Lupremica - Lua Scripting for Supremica

Citation for the original published paper (version of record):
Fabian, M., Malik, R., Mohajerani, S. (2023). Lupremica - Lua Scripting for Supremica. IFAC-PapersOnLine, 56(2): 6099-6104. http://dx.doi.org/10.1016/j.ifacol.2023.10.704

(article starts on next page)

Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnLine 56-2 (2023) 6099–6104

Lupremica – Lua Scripting for Supremica ⋆

Martin Fabian * **Robi Malik** ** **Sahar Mohajerani** *

* *Chalmers University of Technology, Gothenburg, Sweden (e-mail: {fabian, mohajera}@chalmers.se).*

** *University of Waikato, Hamilton, New Zealand (e-mail: robi@waikato.ac.nz)*

Abstract: Supremica is a software tool that implements several state-of-the-art algorithms to manipulate discrete-event systems, such as different types of compositions and compositional supervisor synthesis. Lua is a light-weight programming language suitable as a scripting language embedded into other applications. This paper describes the use of Lua as a scripting language for Supremica. To this end, the LuaJ interpreter is added to Supremica as a bridge between the Java-based implementation of Supremica and the Lua scripts. In this way, Supremica's entire Java API is made available to Lua scripts. Thus, scripts can automatically create automata, and manipulate them with all the algorithms available in Supremica and further manipulate the result with new algorithms implemented by Lua scripts. This opens up a new world of possibilities to try out new ideas and to extend the power of Supremica.

Keywords: Discrete-Event Systems, Synthesis, Verification, Scripting, Lua

1. INTRODUCTION

Discrete-Event Systems (DES) (Cassandras and Lafortune, 2008) are a modeling paradigm useful to model many engineered systems, such as manufacturing systems, traffic systems, and software controlled systems. DES occupy at each time instant one out of a finite number of *states*, and evolve by transiting between the states on the occurrences of *events*. Interaction between DES can be described by various types of *synchronous composition*, which require some events to occur simultaneously in two or more participating DES.

The *Supervisory Control Theory* (SCT) (Wonham and Cai, 2019) is a general approach to automatically compute, that is *synthesize*, control functions for DES. Given a *plant* to control and a *specification* of the desired controlled behavior, a *supervisor* can be synthesized that interacts with the plant and guarantees that the specification is always fulfilled. This *correct-by-construction* guarantee is a major advancement in development of DES, as it helps engineers managing the ever-increasing complexity of these types of systems (Goorden et al., 2021). To fully reap this benefit, tools able to handle industrial-sized systems have to be available, and these tools have to be user-friendly and customizable.

There exists a multitude of tools and library packages that implement manipulation of DES, including supervisor synthesis and verification (TC on DES, 2022). While many of these present full-fledged user interfaces, others implement a set of library functions meant to be used by user-implemented code. Both of these approaches have their advantages and drawbacks, and implementing both in a single tool, as some do, is a great benefit.

This paper is organized as follows. Section 2 briefly describes Supremica, and Section 3 briefly describes Lua. Then, Section 4 describes LuaJ, the Lua-Java bridge embedded in Supremica. Section 5 is the main section describing and exemplifying several Lua scripts and their execution in Supremica. Finally, Section 6 concludes the paper and suggests some future enhancements.

2. SUPREMICA

Supremica (Malik et al., 2017; Supremica Developers, 2022b) is a tool for modeling, analyzing, synthesizing and verifying discrete-event control systems. Though Supremica includes many standard SCT and DES algorithms, its main advancement is the abstraction-based compositional synthesis algorithms (Mohajerani et al., 2014) that can in a few seconds compute supervisors for systems of more than 10^{17} reachable states. For well-structured systems, such as the Transfer Line of Wonham and Cai (2019), these algorithms manage to find and exploit the structure to compute supervisors for systems of more than 10^{1505} reachable states.

