AV Stumpfl GmbH

Dokumentation

Avio Script

Autor:
David Malzner

Version 1.0

Inhaltsverzeichnis

1 Einleitung

1.1 Über Avio Scripts

Avio Script basiert auf der Skriptsprache Lua. Lua wird seit 1993 von der *Pontifical Catholic University of Rio de Janeiro* entwickelt. Hierbei handelt es sich um Open-Source-Software. Lua ist somit für externe Projekte als Bibliothek verfügbar und wird weltweit von vielen Firmen wie z.B. für die künstliche Intelligenz von Charakteren in Computerspielen oder als Erweiterungssprache für Programme eingesetzt.
Aufgrund der geringen Ressourcenanforderungen, der leicht zu erlernenden Sprache und der weiten Verbreitung haben wir uns dazu entschlossen, Lua als Basis für Avio Script zu verwenden. Bei Avio Script handelt sich um die nahtlose Integration der Skriptsprache Lua in das Avio-System.

Diese Dokument soll einen kurzen Überblick über die Verwendung von bestehenden Avio Scripts im Avio-System geben und als Anleitung dienen, selbst eigene Skripts zu schreiben.
Kapitel 3 enthält eine kurze Anleitung für die Skriptsprache Lua, die einen Überblick über die wichtigsten Konstrukte geben soll, damit man möglichst schnell beginnen kann. Damit alleine können fast alle mit Avio mitinstallierten Standardskripts verstanden werden. Für die grundsätzliche Referenz von Lua muss aber auf die Projekthomepage *www.lua.org* verwiesen werden, wo eine Dokumentation in sehr guter Qualität vorliegt.
Zielgruppe dieser Anleitung sind Personen, welche schon mal mit irgend einer Programmiersprache in Berührung gekommen sind. Erweiterte Programmierkenntnisse sind nicht nötig. Es wird vorausgesetzt, dass man bereits mit dem Avio-System vertraut ist.

2 Avio Scripts verwalten

Mit der Installation von Avio werden häufig benötigte Skripts mitinstalliert. Diese Skripts sollen alle grundlegenden Logikaufgaben abdecken können und werden nach Bedarf ständig erweitert. Die Verwaltung der Skripts geschieht über die Weboberfläche des jeweiligen Avio-Knotens.

Öffnet man die Weboberfläche eines Knotens, können auf der Seite *Logic* alle verfügbaren Skripts angezeigt werden. Im Abschnitt *Add Script* werden in einer Auswahlbox die einzelnen Skripts dargestellt. Durch Auswahl des Skriptnamens werden die Beschreibung, der Autor und die zu vergebenen Parameter des Skripts angezeigt. Im Beispiel in Abbildung 2 wurde das Skript *Add* ausgewählt. Als Parameter des Skripts können der Name des Ports, unter dem das Skript installiert werden soll, angegeben werden und der Standardwert, der zu einem Wert addiert wird. Bei jedem zu installierenden Skript kann man zusätzlich den Port und den Slot angeben. Der Parameter Port entspricht der Adresse des erstellten Ports des Skripts, über welches er später in Avio adressiert wird. Dieser Parameter wird automatisch vergeben und muss nicht geändert werden, wenn man nicht bewusst eine bestimmte Adresse verwenden will.
Beim Parameter Slot handelt es sich um den Slot, in dem das Skript installiert wird. Es stehen 10 Slots zur Verfügung und es können beliebig viele Skripts in einem Slot installiert werden. Es kann immer nur 1 Skript pro Slot aktiv laufen. Bei Skripts mit reinen Logikfunktionen, die keine Zeitverzögerung eingebaut haben (wie z.B. Add, And, Compare...), spielt der installierte Slot keine Rolle. Bei Skripts mit Zeitverzögerung jedoch (wie Delay, Ramp,...) blockieren alle anderen Skripts im gleichen Slot, solange das laufende Skript nicht fertig abgearbeitet wurde.

