the process, the Maude library has been used for automating the evaluation of test cases, temporal properties, and verification conditions. For this latter case, we have extended the builtin SMT support in Maude with unsupported theories like arrays and uninterpreted functions. This extension and the other scripts using this library are available at [22].

6.2 The unified Maude model checker

The unified Maude model checking tool umaudemc [31] provides a uniform interface to the Maude LTL model checker [16] and several external model checkers for LTL, CTL, CTL*, and μ-calculus on standard and strategy-controlled Maude specifications. This interface reads the input data of the model-checking problem, builds the corresponding Kripke structure, calls the appropriate backend, and shows the results to the user. Among the supported backends, there are LTSmin, NuSMV [6], pyModelChecking [5], Spot [14], Spin [19], and a builtin μ-calculus implementation written in Python. The `maude` library and the rewrite graphs discussed in Section 4.1 are used to generate the models, evaluate the atomic propositions, parse the temporal formula, and so on. More recently, we have extended umaudemc for specifying probabilities on top of Maude specifications, and checking properties and calculating quantitative values by probabilistic model checking using PRISM [20] and Storm [18] or by statistical model checking through simulation or the MultiVeSta tool [35]. By using external tools, we can efficiently support more logics and techniques while reducing the maintenance effort.

Moreover, umaudemc provides graphical and web-based interfaces for model checking, allows postprocessing the counterexamples, and generates visual representations of the rewrite graphs in different formats. This tool can also be used as a library for application-specific model-checking interfaces [32,33].

7 Related work

As discussed in the introduction, several tools in the verification community maintain programming interfaces in addition to the traditional command-line ones, so that they can be used from other tools. Most applications interacting with Maude use ad hoc text-based communication with the interpreter, and the implementation of Maude has occasionally been extended to interact with external tools. The IMaude component of the IOP framework [24] is the closest precedent to this work in this context, since it provides a reusable and application-agnostic interface between Maude and external programs. However, our language bindings replace the textual communication with the interpreter

with a more efficient binary connection with its implementation, extend the available functionality, simplify the installation process, and can be used from potentially more programming languages.

On the other hand, Maude itself is being extended for a richer connection to the outside world. The notion of external objects used for accessing Internet sockets since Maude 2.0 has been applied to read and write files and standard streams in 3.0, to external processes in 3.1, and to time and filesystem operations in 3.2. External tools have also been integrated into Maude 2.7.1 with limited support for SMT solving via the CVC4 [4] and Yices2 [15] tools.

8 Conclusions

We have introduced a general-purpose efficient programming interface to Maude from Python and other programming languages. Almost all functionality of the Maude interpreter is available through these language bindings along with some useful additions. Moreover, the connection in the opposite direction, calling external code from Maude, is also available via custom special operators. This work facilitates the interoperability between Maude and other tools, and tackles the claim for using Maude from external programs.

As future work, the library can be improved and extended in several directions, like adding native support for multiple interpreter sessions with separate databases through the infrastructure of metainterpreters, allowing the construction and manipulation of modules at the object level, or distributing compiled versions of the bindings for other languages. Moreover, there is currently no clear and explicit C/C++ interface, which can be very useful for applications where performance is a critical matter. Regarding applications, there are many possibilities for the library as we have suggested along the paper, from the elaboration of interfaces for specific frameworks to the development of more general tools.

Acknowledgments. I would like to thank Enrique Martin-Martin, Manuel Montenegro, Adrián Riesco, Juan Rodríguez-Hortalá, and Óscar Martín for the suggestions that brought this library into existence and their feedback to improve it. The first version of the bindings and extensions like custom operators were originally written for [23]. I also thank Narciso Martí-Oliet and Alberto Verdejo for their comments on this manuscript. This work was partially supported by the Spanish Ministry of Science and Innovation through projects TRACES (TIN2015-67522-C3-3-R) and ProCode (PID2019-108528RB-C22), and by the Spanish Ministry of Universities through the grant FPU17/02319.

References

1. Alpuente, M., Ballis, D., Sapiña, J.: Efficient safety enforcement for Maude programs via program specialization in the ÁTAME system. Math. Comput. Sci. **14**(3), 591–606 (2020). https://doi.org/10.1007/s11786-020-00455-3

2. Alpuente, M., Escobar, S., Sapiña, J., Ballis, D.: Symbolic analysis of maude theories with narval. Theory Pract. Log. Program. **19**(5-6), 874–890 (2019). https://doi.org/10.1017/S1471068419000243
3. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
4. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. LNCS, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
5. Casagrande, A.: pyModelChecking: A simple Python model checking package (2020), https://pypi.org/project/pyModelChecking
6. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. LNCS, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., Riesco, A., Verdejo, A.: Mobile Maude. In: Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.) All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350, pp. 485–522. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1_16
8. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude Manual v3.2.1 (2022), https://maude.lcc.uma.es/maude-manual
9. Codescu, M., Mossakowski, T., Riesco, A., Maeder, C.: Integrating Maude into Hets. In: Johnson, M., Pavlovic, D. (eds.) Algebraic Methodology and Software Technology - 13th International Conference, AMAST 2010, Lac-Beauport, QC, Canada, June 23-25, 2010. Revised Selected Papers. LNCS, vol. 6486, pp. 60–75. Springer (2010). https://doi.org/10.1007/978-3-642-17796-5_4
10. developers, T.R.: Robot operating system (2020), https://www.ros.org/
11. developers, T.S.: Simplified Wrapper and Interface Generator. http://www.swig.org/ (2020), http://www.swig.org/
12. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebraic Methods Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497
13. Durán, F., Rocha, C., Álvarez, J.M.: Tool interoperability in the Maude Formal Environment. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings. LNCS, vol. 6859, pp. 400–406. Springer (2011). https://doi.org/10.1007/978-3-642-22944-2_30

14. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. LNCS, vol. 9938, pp. 122–129 (2016). https://doi.org/10.1007/978-3-319-46520-3_8
15. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. LNCS, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
16. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker and its implementation. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings. LNCS, vol. 2648, pp. 230–234. Springer (2003). https://doi.org/10.1007/3-540-44829-2_16
17. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with AProVE. J. Autom. Reason. **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y
18. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **23**(4), 1–22 (2021). https://doi.org/10.1007/s10009-021-00633-z
19. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2011)
20. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_47
21. Martin-Martin, E., Montenegro, M., Riesco, A., Rodríguez-Hortalá, J., Rubio, R.: Verification of the ROS NavFn planner using executable specification languages. J. Log. Algebraic Methods Program. **132** (2023). https://doi.org/10.1016/j.jlamp.2023.100860
22. Martin-Martin, E., Montenegro, M., Riesco, A., Rodríguez-Hortalá, J., Rubio, R.: Maude integration and verification for ROS Nav 2 (2021), https://github.com/demiourgoi/maudeROS
23. Martin-Martin, E., Montenegro, M., Riesco, A., Rodríguez-Hortalá, J., Rubio, R.: Verification of ROS Navigation using Maude. In: Martí-Oliet, N. (ed.) XX Jornadas de Programación y Lenguajes (PROLE). Sistedes (2021), https://hdl.handle.net/11705/PROLE/2021/008, superseded by [21].
24. Mason, I.A., Talcott, C.L.: IOP: the InterOperability Platform & IMaude: An interactive extension of Maude. In: Martí-Oliet, N. (ed.) Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications, WRLA 2004, Barcelona, Spain, March 27-28, 2004. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 315–333. Elsevier (2004). https://doi.org/10.1016/j.entcs.2004.06.016
25. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Pe-

tersburg, Russia, July 13-19, 2013. Proceedings. LNCS, vol. 8044, pp. 696–701. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_48
26. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
27. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
28. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings. LNCS, vol. 12699, pp. 625–635. Springer (2021). https://doi.org/10.1007/978-3-030-79876-5_37
29. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. J. Log. Algebraic Methods Program. **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
30. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Model checking strategy-controlled systems in rewriting logic. Autom. Softw. Eng. **28**(3), 55 (2021). https://doi.org/10.4230/LIPIcs.FSCD.2019.34
31. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Strategies, model checking and branching-time properties in Maude. J. Log. Algebraic Methods Program. **123** (2021). https://doi.org/10.1016/j.jlamp.2021.100700
32. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Metalevel transformation of strategies. J. Log. Algebr. Methods Program. **124** (2022). https://doi.org/10.1016/j.jlamp.2021.100728
33. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Simulating and model checking membrane systems using strategies in Maude. J. Log. Algebr. Methods Program. **124** (2022). https://doi.org/10.1016/j.jlamp.2021.100727
34. Santiago, S., Talcott, C.L., Escobar, S., Meadows, C.A., Meseguer, J.: A graphical user interface for Maude-NPA. In: Lucio, P., Moreno, G., Peña, R. (eds.) Proceedings of the Ninth Spanish Conference on Programming and Languages (PROLE 2009), San Sebastián, Spain, 9-11 September, 2009. Electron. Notes Theor. Comput. Sci., vol. 258(1), pp. 3–20. Elsevier (2009). https://doi.org/10.1016/j.entcs.2009.12.002
35. Sebastio, S., Vandin, A.: MultiVeStA: statistical model checking for discrete event simulators. In: Horváth, A., Buchholz, P., Cortellessa, V., Muscariello, L., Squillante, M.S. (eds.) 7th International Conference on Performance Evaluation Methodologies and Tools, ValueTools '13, Torino, Italy, December 10-12, 2013. pp. 310–315. ICST/ACM (2013). https://doi.org/10.4108/icst.valuetools.2013.254377
36. Talcott, C.L.: Pathway logic. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced Lectures. LNCS, vol. 5016, pp. 21–53. Springer (2008). https://doi.org/10.1007/978-3-540-68894-5_2

37. Zankl, H., Felgenhauer, B., Middeldorp, A.: CSI - A confluence tool. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings. LNCS, vol. 6803, pp. 499–505. Springer (2011). https://doi.org/10.1007/978-3-642-22438-6_38