Supremica supports ordinary "flat" Finite-State Machines (FSM), as well as *Extended Finite-State Machines* (EFSM) (Sköldstam et al., 2007), which allow compact models due to the inclusion of bounded discrete variables, and guards and actions over these variables associated to the transitions. Supremica can convert from EFSM to equivalent FSM representations by "flattening", and thus all the algorithms available for FSM can be used for EFSM (Malik et al., 2017).

Supremica provides an *integrated development environment (IDE)* that makes available all the actions to cre-

⋆ This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the Swedish Research Council (VR) under grant number 2016-06204 (SyTeC).

10.1016/j.ifacol.2023.10.704

ate, compose, analyze, verify, and synthesize discrete-event models through a graphical user interface. While this is a well-proven user-friendly approach, it is also a bit limiting, as the user can only the apply actions made available through menus and buttons. To overcome this limitation, SUPREMICA 2.8 now supports the use of scripts written in the Lua (Ierusalimschy et al., 2018) scripting language, as described in this paper.

The Java API available for SUPREMICA (Supremica Developers, 2022a), lists more than 19 000 public classes, interfaces, and methods. This entire API is made available to Lua scripts, which opens up a new world of possibilities to try out new ideas and extend the power of SUPREMICA.

3. LUA

Lua (Ierusalimschy et al., 2018) is a light-weight scripting language developed at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio). It is used in video games (Emmerich, 2009), TVs and set-top boxes (Januario et al., 2014), and other embedded devices (Lua Community, 2022), as well as for engineering applications (Pleune et al., 2020). Even the document processing tool TeX has a version with Lua embedded (LuaTeX.org, 2022).

A Lua file is read by the compiler from start to finish, and every line is executed. Typical scripts start with declarations of variables and functions, which are then used by program statements placed at the end. Lua source files can load other Lua files with the `dofile` or `require` commands. This is handy to build libraries that encapsulate frequently occurring constructs.

Lua is dynamically typed, meaning that only values have type, not variables. Thus, type definitions are absent, and all values carry their own type. All values are also first-class values, meaning that any value can be stored in variables, passed as arguments to functions, and returned as results; and this includes functions themselves.

The eight types supported by Lua are `nil`, `boolean`, `number`, `string`, `function`, `table`, `userdata`, and `thread`. Here, `nil` is a special type of value, different from all other values, typically representing the absence of a useful value; `boolean` is the type of the values *true* and *false*, and both `nil` and *false* make conditions false, any other value makes them true. The `number` type represents double-precision floating-point numbers, typically implemented as per the IEEE 754 standard. The `string` type represents immutable, pooled, sequences of bytes, that can include any 8-bit value, including zero. The `function` type represents Lua functions, which can take a variable number of parameters and return multiple values. The type `table` represents associative arrays that can be indexed by any Lua value except `nil`. The `userdata` type allows arbitrary C data to be stored in Lua variables, and is a pointer to a block of memory. The `thread` type represents independent threads of execution and is used to implement coroutines.

As mentioned above, SUPREMICA's entire Java API is available to Lua scripts. The benefit of Lua over direct programming with this API in Java is that Lua is much less verbose; there simply is less to write in Lua compared to Java, even though the API bindings have to be made explicit.

4. LUAJ

LuaJ (Roseborough and Farmer, 2014) is a Lua virtual machine written in Java. It includes a compiler that compiles Lua source code to Java bytecode. LuaJ version 3.0.1, used with SUPREMICA 2.8, supports Lua version 5.2, with all standard features of the language.

The intention behind LuaJ is to allow embedding into Java applications, and a major goal has been to achieve good performance. In some benchmarks, Lua compiled to Java bytecode executes faster than C-based Lua (Roseborough and Farmer, 2014).

LuaJ loads and compiles Lua scripts as *chunks* that are then evaluated as executable `LuaValue` instances. LuaJ allows to run Lua scripts as Java applications, as well as MIDlets, and JSR-223 Dynamic scripts. SUPREMICA only supports the Java application type.