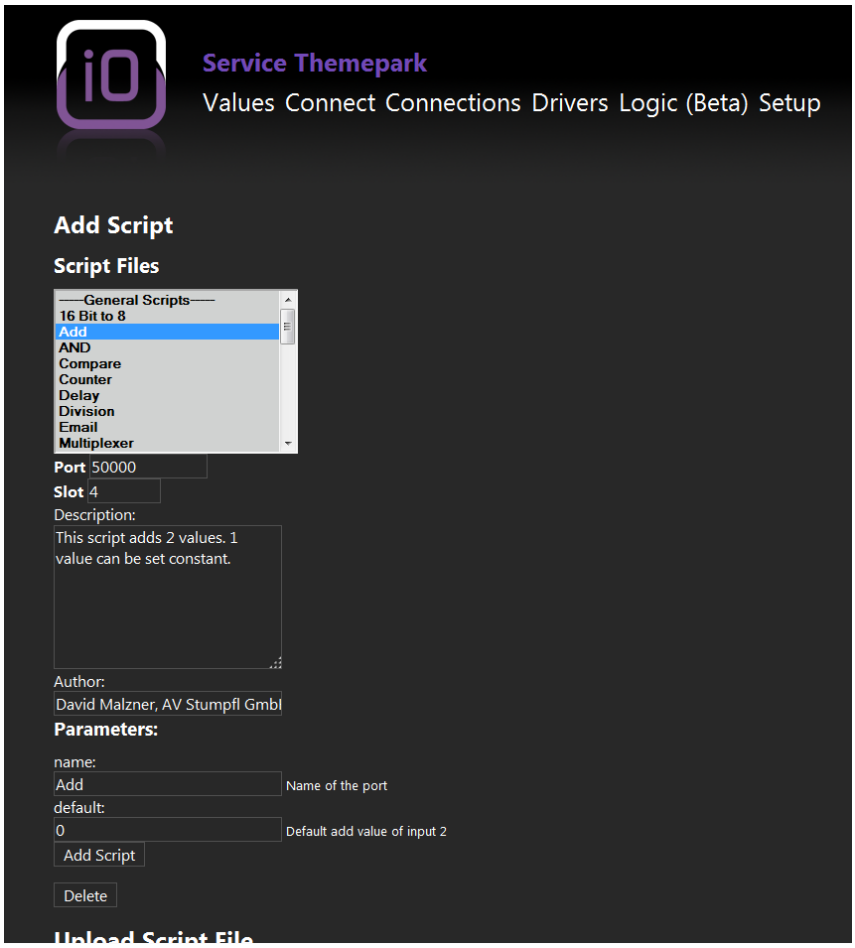

Abbildung 1: Anzeige der verfügbaren Skripts

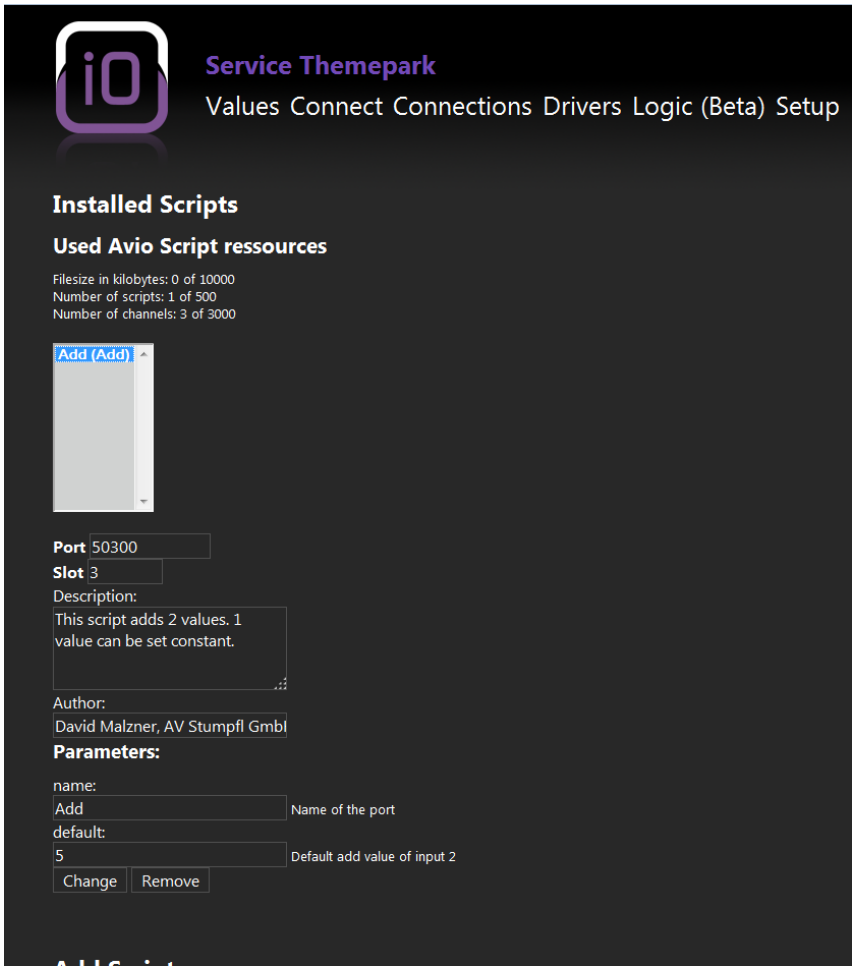

Abbildung 2: Anzeige der gestarteten Skripts

Bereits gestartete Skripts könnnen im Abschnitt *Installed Scripts* betrachtet und die vergebenen Parameter nachträglich geändert werden. Die Änderungen werden sofort im Avio System übernommen und im Avio Manager angezeigt (siehe Abbildung 2).

