General Purpose Lua Interpreter for Modelica

Fabian Buse[1] Tobias Bellmann[1]

[1]Institute of System Dynamics and Control, German Aerospace Center (DLR), Germany,
{fabian.buse,tobias.bellmann}@dlr.de

Abstract

Simulation becomes more and more important in the development of complex systems. Modeled systems often are comprised of mechanical, electrical as well as software systems. It is often not possible to evaluate the performance of a system without considering some higher level logic anymore. Scripting languages, such as Lua, are usually well suited to implement these logic elements. This paper shows the integration of the Lua interpreter into Modelica, and gives examples how the library can be used to help with the simulation of industrial robots or in the development of a planetary exploration rover in the MMX Mission.

Keywords: Modelica, Lua, script language, robotics, finite state machine

1 Introduction

When simulating complex systems, it is often necessary to model not only the mechanics, electronics and control system correctly, but also the higher-level logic that governs the general behavior. However, this high-level logic is generally difficult to implement with native Modelica methods. Although it is possible to build a control logic based on multiple inputs and outputs, e.g. by using state machines, it is error-prone and not very flexible. The requirement to balance all equations and unknowns can be difficult and tedious if the different control states or modes have distinct structures. If a model is to be used to test and develop this high-level logic, the limitations of native Modelica methods are even more pronounced. The developer with in-depth Modelica knowledge may not be responsible for developing and testing the high-level logic. One common method to solve this problem is the FMI standard (Blockwitz et al. 2012). With this approach the system model is built in Modelica and then imported into a different tool that is better suited to handle high-level logic. The Modelica Lua library presented in this paper was developed to combine Modelica's strengths in physical modeling with the flexibility of a script-based interpreter to solve the problem stated before without the need of an additional tool.

The concept of this library is an extension and generalization of the Lua interpreter presented and developed for the DLR Robots library (Bellmann, Seefried, and Thiele 2020). Lua was chosen due to its lightweight, well maintained interpreter, its simple and easy to learn syntax and already established application in other fields, such as game development. With Lua being a commonly used language, many tools and third party libraries are already available to use.

1.1 Design Goal

The goal for this library was to provide an application independent Lua interpreter with a lightweight interface. The integration into existing Modelica models should be as easy as possible. All basic Modelica data types, Real, Integer, Boolean and String, should be supported as input and output of Lua. In order to meet the requirements of most applications, different modes of operation shall be supported, both a loop-like repetition of the Lua script and a single and serial execution of the script. To allow multiple instances or different systems controlled by Lua in a single Modelica model, it is desired to allow multiple, independent instances of the Lua interpreter in the same Modelica model. It should also be possible to write Lua in Modelica directly as a string or to load a script from disk. It should be possible to use the full Lua functions and also to enable third-party Lua libraries. Finally, the Lua library should be platform-independent.

2 Structure of the library

From the previously stated goals a simple structure of the library has been developed. The following section will explain the general setup, both interfaces to the Modelica and Lua side of the library as well as some further considerations.

2.1 Implementation

Similar to the implementation of the Lua interpreter in the DLR Robots library, each Lua interpreter instance runs in a separate thread independent of the main Modelica thread. The data exchange between the interpreter and Modelica takes place via a data core which can be accessed by Modelica as well as Lua. Extending from the implementation in the DLR Robots library, where no further synchronization between Lua and Modelica was possible, the new Lua library offers various options. Similar to the original implementation, both programs can run completely independently of each other, with the timing on both sides being handled manually by custom signals transmitted through the data core. To enable the timed execution of certain instructions, waiting functions based either on simulation-time or CPU-time are now provided.

Alternatively, the Lua code can be synchronized to specific time events. This can be further configured so that Lua waits for Modelica, Modelica for Lua, or both. If synchronization is activated, the data is only read and written to the data core at this time to ensure data coherence.

2.2 Modelica Interface

The Modelica library, shown in Figure 1, was kept as simple as possible. The communication, script selection and synchronization is handled by the main Lua block, inputs and outputs are handled by their type specific blocks. To reduce the total number of required blocks all interfaces except strings are treated as vectors.