Using LuaJ in a Java app is a simple matter of including the library `luaj-jse-3.0.1.jar` in the class path when compiling. SUPREMICA embeds LuaJ at compile time into the `SupremicaLib.jar` file to avoid any external dependencies.

The straightforward way to run a Lua script inside a Java application is, as shown by Roseborough and Farmer (2014):

```
import org.luaj.vm2.*;
import org.luaj.vm2.lib.jse.*;

Globals globals = JsePlatform.standardGlobals();
LuaValue chunk = globals.loadfile(script);
chunk.call();
```

This creates a standard global environment for the script to run in, then loads the script and executes it. This is in essence the way Lua scripts are run inside SUPREMICA.

LuaJ uses *reflection* (McCluskey, 1998) for its bindings between Lua and Java. Java 9 introduced the *module* concept, which restricts many reflection operations. Thus, for full functionality, LuaJ and Lua scripts in SUPREMICA require Java 8.

5. LUA SCRIPTS IN SUPREMICA

Lua scripts are run from inside SUPREMICA through the new (in version 2.8) menu option `File > Run Script...` (shortcut Ctrl/Cmd+R) This will open a file chooser dialog box in the default `scripts` folder filtering out the *.lua files, see Fig. 1. The Lua file chosen from here is then loaded and run. SUPREMICA's default `scripts` folder contains many Lua scripts aimed at illustrating how to write such scripts.

The first thing almost every script has to do is to get a reference to the SUPREMICA IDE. This can be achieved in two ways; either by simply accepting the arguments that are passed to the script, as:

```
local script, ide, log = ... -- params from Supremica
```

The double hyphen starts a Lua comment, and the triple dots is Lua syntax for variable number of arguments, the first two of which are the name of the script itself, and a reference to the SUPREMICA IDE that currently

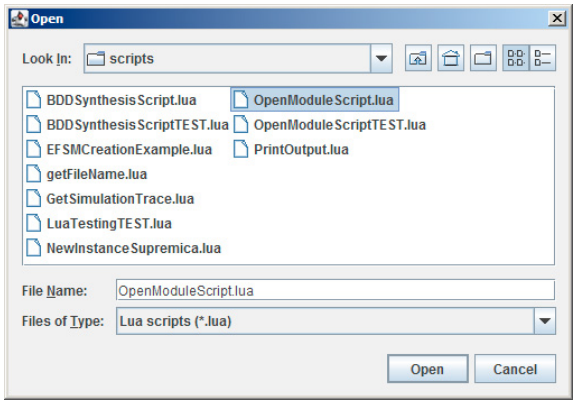

Fig. 1. Dialog to choose and run Lua scripts.

runs the script. The `log` variable receives a reference to SUPREMICA's logging pane, which is useful for writing messages to the user, especially when debugging scripts.

The other way to get a reference to the IDE looks a bit more complicated, but also illustrates some general techniques for accessing a Java API via LuaJ.

```
local IDE = luajava.bindClass("org.supremica.gui.ide.
    ↪ IDE")
local ide = IDE:getTheIDE()
local log = IDE:getTheLog()
```

Here, `luajava` is a reference to the LuaJ bridge, and the call to `luajava.bindClass` tells LuaJ to return a reference to the `org.supremica.gui.ide.IDE` class into the `IDE` variable. This approach to bind a Java class reference to a local Lua variable is general and appears in most scripts.

The `IDE` variable on the next line is used to put a reference to the currently running SUPREMICA instance into the local `ide` variable by calling the static method `getTheIDE()` of class `IDE`. And a similar thing happens for the `log` variable. The `ide` and `log` variables can now be used to access the currently running SUPREMICA instance.

In the following it is assumed that the `ide` and `log` variables exist and reference the currently running SUPREMICA instance. For brevity we will also assume that the `luajava` reference to LuaJ has been aliased to `luaj`, which is done inside the Lua file as:

```
local luaj = luajava
```

This creates the local variable `luaj` as a shorthand for `luajava`. The reference being local also brings slight performance improvements.