In Abbildung 3 ist das installierte Skript *Add* im Avio Manager zu sehen.

3 Kurzanleitung der Programmiersprache Lua

Dieses Kapitel enthält einen kurzen Überblick über die Programmiersprache Lua. Es werden die wichtigsten Elemente beschrieben, um möglichst schnell eigene Skripts entwickeln zu können. Obwohl Lua relativ einfach zu erlernen ist, sind auch sehr komplexe Konstrukte damit möglich. Für eine komplette Dokumentation der Sprache wird auf die Projekthomepage *www.lua.org* verwiesen. Da es sich bei Lua um eine eigene, externe Programmiersprache handelt, ist der Anwender für die Verwendung selbst verantwortlich, d.h. AV Stumpfl kann nur einen sehr beschränkten Support dafür leisten. Die wichtigsten Elemente wie Funktionen, Variablen, Bedingungen und Schleifen sind schnell erlernt. Damit alleine kann man schon die meisten Logikaufgaben mit Avio Script lösen.
Avio Script basiert auf der Lua Version 5.1.

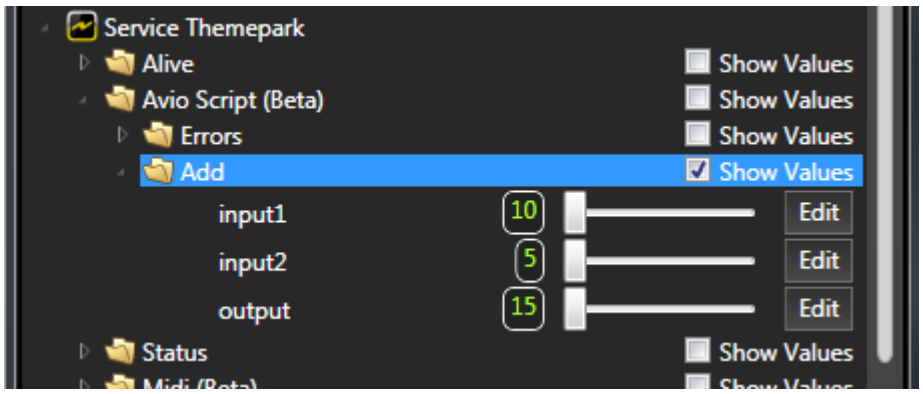

Abbildung 3: Anzeige Skript Add im Avio Manager

Kommentare können in Lua mit `--` eingeleitet werden und gelten für jeweils eine Zeile.

3.1 Variablen

Wie in den meisten Skriptsprachen üblich ist Lua eine dynamisch typisierte Sprache, d.h. der Typ einer Variable braucht vor der Zuweisung nicht angegeben werden, dies erfolgt zur Laufzeit. Vor der Zuweisung eines Wertes besitzt die Variable den Wert `nil` .
Variablen können die booleschen Werte `true` und `false` , Zahlen, Zeichenketten (werden zwischen " " deklariert), Tables, Funktionen, Userdata und Threads zugewiesen werden. In dieser Kurzanleitung werden wir uns auf die für Avio Script wichtigsten Typen, nämlich auf boolesche Werte, Zahlen, Zeichenketten und Tables beschränken. Jede Variable ist global verfügbar, auch wenn sie innerhalb einer Funktion definiert wird. Will man die Variable nur lokal verwenden, wird vor der Zuweisung eines Wertes der Variable das Schlüsselwort `local` verwendet.
Beispiel:

Listing 1: Beispiel Deklaration und Verwendung von Variablen

```
1 x=5;
2 y="Hello from Lua";
3 z=true;
4 print(x);                          --Output: 5
5 print(y);                          --Output: "Hello from Lua"
6 print(z);                          --Output: true
7 print(a);                          --Output: nil
```