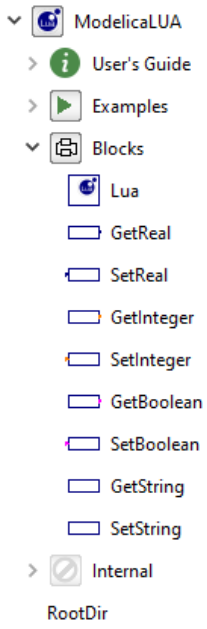

Figure 1. Structure of the Modelica Lua library as shown in Dymola.

To define the hierarchy and enable multiple interpreter instances an inner outer construct was chosen. Each instance of a `Lua` block holds one independent interpreter, thus the set and get blocks have their interpreter assigned based on the model structure in Modelica. The main `Lua` block contains all parameters defining the created instance. These parameters can be divided into three categories, script, timing and synchronization. The script category, defines what script to load or if selected, the string containing the entire script, as well as a unique name identifying the interpreter instance. Additionally, further search directories for locating scripts or libraries can be defined. The timing category defines the sampling time for communication, as well as the start time for the script. The different synchronization modes are configured here as well, if synchronization is enabled, the communication time step is used as the synchronization time step. In contrast, each get or set block is only parameterized by its name and dimension. If the dimension of a value in the data core does not match the requested size, the returned data will be truncated or padded with zeros.

One critical feature for a flexible usage of this library is a robust initialization setup. To ensure that the representation in the data core is always consistent with the model state in Modelica, the initialization of both get and set blocks is handled in Modelica. For the set functions the initial value of the input is simply communicated in an initial equation to the data core. For the get blocks either an explicit start value can be defined or the start value can be implicitly defined by the Modelica models and equations connected to its output.

2.3 Lua Interpreter

The Lua interpreter itself is based on the official sources provided by the Lua community[1]. By encapsulating both the Lua stack and an additional data broker into an external object, each instance of the `Lua` block has separate interpreters. Due to the separation of the Modelica and Lua thread an intermediate data core is required. On both sides bindings to the data core have been implemented to enable thread safe access. The corresponding Modelica blocks were shown above. Lua bindings are provided in an additional Lua library `modelica.lua`, see Table 1, its functions correspond to the Modelica get and set blocks as well as additional Modelica interfaces. Get methods in Lua have an optional `dim` parameter. If `dim` is defined and larger than zero, the returned data will have this dimension and will be truncated or padded with zeros to match the desired size whereas if it is not defined or zero, the returned value has the dimension currently present in the data core.

2.4 Synchronization

The functions `sync`, `wait` and `wait_until` provide means to synchronize the execution of the Lua code to Modelica. Two approaches are available. First, a program is executed sequentially and uses the `wait` and `wait_until` to perform actions at predefined times. Both functions block until the required time has passed:

```
local t = 0
t = Modelica.time() -- t = 0.0
Modelica.wait_until(10)
t = Modelica.time() -- t = 10.0
Modelica.wait(2)
t = Modelica.time() -- t = 12.0
```

The `wait` function can either use simulation or CPU time, with simulation time as a default. Whereas the `wait_until` is always linked to simulation time. Alternatively, the code in Lua can be structured in loop, where every cycle is synchronized to a sampled clock in Modelica by using `sync`:

```
local t = 0
while(Modelica.sync())
do
 -- this loop is executed with
 -- the sample time defined in Modelica
 t = Modelica.time()
 -- t = current simulation time
end
```

The `sync` function blocks the Lua thread until the next pulse of the clock in Modelica, its rate is defined in the parameters of the `Lua` block. If the execution of the Lua code takes longer than the defined sample rate, it is possible to block the execution in the Modelica thread until the `sync` function is reached again.

2.5 Lua Libraries

One of the key features of Lua is the ability to use additional libraries. This enables not only the usage of the extensive Lua

[1] `https://www.lua.org/`

DOI 10.3384/ecp21181425

Table 1. Modelica interface functions provide in Lua.