5.1 Creating Automata

A script can load automata of various forms into SUPREMICA either as a new project, called *module*, or as individual automata inside a module created by the script.

Open a module SUPREMICA modules are stored in XML files with the `wmod` extension. Three lines of Lua code load such files:

```
local file = luaj.newInstance("java.io.File", fname)
local manager = ide:getDocumentContainerManager()
manager:openContainer(file)
```

The first line asks LuaJ to create a new instance of the Java File class, and instantiate it with the file name given in `fname`. Then the `ide` is asked to create a new document container manager, and this manager is then asked to open the given file. The result is that a new module containing the automata in the `wmod` file is displayed in the IDE.

Create automata Programmatically creating automata is a little more involved, especially as concerns EFSM. First the events, variables, locations, and edges with guards and actions have to be created. These are then combined into an EFSM, and then loaded into the currently open module.

In SUPREMICA 2.8's `scripts` folder is an example of this, `EFSMsimpleCreate.lua`, the main parts of which are listed below. The script creates inside SUPREMICA the EFSM shown in Fig. 2.

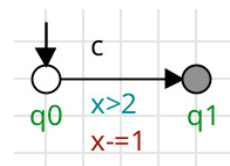

Fig. 2. EFSM resulting from the Lua script below.

This EFSM has two locations, named `q0` and `q1`, with `q0` inital and `q1` marked, and a transition from `q0` to `q1` labeled by a controllable event `c`. The transition is guarded by the expression `x > 2`, and can thus only execute when the variable `x` has a value greater than two. When the transition is executed, the action `x -= 1` decreases the value of `x` by one.

The script to create this EFSM starts with binding some Java classes and interfaces to Lua variables, as done in the previous examples. This is akin to Java's `import` statements.

```
local waters = "net.sourceforge.waters." -- shorthand

local ModuleSubjectFactory = luaj.bindClass(waters..
    ↪ "subject.module.ModuleSubjectFactory")
local CompilerOperatorTable = luaj.bindClass(waters..
    ↪ "model.compiler.CompilerOperatorTable")
local EventDeclProxy = luaj.bindClass(waters..
    ↪ "model.module.EventDeclProxy")
local EventKind = luaj.bindClass(waters..
    ↪ "model.base.EventKind")
local ComponentKind = luaj.bindClass(waters..
    ↪ "model.base.ComponentKind")
local Collections = luaj.bindClass(
    ↪ "java.util.Collections")
local factory = ModuleSubjectFactory:getInstance()
local optable = CompilerOperatorTable:getInstance()
```

The double dot is Lua's string concatenation operator, and it is used to generate the fully qualified Java package names, with the contents of the `waters` variable as base.

Then the code creates some convenience functions to easily create an event:

```
local function createEvent(name, kind)
  local eventName = factory:createSimpleIdentifierProxy(
      ↪ name)
  local event = factory:createEventDeclProxy(eventName,
      ↪ kind)
  return event
end -- createEvent
```

a location:

```
local function createLocation(label, initial, marked)
  if not marked then -- Create nonmarked location
    return factory:createSimpleNodeProxy(label, nil, nil
        ↪ , initial, nil, nil, nil)
  end
  -- Create a marked location
  local nodeLabelAccepting = factory:
      ↪ createSimpleIdentifierProxy(
      ↪ EventDeclProxy.DEFAULT_MARKING_NAME)
  local nodeLabelList = Collections:singletonList(
      ↪ nodeLabelAccepting)
  local marking = factory:createPlainEventListProxy(
      ↪ nodeLabelList)
  return factory:createSimpleNodeProxy(label, marking,
      ↪ nil, initial, nil, nil, nil)
end -- createLocation
```