3.2 Tables

Tables sind ein wichtiges Konzept von Lua und sehr leistungsfähig. In dieser Kurzanleitung wird darauf eingegangen, wie man sie als Datenfelder (Arrays) verwenden oder darin Schlüssel-Wert-Paare speichern kann.
Eine leere Tabelle wird mit dem Wert { } definiert, also zum Beispiel `t = {}` . Will man die Tabelle als Datenfeld (Array) verwenden, werden die einzelnen Indizes in eckiger Klammer nach dem Tabellennamen ausgewählt, also zum Beispiel `t[1] = 3` . Zu beachten ist, dass der erste Eintrag bei Index 1 beginnt, nicht bei 0 (es wird also der n-te Eintrag angegeben, nicht wie bei manch anderen Programmiersprachen der Offset zur Speicheradresse des Datenfeldes). Es können beliebige Indizes ausgewählt werden, es brauchen auch nicht alle Indizes vergeben werden. Nicht vergebene Indizes erhalten den Wert `nil` . Jeder Index kann einen beliebigen

Datentyp oder eine weitere Table enthalten. Es ist also nicht nötig, dass alle Einträge der Table den gleichen Datentyp besitzen.
Die Länge einer Table kann mit der Funktion `table.getn(t)` ermittelt werden.
Beispiel:

Listing 2: Beispiel Deklaration und Verwendung von Tabellen als Datenfeld

```
1  t = {};
2  t[1] = 5;
3  t[2] = 6.5;
4  t[3] = "Hello there";
5  t[5] = 7;
6  print(t[1]);                          --Output: 5
7  print(t[2]);                          --Output: 6.5
8  print(t[3]);                          --Output: "Hello there"
9  print(t[4]);                          --Output: nil
10 print(t[5]);                          --Output: 7
11 print(table.getn(t))        --Output: 3 (because value 4 is nil, rest of table is not
       counted)
```

Eine weitere Verwendung von Tables in Lua ist die Adressierung von Werten über Schlüssel. Statt dem Index wird der Name des Schlüssels der Tabelle angegeben, also z.B. `t["a"] = 5` . Der Schlüssel „a“ kann auch genauso folgendermaßen deklariert werden: `t.a = 5` .

Beispiel:

Listing 3: Beispiel Deklaration und Verwendung von Tabellen als Schlüssel-Wert-Paar

```
1 t = {};
2 t["a"] = 5;
3 t["b"] = 6;
4 t.c = 7;
5 print(t["a"]);                  --Output: 5
6 print(t["b"]);                  --Output: 6
7 print(t["c"]);                  --Output: 7
```

3.3 Funktionen

Funktionen werden in Lua mit dem Schlüsselwort `function` eingeleitet. Danach folgt der Name der Funktion gefolgt von den Eingangs-Parametern in Klammer, also z.B. `function foo(input1,input2)` . Beendet werden Funktionen mit `end` . Optional können innerhalb von Funktionen Werte mit dem Schlüsselwort `return` zurückgegeben werden.

Beispiel:

Listing 4: Beispiel Funktion

```
1 function add(input1,input2)
2     res = input1+input2;
3     return res;
4 end
5 print(add(7,8));                  --Output: 15
```

Funktionen können auch mit einer variablen Anzahl von Parametern definiert werden. Dies geschieht mit der Angabe `...` als Parameter, also z.B. `function foo(...)` . Über die einzelnen variablen Parameter kann anschließend über eine *for*-Schleife (siehe Kapitel 3.5) iteriert werden (siehe Listing 5, Zeile 3). Weiters können auch mehrere Argumente als Ergebnis zurückgegeben werden, indem die einzelnen Werte nach dem *return*-Schlüsselwort mittels „,“ getrennt werden.

Listing 5: Beispiel Funktion mit variable Anzahl von Parametern und mehreren Rückgabewerten

```
function foo(a,...)
    print(a);
    for i,v in ipairs(arg) do
        print(v)
    end
    return a,arg[1];
end
ret1, ret2 = foo(5,10,11,12,13);            --Output: 5 10 11 12 13
print(ret1)                                                 --Output: 5
print(ret2)                                                 --Output: 10
```

3.4 If-then-else Bedingungen

If-Statements werten eine Bedingung aus und führen bei Erfüllung der Bedingung den dazugehörigen *then*-Teil aus, ansonsten den *else*-Teil. Der *else*-Teil ist optional. Der Block wird mit einem *end* beendet. Folgende Vergleiche sind in einem *if*-Statement möglich:

- == für die Prüfung auf gleich
- ~= für die Prüfung auf ungleich
- < für die Prüfung auf kleiner
- <= für die Prüfung auf kleiner-gleich
- > für die Prüfung auf größer
- >= für die Prüfung auf größer-gleich
- *and* prüft ob die linke *und* rechte Bedingung erfüllt ist
- *or* prüft ob die linke *oder* rechte Bedingung erfüllt ist

Beispiel:

Listing 6: Beispiel if-then-else Bedingung

```
a=5;
if a==5 then
    print("condition␣met")       --will be executed
end

a=6;
if a==7 then
    print("condition␣1␣met")     --will not be executed
elseif a==8 then
    print("condition␣2␣met")     --will not be executed
else
    print("no␣condition␣met")    --will be executed
end

a=8
if a~=8 then
    print("condition␣met")       --will not be executed
else
    print("condition␣not␣met")   --will be executed
end

a=9
b=10
if a == 9 and b > 9 then
```

```
25      print("condition␣met")       --will be executed
26  end
27
28  a=11;
29  if(a == 15 or a <= 11) then
30      print("condition␣met")       --will be executed
31  end
```

3.5 Schleifen

Unter Schleifen versteht man Bereiche, die mehrmals hintereinander ausgeführt werden können, wie z.B. das Hochzählen von 1 auf 100. Im Folgenden werden die Schleifenarten *for* und *while* behandelt.

3.5.1 for-Schleife

In Lua gibt es 2 Arten von *for*-Schleifen, nämlich numerische und generische Schleifen.
Bei numerischen Schleifen wird ein Block n-mal ausgeführt. Die Anzahl der Schleifendurchgänge ist durch eine Laufvariable definiert, die von einem Startwert zu einem Endwert bei einem definierbaren Inkrement iteriert (hoch- oder heruntergezählt).
Eine numerische *for*-Schleife hat folgenden Syntax:

Listing 7: Syntax numerische *for*-Schleife

```
1  for var=startValue,endValue,incValue do
2      something
3  end
```

Der Variable *var* wird der Startwert *startValue* zugewiesen. Die Variable wird in jedem Schleifendurchgang, in dem *something* aufgerufen wird mit dem Wert *incValue* inkrementiert, bis der Endwert *endValue* erreicht wurde. Der Inkrementwert *incValue* ist optional. Wird er nicht angegeben, wird der Wert 1 angenommen.

Beispiel:

Listing 8: Beispiel numerische *for*-Schleife

```
1  for i=2,6,2 do       --Output: 2 4 6
2      print(i)
3  end
4
5  for i=1,5 do         --Output: 1 2 3 4 5
6      print(i)
7  end
```

Bei generischen Schleifen ist es nicht nötig den Wertebereich, über den iteriert werden soll, anzugeben. Sie erlaubt es, alle Elemente zu durchlaufen, die von einer Iterator-Funktion zurückgegeben werden.

Listing 9: Syntax generische for Schleife

```
1  for i,v in f(x) do
2      something
3  end
```

Die Variable i bekommt den Wert des n-ten Schleifendurchgangs. Die Variable v enthält jeweils den Wert des aktuellen Elementes. Die Funktion *f(x)* repräsentiert die Iterator Funktion, welche alle zu durchlaufenden Werte zurückgibt.

Beispiel:

Listing 10: Beispiel generische *for*-Schleife

```
1 t = {4,5,6,7,8}
2 for i,v in ipairs(t) do      --Output: 4,5,6,7,8
3     print(v)
4 end
```

In Listing 10 werden die Werte der Tabelle t ausgegeben. Die in Lua bereits vorhandene Funktion *ipairs* dient dabei als Iterator-Funktion, welche alle Werte der Tabelle t zurückgibt.

3.5.2 while-Schleife

Eine *while* Schleife führt einen Block solange aus, solange eine definierbare Bedingung erfüllt ist. Sie wird mit dem Schlüsselwort `while` eingeleitet, der iterierende Block mit `do` gestartet und mit `end` beendet.

Beispiel:

Listing 11: Beispiel while Schleife

```
1 a=10
2 while a>1 do         --Output: 10 5 2.5 1.25
3     print(a)
4     a = a/2;
5 end
```

Das Beispiel in Listing 11 zeigt eine *while* Schleife, die solange ausgeführt wird, solange die Bedingung *a>1* erfüllt ist. In jedem Schleifendurchgang wird die Variable a halbiert, die Schleife wird also beim Wert 0.75 beendet.

3.6 Weiterführende Informationen

- Komplette Dokumentation der Skriptsprache Lua in der Version 5.1: `http://www.lua.org/manual/5.1/`
- Dokumentation der Bibliothek *LuaXML*: `http://viremo.eludi.net/LuaXML/`
- Dokumentation der Bibliothek *LuaSocket* : `http://w3.impa.br/~diego/software/luasocket/introduction.html`