Function	*Description*
`sync()`	synchronizes Lua to Modelica
`dt()`	returns the current Lua time step, only works if `sync` is used
`time()`	returns the current simulation time in seconds
`setReal(name,u)`	writes a real value or vector `u` to the data core
`setInteger(name,u)`	writes an integer value or vector `u` to the data core
`setBoolean(name,u)`	writes a boolean value or vector `u` to the data core
`setString(name,u)`	writes a single string `u` to the data core
`getReal(name,dim)`	reads a real value or vector from the data core, `dim` is optional
`getInteger(name,dim)`	returns an integer value or vector from the data core, `dim` is optional
`getBoolean(name,dim)`	returns a boolean value or vector from the data core, `dim` is optional
`getString(name)`	returns a single string from the data core
`wait(duration, cpu_time)`	blocks for a specific duration, either in simulation or cpu time, if `cpu_time` is not set it defaults to `false`
`wait_until(time)`	blocks until the simulation time is reached
`print(msg)`	prints a message with the `ModelicaFormatMessage` function, usually to the log file resulting from model execution
`terminate(msg)`	terminates the simulation, equivalent to `terminate` in Modelica

standard library which already supports basic file system support, basic math functions, and string manipulations but also useful features like 3D vector math (Bjorn 2021) or finite state machines (Conroy 2021). These libraries can be easily added to the Lua script via the Lua `require` functionality, which is similar to the C/C++ include. This `require` functionality is also used to include the Modelica add-on to each loaded Lua script.

2.6 Logging

During the execution of a Lua script, the user can print out messages over the logging interface. Several output channels can be used:

- Log messages to a file:

  ```
  Logger.createFileOutput(title,
  filename,append)
  ```

- The system console is used for logging:

  ```
  Logger.createConsoleOutput(title)
  ```

- The Modelica message command is used to log to the Modelica tool:

  ```
  Logger.createModelicaLogOutput(title)
  ```

- A logging window is created to log messages in. (MS Windows only):

  ```
  Logger.createLogWindow(title,x,y,w,h)
  ```

After the initialization, the `Logger.log` command can be used to log information in one of the output channels. The channel to be used is defined by the parameter string `title`:

```
Logger.createLogWindow('Lua script log'
    ,100,100,300,150);
Logger.log('Lua script log', 'Some text
    for the console')
```

It is also possible to add a timestamp at the beginning of the text or, in case of the log window define different colors via the severity flag (See Figure 2) or filter the shown messages (errors only, errors and warnings, all) by setting the verbosity of the log channel.

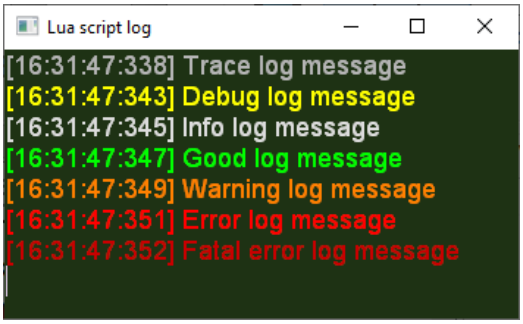

Figure 2. Logging window with log messages of varying severity and timestamps.

3 Examples

3.1 Library Examples

To provide an overview and show the functionalities some of the library's simple examples models will be shown here. The examples in this section would be rather easy to replicate in pure Modelica but are designed to highlight some of the core features.

The first example, shown in Figure 3, uses a synchronized Lua script to count the number of times an input value `u` exceeds a threshold of 0.5 and output it as `y`. Once the counter exceeds five, it is reset to zero. The initial value of the counter is set in Modelica. The used Lua code:

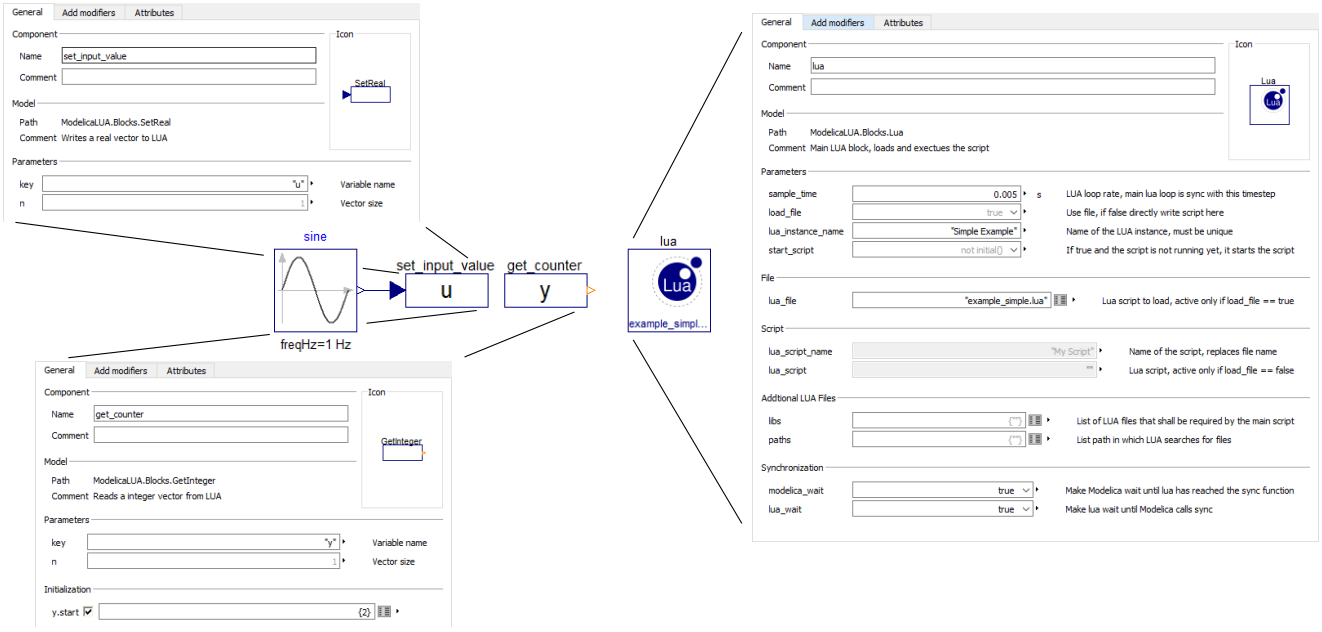

Figure 3. Diagram layer of a simple Lua example. Parameters of the Lua, input and output blocks are shown in more details.

```
local hysteresis = true
local threshold = 0.5
while(Modelica.sync())
do
 local u = Modelica.getReal('u')
 local y = Modelica.getInteger('y')
 if u > threshold and hysteresis then
  y = y + 1
  hysteresis = false
 elseif u < threshold then
  hysteresis = true
 end
 if y > 5 then
  y = 0
 end
 Modelica.setInteger('y',y)
end
```

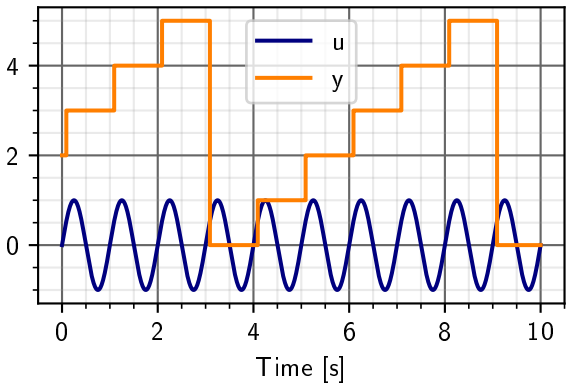

Figure 4. Input `u` and output `y` of the simple Lua example.

reads out the initial values of the counter an then goes into a while loop that is synchronized to Modelica with the `sync` function. A simple **`if elseif`** block increments the counter `y` when the input `u` exceeds the threshold. The resulting behavior is shown in Figure 4.

A second example uses the publicly available finite state machine implementation by (Conroy 2021). In this example a state machine switches between its states `Green`, `Yellow`, `Red` and `Black` based on thresholds of a single input signal. When the state `Red` is entered the counter `nAlerts` is incremented by one, once this counter exceeds 20 the final state `Black` is activated. The behavior of the output variable `result` can be defined for each state individually. In case of the state `Red`, `result` variable will hold the time since the state became active. A variable `state` is used to communicate the active state encoded into an integer to Modelica. The Modelica model, shown in Figure 5 is rather simple and just provides the necessary inputs and outputs. The Lua code first defines the structure of the state machine based on its state transitions by defining the event name as well as the start and end of the transition. Optionally `onEnter`, `onExit` and `run` functions can be defined for each sate. This structure and the function definition for the `Red` state, as well as the Lua code triggering the events is shown here:

```
-- import state machine library
local machine = require('statemachine')
local nAlerts = 0
local tRed = 0 -- time when red becomes
    active
-- state machine definition
local fsm = machine.create({
initial = 'Green',
events = { -- event definitions
 { name = 'warn',
   from = 'Green',
   to = 'Yellow' },
 { name = 'alarm',
   from = 'Yellow',
   to = 'Red'    },
 { name = 'calm',
   from = 'Red',
```