an integer variable:

```
local function createIntegerVariable(name, min, max,
    ↪ init)
  local varName = factory:createSimpleIdentifierProxy(
      ↪ name)
  local varMin = factory:createIntConstantProxy(min)
  local varMax = factory:createIntConstantProxy(max)
  local varRange = factory:createBinaryExpressionProxy(
      ↪ optable:getRangeOperator(), varMin, varMax)
  local varRef = factory:createSimpleIdentifierProxy(
      ↪ name)
  local varInitVal = factory:createIntConstantProxy(init
      ↪ )
  local varInitPred = factory:
      ↪ createBinaryExpressionProxy(optable:
      ↪ getEqualsOperator(), varRef, varInitVal)
  local var = factory:createVariableComponentProxy(
      ↪ varName, varRange, varInitPred)
  return var
end -- createIntegerVariable
```

an enumeration:

```
local function createEnumeration(name, values, init)
  local varName = factory:createSimpleIdentifierProxy(
      ↪ name)
  local enumMembers = luaj.newInstance("
      ↪ java.util.ArrayList", #values)
  for i = 1, #values do
    local member = factory:createSimpleIdentifierProxy(
        ↪ values[i])
    enumMembers:add(member)
  end
  local varRange = factory:createEnumSetExpressionProxy(
      ↪ enumMembers)
  local varRef = factory:createSimpleIdentifierProxy(
      ↪ name)
  local varInitVal = factory:createSimpleIdentifierProxy
      ↪ (init)
  local varInitPred = factory:
      ↪ createBinaryExpressionProxy(optable:
      ↪ getEqualsOperator(), varRef, varInitVal)
  local var = factory:createVariableComponentProxy(
      ↪ varName, varRange, varInitPred)
  return var
end -- createEnumeration
```

a binary expression:

```
local function createBinaryExpression(op1, op, op2)
  local function getOperandType(operand)
    if type(operand) == "number" then
      return factory:createIntConstantProxy(tonumber(
          ↪ operand))
    elseif type(operand) == "string" then
      return factory:createSimpleIdentifierProxy(operand
          ↪ )
    end
    -- Type is neither number nor string, just return as
        ↪ is
    return operand
  end
  local operand1 = getOperandType(op1)
  local operand2 = getOperandType(op2)
  return factory:createBinaryExpressionProxy(op,
      ↪ operand1, operand2)
end -- createBinaryExpression
```

and a label block:

```
local function createLabelBlock(events)
  local labels = luaj.newInstance("java.util.ArrayList",
      ↪ #events)
  for i = 1, #events do
    local event = factory:createSimpleIdentifierProxy(
        ↪ events[i])
    labels:add(event)
  end
  return factory:createLabelBlockProxy(labels, nil)
end -- createLabelBlock
```

Also, a variable-argument list-creation function is given as:

```
local function makeList(...)
  local args = {...} -- variable arguments list as table
  if #args == 1 then
    return Collections:singletonList(args[1])
  end
  local list = luaj.newInstance("java.util.ArrayList", #
      ↪ args)
  for i = 1, #args do
    list:add(args[i])
  end
  return list
end -- makeList
```

Here, the Lua script creates a new instance of the `java.util.ArrayList` and fills it in with the elements given as arguments to the function (if more than one).