4 Avio Scripts erstellen

4.1 Lua Development Tools

Bei den *Lua Development Tools* [1] handelt es sich um eine Entwicklungsumgebung für die Skriptsprache Lua. *Lua Development Tools* basiert auf der Entwicklungsumgebung *Eclipse* [2]. *Eclipse* wurde ursprünglich als Entwicklungsumgebung für die Programmiersprache *Java* erstellt und ist auch selbst in dieser Programmiersprache geschrieben. Mittlerweile gibt es Versionen und Plugins für viele Programmiersprachen. Bei *Eclipse* handelt es sich um Open-Source-Software, d.h. der Quellcode ist frei verfügbar. Weiters ist *Eclipse* unter der sehr liberalen *Eclipse Public*

[1] http://www.eclipse.org/koneki/ldt/
[2] http://www.eclipse.org/

License [3] veröffentlicht, die es erlaubt Anpassungen zu machen und diese wieder zu verbreiten. *Lua Development Tools* basiert auf einem Open-Source-Projekt welches die Entwicklungsumgebung *Eclipse* für die Entwicklung und das Debuggen von Lua Skripts angepasst und erweitert hat.
Aufgrund der Ausgereiftheit und des Komforts dieser Entwicklungsumgebung haben wir uns dazu entschlossen, diese als Standardwerkzeug für die Entwicklung und das Debuggen von *Avio Scripts* mitzuliefern. Wir haben diese Entwicklungsumgebung soweit in das Avio System integriert, dass darin geschriebene Skripts komfortabel auf Avio-Knoten übertragen werden können und dass damit das Debuggen von Skripts direkt auf den Avio-Knoten und „offline“ in der Entwicklungsumgebung funktioniert.

4.2 Erstellen von Skripts mit den „Lua Development Tools“

4.2.1 Neues Lua Skript hinzufügen

Nach dem Start der *Lua Development Tools* ist auf der linken Seite die Baumansicht mit den im Projekt verfügbaren Skripts ersichtlich. Um eine Datei hinzuzufügen, muss im Kontextmenü, welches nach einem Rechtsklick auf *LuaAvio* erscheint, *New - Lua File* ausgewählt werden. Im nun erscheinenden Dialog muss der Dateiname mit der Erweiterung *.lua* angegeben werden. Dies ist in Abbildung 4 zu sehen. Weiters kann eine Datei in das Projekt hinzugefügt werden, in dem ein bestehendes Skript per Drag&Drop in den in den Ordner *src* des Baums gezogen wird.

Alle Skripts, die dem Projekt hinzugefügt wurden, sind auch in der Weboberfläche des lokal installierten *Services* wie alle anderen Skripts ersichtlich. Auf der Seite *Logic* können also unter dem Abschnitt *Add Script* alle Skripts des Projekts in der Entwicklungsumgebung installiert werden (siehe Abbildung 5).
Wird nun ein installiertes Skript geändert (egal in welchem Editor), kommt automatisch eine Benachrichtigung vom Avio Service, dass auf die Änderung hinweist. Per Mausklick können auf Wunsch die Änderungen sofort übernommen werden, um die Modifikationen im Skript zu testen.

4.3 Hello World

Wie in den meisten Dokumentationen einer Programmiersprache üblich, auch hier zuerst ein „Hello World“-Programm, welches das nötige Grundgerüst zeigt, um „Hello World“ auszugeben. In der Programmiersprache Lua alleine wäre das mit der Zeile `print("Hello World")` erledigt. Da wir in unserem Avio System aber über die Channels eines Avio-Knoten kommunizieren, fällt das „Hello World“-Programm für Avio etwas komplexer aus.
Dieses minimale Skript ist in Listing 12 abgebildet.

Listing 12: HelloWorld

```
1 require("avio")
2 function init()
3     avio.addPort("Hello Port","Description of port. This is displayed in Wings Avio
          Manager","string");
4     avio.addChannel("Hello Port","Hello Channel");
5     avio.setChannel("Hello Channel","Hello World")
6 end
```

Wie bereits in Kapitel 4.2.1 beschrieben, werden neu erstellte Skripts sofort in der Weboberfläche bei den verfügbaren Skripts angezeigt. Wird das Skript nun installiert, wird ein Port mit