DOI 10.3384/ecp21181425

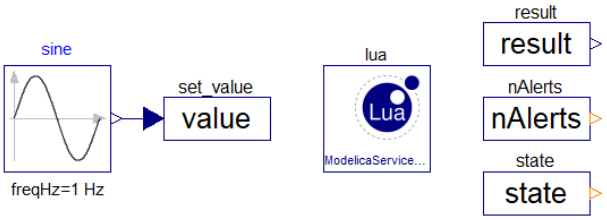

Figure 5. Input `value` and outputs `result`, `nAlerts` and `state` of the simple state machine Lua example

```
    to = 'Yellow' },
  { name = 'clear',
    from = {'Yellow','Red' },
    to = 'Green'  },
  { name = 'panic',
    from = 'Red',
    to = 'Black'},
},

callbacks = { -- callback definitions

-- define function when Red becomes
    active
onenterRed= function()
 Modelica.print('Entering Red')
 nAlerts = nAlerts + 1
 Modelica.setInteger('nAlerts',nAlerts)
 Modelica.setInteger('state',3)
 tRed = Modelica.time()
end,

-- define function when Red becomes
    inactive
onleaveRed = function()
 Modelica.print('Leaving Red')
end,

-- define function while Red is active
runRed = function()
 result = Modelica.time() - tRed
end,

-- callback definitions for other states
     skipped

}})
-- state machine fully defined
-- main loop, executed in sync with
    Modelica
while(Modelica.sync())
do
 local value = Modelica.getReal('value')
 -- trigger events
 if nAlerts > 20 then
  fsm:panic()
 elseif value > 0.9 then
  fsm:alarm()
 elseif value > 0.5 then
  fsm:warn()
 elseif value < 0.0 then
  fsm:clear()
 elseif value < 0.5 then
  fsm:calm()
 end
 fsm:run() -- executes run of active
     state
 Modelica.setReal('result',result)
end
```

In this case the state transitions are triggered by different thresholds of the input `value`. The `fsm:run()` call executes the `run` function associated with the active state. Since the `Modelica.print()` function is called in every states' `onenter` and `onleave` functions, the state-machine behavior can be observed in the log generated by Modelica:

```
Lua[Example] @0.085: Leaving Green
Lua[Example] @0.09: Entering Yellow
Lua[Example] @0.18: Leaving Yellow
Lua[Example] @0.185: Entering Red
Lua[Example] @0.42: Leaving Red
Lua[Example] @0.425: Entering Yellow
 ...
Lua[Example] @20.18: Leaving Yellow
Lua[Example] @20.185: Entering Red
Lua[Example] @20.185: Leaving Red
Lua[Example] @20.19: Entering Black
```

Alternatively the previously described logger could be used in a similar manner.

3.2 DLR Robots Library

The DLR Robots Modelica library was the first one to use Lua to control the movements of a simulated robot (Bellmann, Seefried, and Thiele 2020). However, in the version presented then, the library was an integral part of the robot controller C code and could not be used for other purposes. With the separation of the Lua interpreter code in a stand-alone Modelica library, the Robots library had to be adapted to utilize this new generalized approach. The basic principle for controlling the robot stays the same:

A movement command, e.g. a point-to-point (PTP) request, sets a status flag (Robot Command State) and the desired target position in the virtual robot controller, the Modelica model simulates the movement, while the Lua script waits until the execution of the command is finished (Figure 6).

The major difference now is that the data is no longer stored in a special robot controller code but in the data storage of the ModelicaLua interpreter external object. Furthermore, the commands for the robot are no longer hard-coded functions defined in the compiled C interpreter but now defined in a small Lua script using the commands from Table 1. For example, the command to move a robot on a Cartesian path from its current position to the position `(x,y,z,A,B,C)` is defined like this:

```
Robots.ptpCartesianSpace =
function(x,y,z,A,B,C)
 Modelica.setReal('brl_pos_ref',
    {x,y,z,A,B,C})
 Modelica.setInteger(
    'brl_robotCommandState',
    Robots.PTPCARTESIAN)
 Robots.waitForRobot()
end
```