These convenience functions can be stored in a Lua file and reused between scripts. With these functions available, we can generate an EFSM:

```
-- controllable event "c", stored in an alphabet
local eventC = createEvent("c", EventKind.CONTROLLABLE)
-- marking proposition
local mark = createEvent(
    ↪ EventDeclProxy.DEFAULT_MARKING_NAME,
    ↪ EventKind.PROPOSITION)
local alphabet = makeList(eventC, mark)
-- integer variable x, range 0..10, initial value 5
local varX = createIntegerVariable("x", 0, 10, 5)
-- locations: q0, initial, unmarked; q1, marked
local loc0 = createLocation("q0", true, false)
local loc1 = createLocation("q1", false, true)
local locations = makeList(loc0, loc1)
-- create guard x > 2
local guard = createBinaryExpression("x", optable:
    ↪ getGreaterThanOperator(), 2)
local guards = makeList(guard)
-- create action x -= 1
local action = createBinaryExpression("x", optable:
    ↪ getDecrementOperator(), 1)
local actions = makeList(action)
local label = createLabelBlock({"c"})
-- create a guard/action block
local gaBlock = factory:createGuardActionBlockProxy(
    ↪ guards, actions, nil)
-- edge from q0 to q1
local edge = factory:createEdgeProxy(loc0, loc1, label,
    ↪ gaBlock)
local edges = makeList(edge)
-- deterministic, no blocked events, locations and edges
local graph = factory:createGraphProxy(true, nil,
    ↪ locations, edges)
local efsmName = factory:createSimpleIdentifierProxy(
    ↪ "MyEFSM")
```

```
local efsm = factory:createSimpleComponentProxy(efsmName
    ↪ , ComponentKind.PLANT, graph)
-- put the components together
local components = makeList(varX, efsm)
-- Combine events, variables, and EFSM to make module
local mod = factory:createModuleProxy("MyModule",
    ↪ "Module comment", nil, nil, alphabet, nil,
    ↪ components);
```

Once the module is created, it can be saved:

```
local MarshallingTools = luaj.bindClass(waters..
    ↪ "model.marshaller.MarshallingTools")
MarshallingTools:saveModule(mod, "in/this/path")
```

The above code is available as `EFSMsimpleCreate.lua` in Supremica 2.8. Though it may look a mouthful, consider that convenience functions can be created to hide the direct calls to Supremica's Java API.

5.2 Synthesize a supervisor

Once there are automata in the currently open Supremica module, a script can manipulate them, for instance by applying a standard monolithic synthesis operation (Wonham and Cai, 2019) as demonstrated below. Other operations such as verification follow a similar pattern. The file `MonolithicSynthesis.lua` is available in Supremica 2.8's `scripts` folder, here we just point out the basics.

First we bind some Java classes to Lua variables:

```
local ProductDESElementFactory = luaj.bindClass(waters..
    ↪ "plain.des.ProductDESElementFactory")
local ModuleSubjectFactory = luaj.bindClass(waters..
    ↪ "subject.module.ModuleSubjectFactory")
```

Then, we create a compiler to "flatten" EFSM. This is necessary, since the monolithic synthesis works on ordinary FSM, and cannot cope with variables or guards or actions. The compiler is then configured, and asked to compile the components into a set of ordinary FSM:

```
local manager = ide:getDocumentManager()
local desFactory = ProductDESElementFactory:
    ↪ getInstance()
local module = ide:getActiveDocumentContainer():
    ↪ getEditorPanel():getModuleSubject()
local compiler = luaj.newInstance(waters..
    ↪ "model.compiler.ModuleCompiler", manager,
    ↪ desFactory, module)
-- Configure the compiler...
-- optimisation removes selfloops and redundant
    ↪ components
compiler:setOptimizationEnabled(true)
-- normalisation is needed for advanced feature modules
compiler:setNormalizationEnabled(true)
-- only report the first error even if there are several
compiler:setMultiExceptionsEnabled(false)
-- Compile the module
local des = compiler:compile()
```

The set of FSM are now in the `des` variable. A monolithic synthesizer is instantiated and invoked on the FSM:

```
local synthesizer = luaj.newInstance(waters..
    ↪ "analysis.monolithic.MonolithicSynthesizer",
    ↪ desFactory)
synthesizer:setModel(des)
synthesizer:run()
```

The result from the synthesis can be empty, that is, no supervisor exists, and then there is nothing to do. Else, if the synthesis result consists of one (or more, as in the general case with non-monolithic synthesis) FSM, we add it to the module that the components came from:

```
local result = synthesizer:getAnalysisResult()
if result:isSatisfied() then
  local supervisor = result:getComputedProductDES()
  local factory = ModuleSubjectFactory:getInstance()
  local importer = luaj.newInstance(waters.."
      ↪ model.marshaller.ProductDESImporter", factory)
  -- iterate over all components and add to the module
  local iterator = supervisor:getAutomata():iterator()
  while iterator:hasNext() do
    local aut = iterator:next()
    local comp = importer:importComponent(aut)
    module:getComponentListModifiable():add(comp)
  end -- while
end -- if
```