[3] http://www.eclipse.org/legal/epl-v10.html

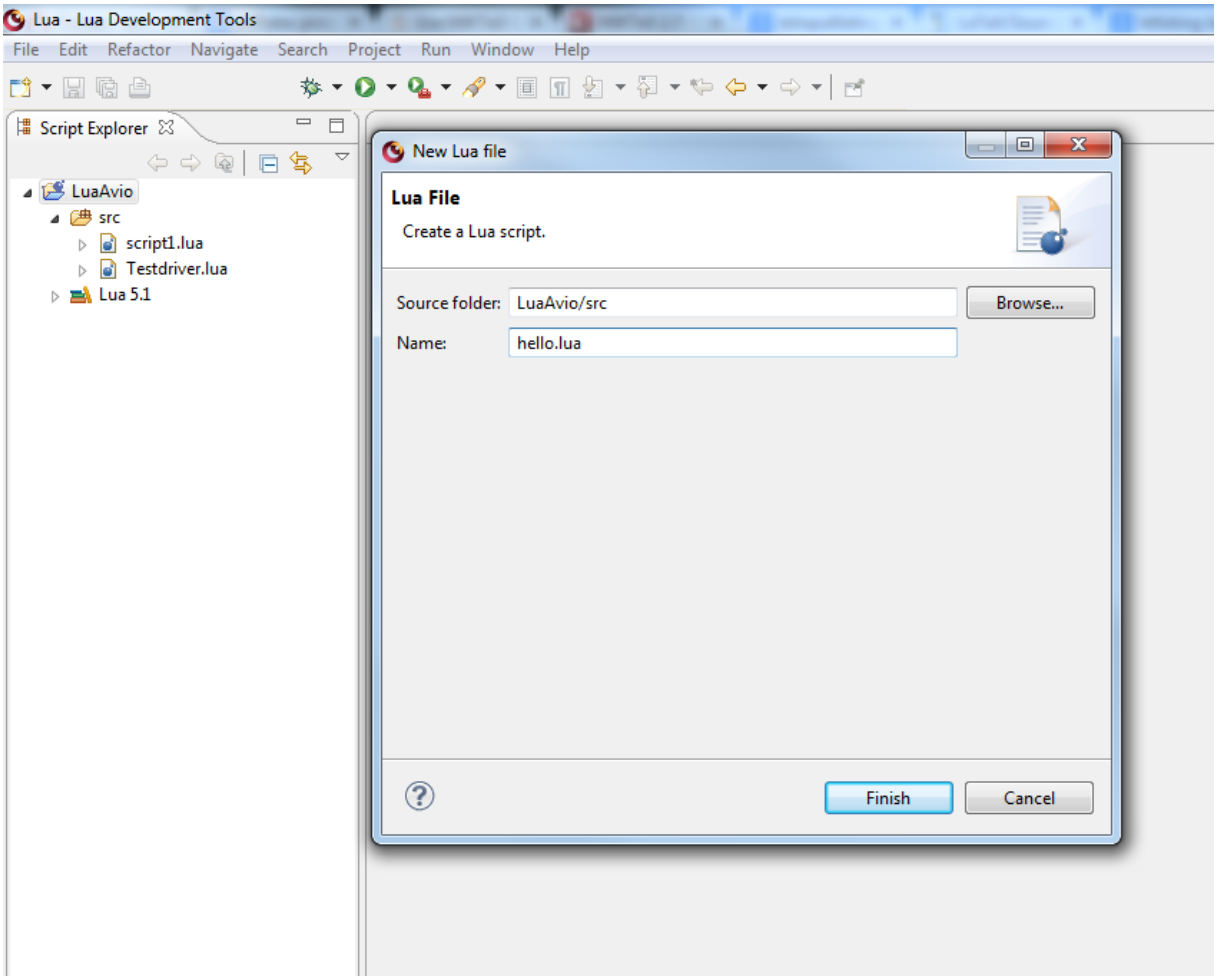

Abbildung 4: Neues Lua Skript hinzufügen

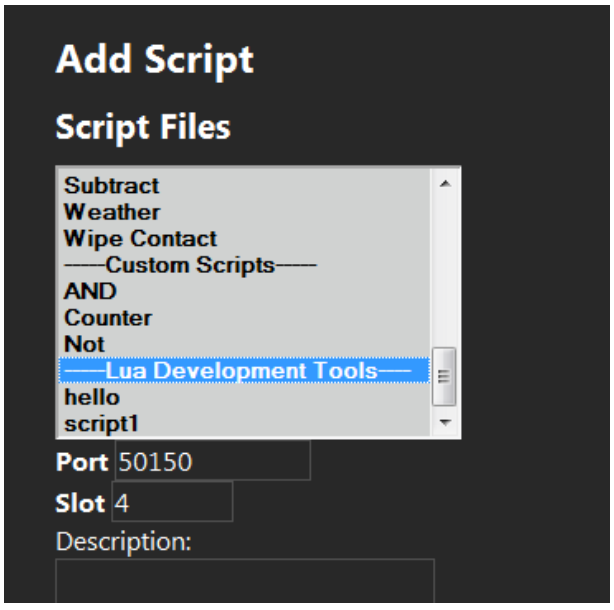

Abbildung 5: Neu hinzugefügtes Skript ist über die Weboberfläche verfügbar

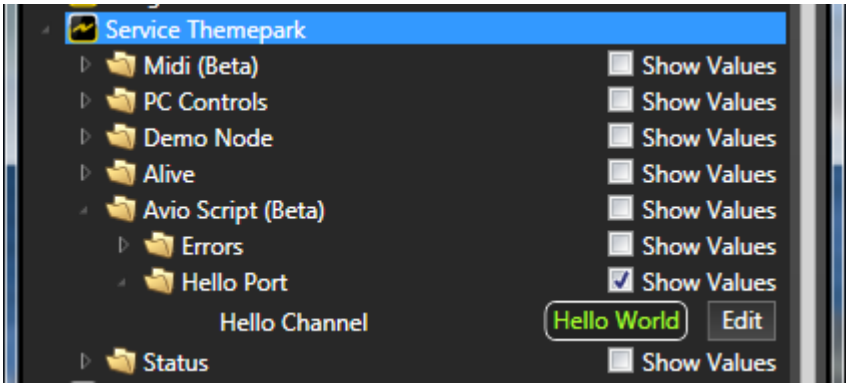

Abbildung 6: Die Ausgabe des Hello World Skripts im Avio Manager

dem Namen „Hello Port“ angelegt. Es wird mit dem Parameter `string` angegeben, dass der Datentyp in den Channels des Ports Text ist. Will man Zahlen übertragen (Datentyp Integer) lässt man diesen Parameter einfach weg (siehe Listing 12, Zeile 3);
In Zeile 4 wird dem Port mit dem Namen „Hello Port“ der Channel mit dem Namen „Hello Channel“ hinzugefügt. In Zeile 5 wird dem Channel mit dem Namen „Hello Channel“ der Wert „Hello World“ zugewiesen. Diese Ausgabe ist nun im Avio Manager ersichtlich, wenn das Skript über die Weboberfläche installiert wird (siehe Abbildung 6).

4.4 Die Bibliothek Avio

Die Skriptsprache Lua lässt sich neben den Standardbibliotheken um weitere Bibliotheken erweitern. Bei einer Bibliothek handelt es sich um eine Zusammenfassung von Funktionen. Die Avio Bibliothek enthält alle Funktionen, die nötig sind, um mit dem Avio-System zu interagieren. Das Einbinden der Avio-Bibliothek geschieht mit dem Befehl `require("avio")` .
In der nachfolgenden Dokumentation werden optionale Parameter in eckigen Klammern [ ] dargestellt. Textparameter werden mit dem Präfix *#string* angegeben, numerische Parameter mit dem Präfix *#number*.

4.4.1 addPort

```
avio.addPort(name,description,[datatype])
```