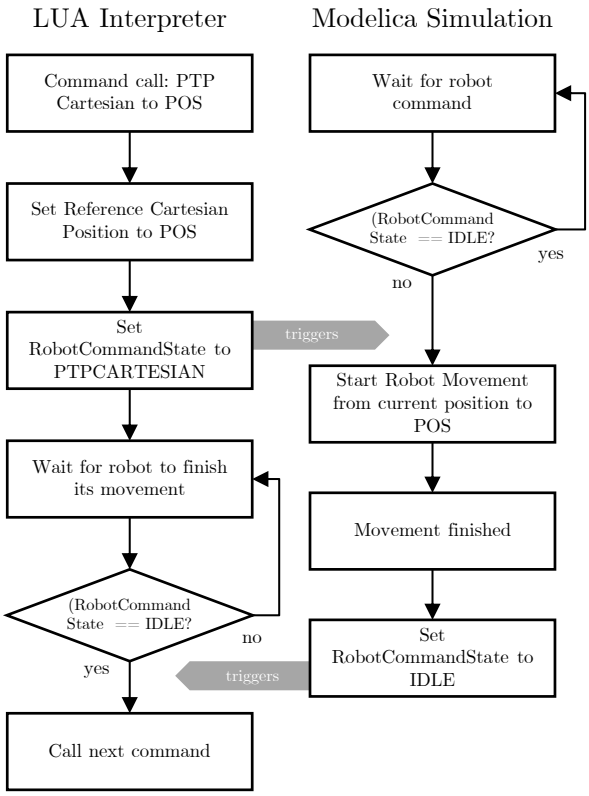

Figure 6. Principle of how the execution of movement commands is performed, from (Bellmann, Seefried, and Thiele 2020).

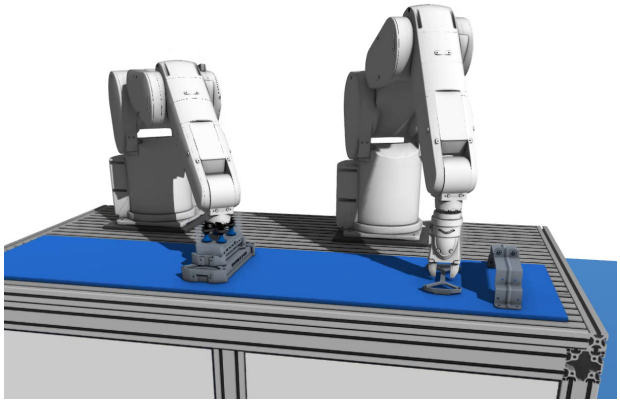

Figure 7. Two robots performing an assembly task programmed by Lua scripts, see (Reiser 2021)

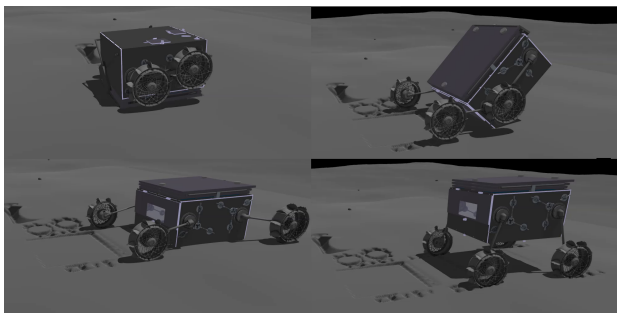

Figure 8. Visualization of the MMX up-righting sequence, shown from top left, to bottom right

This allows the user to easily integrate new commands in the robot controller script language, or modify the existing one. Figure 7, from (Reiser 2021), shows a simulation visualization of two Mitsubishi RV7-FL robots, performing an assembly task. In this use-case, two Lua interpreters run in parallel, each providing a single robot with the movement commands to perform its task. Additionally, Lua script commands are used to operate the robot tools, e.g. controlling the gripper.

3.3 MMX Rover Development

A second example for the usage of the Lua library is in the development of the MMX rover (Ulamec et al. 2019; Bertrand et al. 2019; Buse et al. 2021). This rover, jointly developed by CNES and DLR, will fly with JAXA's MMX mission to the Martian moon Phobos. There, it will be dropped onto the surface and will come to rest in a random orientation. Once it has come to rest, it has to unfold its legs in the correct order, to reorient itself on its belly and stand up. This sequence, called up-righting by the development team, must work reliably in a variety of situations. A simple example of the rover up-righting itself from its back to the belly is shown in Figure 8.