Instead of adding the synthesized result back to the module, we could save it to a file:

```
local supervisor = result:getComputedProductDES()
-- bind the MarshallingTools class to Lua variable
local MarshallingTools = luaj.bindClass(waters..
    ↪ "model.marshaller.MarshallingTools")
MarshallingTools:saveProductDES(supervisor, getFileName(
    ↪ "supervisor.wdes"));
```

This code uses the convenience function `getFileName`, which is available in the `scripts` folder and can be loaded into a script with:

```
local Config = luaj.bindClass(
    ↪ "org.supremica.properties.Config")
local getFileName = dofile(Config.FILE_SCRIPT_PATH:
    ↪ getValue():getPath().."/getFileName.lua")
```

This uses `dofile` to read in the `getFileName` function from the `getfileName.lua` file in the set `scripts` folder. The function generates a file name valid for the particular operating system, using the default save path that is set in Supremica's IDE under the Configure menu.

5.3 User Interfacing

Supremica redirects the output of Lua's `print` function to the logger pane, which allows for some feedback to the user. By using the `log` variable, initialized as described above, all the functions of the standard logger are available:

```
-- Print calls are intercepted and shown in the logger
    ↪ pane
print("Let's print an important number: ", 42, "!")
-- Supremica's standard logger is also available
log:error("This is an error message!", 0)
log:info("This is just info (same as 'print')", 0)
log:debug("Debug messages also work", 0)
log:trace("This is a trace message", 0)

print(_VERSION) -- print LuaJ version
```

The above script results in the logger pane output shown in Fig. 3.

Scripts may also want to interface with the user in more advanced ways, like getting keyboard input etc. For this, Lua scripts can use all of the existing Java components. For instance, opening a file chooser is done by a script like:

```
local fc = luaj.newInstance("javax.swing.JFileChooser")
fc:setDialogTitle("Lupremica file chooser")
local retval = fc:showOpenDialog()
```

```
Lua script: D:\Extra\Supremica\scripts\PrintOutput.lua
Let's print an important number: 42!
This is an error message!
This is just info (same as 'print')
Debug messages also work
Luaj-jse 3.0.1
```

Fig. 3. Output from Lua script in the logger pane.

```
if retval == fc.APPROVE_OPTION then
  local fname = fc:getSelectedFile():getPath()
  print("File: "..fname)
else
  print("User cancelled")
end
```

This will display the full path and name of the chosen file (if any) in the logger pane.

In the same way as for the `print` function, output from the Lua interpreter appears in SUPREMICA's logger pane. Error messages are admittedly cryptic, but at least they give the script name and a line number, which is a first help. For instance, Fig. 4 shows the error given when a script tries to add an element to a singleton list returned by `makeList` given above.

```
Waters/Supremica IDE 2.7.1, built 9 okt. 2022 11:37:39 CEST
JVM:17.0.2, Free/Total/Max mem: 2012/2048/8192 MiB
Lua script: D:\Extra\Supremica\scripts\EFSMsimpleCreateTEST.lua
Something went wrong, sorry!
org.luaj.vm2.LuaError: @D:\Extra\Supremica\scripts\EFSMsimpleCreateTEST.lua:147 vm
error: java.lang.reflect.InaccessibleObjectException: Unable to make public java.lang.Object
java.util.Collections$SingletonList.get(int) accessible: module java.base does not "opens
java.util" to unnamed module @30c15d8b
```

Fig. 4. Lua compiler error on line 147 in the script `EFSMsimpleCreate.lua`.

6. CONCLUSIONS

This paper presents the new scripting possibilities included in SUPREMICA 2.8, which uses the Lua scripting language through the LuaJ interpreter as a bridge between Lua and the Java API of SUPREMICA.

Many of the Lua code examples presented in this paper are available in SUPREMICA 2.8 in the `scripts` folder.

Currently, the Lua scripting implementation in SUPREMICA is rather bare-bones. The main thing missing is a set of Lua files to be used as libraries by user scripts via the `dofile` or `require` calls. Such libraries would simplify the API calls and hide a lot of complexity for the user. Things like

```
local EventKind = luaj.bindClass(waters.."model.base.
    ↪ EventKind")
```

which binds the Java enumeration `EventKind` to the same-named local Lua variable, would then be readily available and not have to be written into every script file. Also, a lot of the complex code to, for instance, generate automata and EFSM could be hidden inside such library functions. Work on this is on-going.

REFERENCES

Cassandras, C.G. and Lafortune, S. (2008). *Introduction to Discrete Event Systems*. Springer US, Boston, MA. doi:10.1007/978-0-387-68612-7_2.

Emmerich, P. (2009). *Beginning Lua with World of Warcraft Add-ons*. Apress, Berkeley, CA. doi:10.1007/978-1-4302-2372-6_1.

Goorden, M.A., Fabian, M., van de Mortel-Fronczak, J.M., Reniers, M.A., Fokkink, W.J., and Rooda, J.E. (2021). Compositional coordinator synthesis of extended finite automata. *Discrete Event Dynamic Systems*, 31, 317–348. doi:10.1007/s10626-020-00334-w.

Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. (2018). A look at the design of Lua. *Communications of the ACM*, 61, 114–123. doi:10.1145/3186277.

Januario, F.A.P., Cordeiro, L.C., De Lucena, V.F., and De Lima Filho, E.B. (2014). BMCLua: Verification of Lua programs in digital TV interactive applications. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, 707–708. doi:10.1109/GCCE.2014.7031344.

Lua Community (2022). Lua Uses. `https://www.lua.org/uses.html`. Accessed: 2022-10-07.

LuaTeX.org (2022). LuaTeX. `https://www.luatex.org/`. Accessed: 2022-10-29.

Malik, R., Åkesson, K., Flordal, H., and Fabian, M. (2017). Supremica–an efficient tool for large-scale discrete event systems. *IFAC-PapersOnLine*, 50(1), 5794–5799. doi:10.1016/j.ifacol.2017.08.427.

McCluskey, G. (1998). Using java reflection. `https://www.oracle.com/technical-resources/articles/java/javareflection.html`. Accessed: 2023-03-23.

Mohajerani, S., Malik, R., and Fabian, M. (2014). A framework for compositional synthesis of modular nonblocking supervisors. *IEEE Transactions on Automatic Control*, 59(1), 150–162. doi:10.1109/TAC.2013.2283109.

Pleune, M., Paul, N., Faulkner, C., and Chung, C.J. (2020). Specifying route behaviors of self-driving vehicles in ROS using Lua scripting language with web interface. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, 535–539. doi:10.1109/EIT48999.2020.9208285.

Roseborough, J. and Farmer, I. (2014). Getting started with LuaJ. `http://www.luaj.org/luaj/3.0/README.html`. Accessed: 2022-10-09.

Sköldstam, M., Åkesson, K., and Fabian, M. (2007). Modeling of discrete event systems using finite automata with variables. In *2007 46th IEEE Conference on Decision and Control*, 3387–3392. doi:10.1109/CDC.2007.4434894.

Supremica Developers (2022a). Supremica – the API. `https://www.cs.waikato.ac.nz/~robi/waters-doc/index.html`. Accessed: 2022-10-15.

Supremica Developers (2022b). Supremica – the Web Site. `https://supremica.org/`. Accessed: 2022-10-10.

TC on DES (2022). Discrete event systems resources – software tools. `http://ieeecss.org/tc/discrete-event-systems/resources`. Accessed: 2022-10-13.

Wonham, W.M. and Cai, K. (2019). *Supervisory Control of Discrete-Event Systems*. Springer International Publishing, Cham. doi:10.1007/978-3-319-77452-7.