The algorithm for controlling the up-righting sequence must be developed and tested. Implementing this complex, nested state machine with the Modelica state machine would be complicated and would also require recompiling the model after each change. With the Modelica Lua library, it is possible to test different variations of the algorithm quickly. Since the script is loaded from disk, the same compiled model of the rover system can be used to launch a multitude of instances with different, random initial conditions to analyze the algorithm's robustness. Mechanical components of the rover as well as its interaction with the environment are modeled with the DLR Rover Simulation Toolkit (Hellerer, Barthelmes, and Buse 2017).

The Lua code used in this application performs multiple tasks. First, the control functions specific to the rover itself are abstracted in a separate Lua library. This enables a very direct and comprehensive approach to program the rover actions. Based on this, the high level logic of the rover is modeled as a state machine using the same approach as in the example above. The state machine controlling the rover has 36 unique states and 46 transitions in total. The `StandUp` state is used as an example here. This state becomes active once the rover successfully reoriented itself onto its belly and is now ready to stand up. This corresponds to the transition from bottom-right to bottom-left in the Figure 8. When the state becomes active, the `onStandUp` function is called once, with the interface of the rover abstracted into `Rover` a predefined angle and velocity is commanded:

```
onStandUp = function()
 Rover:setTargetLegVelocity(
     Rover.standupVelocity)
 Rover:setTargetLegAngles(
     Rover.legStandingAngles)
end
```

While the `StandUp` state is active, the `runStandUp` function is called periodically, here a timer and the rover interface are used to wait 15 seconds once the legs have reached the commanded target:

DOI 10.3384/ecp21181425

```
runStandUp = function()
 if not Rover:legsAtTarget() then
  timer:reset()
 elseif timer:elapsed() > 15 then
  uprighting:Standing()
 end
end
```

Once this condition is met, the next transition in the state machine is triggered by `uprighting:Standing()`.

Other than controlling the rover itself, the Lua code also reads initial configuration parameters and logs internal states to disk. With the results of both Modelica and the Lua scripts, a statistical analysis of the systems behavior is performed. For example, causalities between specific situations in the mechanical system and behavior of the control logic could be identified.

4 Conclusions

The presented Modelica Lua library allows easy and fast integration of high-level logic into Modelica. The simple interface makes integration into existing models easy. The library developed from the initial implementation in the DLR Robots library has been extended and generalized and has proven its usefulness especially in the development of the MMX up-righting algorithm. By providing access to existing third-party libraries in Lua, a language widely used in game development, a wide range of powerful tools is now available to be used in Modelica. It is planned to release this library with an open source license to make it available to the public.

References

Bellmann, Tobias, Andreas Seefried, and Bernhard Thiele (2020). "The DLR Robots library – Using replaceable packages to simulate various serial robots". In: *Proceedings of the Asian Modelica Conference 2020*. DOI: 10.3384/ecp2020174153.

Bertrand, Jean et al. (2019). "Roving on Phobos: Challenges of the MMX Rover for Space Robotics". In: *Proceedings of 15th Symposium on Advanced Space Technologies in Robotics and Automation*.

Bjorn (2021). *Lua vector math library*. https://github.com/bjornbytes/maf. [Online; accessed 3-May-2021].

Blockwitz, Torsten et al. (2012). "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In: *Proceedings of the 9th International Modelica Conference*. DOI: 10.3384/ecp12076173.

Buse, Fabian et al. (2021). "Wheeled locomotion in milligravity: A technology experiment for the MMX Rover (accepted)". In: *72th International Astronautical Congress*. International Astronautical Federation.

Conroy, Kyle (2021). *LUA state machine library*. https://github.com/kyleconroy/lua-state-machine. [Online; accessed 3-May-2021].

Hellerer, Matthias, Stefan Barthelmes, and Fabian Buse (2017). "The DLR Rover Simulation Toolkit". In: *Proceedings of Advanced Space Technologies in Robotics and Automation 2017*. ESA's Automation and Robotics group.

Reiser, Robert (2021). "Object Manipulation and Assembly in Modelica". In: *Proceedings of the 14th International Modelica Conference 2021*.

Ulamec, Stephan et al. (2019). "A rover for the JAXA MMX Mission to Phobos". In: *70th International Astronautical Congress*. International Astronautical Federation.