Dieser Befehl fügt einen neuen Port zum Avio-Knoten hinzu. Bei einem Port handelt es sich um eine Gruppe von Channels. Jedes Avio Script benötigt mindestens einen Port, in dem später die Channels hinzugefügt werden, welche nach außen verfügbar sind.

Parameter:

- *#string name*: Der Name des Ports
- *#string description*: Die Beschreibung der Funktion des Ports. Diese Beschreibung wird im *Avio Manager* angezeigt.
- *#string datatype (optional)*: Dieser Parameter gibt den Datentyp der später hinzugefügten Channels dieses Ports an. Wird der Wert *string* übergeben, sind alle enthaltenen Channels Textchannels, über die Zeichenketten gelesen und geschrieben werden können. Wird der

Parameter weggelassen, können über alle enthaltenen Channels numerische Werte gelesen und geschrieben werden.

Rückgabewerte: keine

Beispiel:

```
avio.addPort("Add","This␣port␣adds␣2␣values␣and␣stores␣the␣result␣in␣the␣output␣channel")
```

4.4.2 addChannel

```
avio.addChannel(port,channel,[maxValue])
```

Dieser Befehl fügt einen Channel zu einem bereits erstellten Port hinzu.

Parameter:

- *#string port*: Der Name des Ports, zu dem der Channel hinzugefügt werden soll. Es können nur Ports angegeben werden, die in diesem Skript erstellt wurden.
- *#string channel*: Der Name des neuen Channels
- *#number maxValue (optional)*: Gibt den maximalen Wert des Channels an. Dies ist wichtig, damit im Avio das automatische Skalieren beim Verbinden von Channels mit unterschiedlichem Wertebereich funktioniert. Wird der Parameter nicht angegeben, wird als maximaler Wert automatisch der maximale Wert einer vorzeichenbehafteten 32-Bit Ganzzahl angegeben ($2^{31} - 1$ oder 2.147.483.648). Dies ist auch die größte in Avio verfügbare vorzeihenbehaftete Zahl.

Rückgabewerte: keine

Beispiel:

```
avio.addChannel("Add","input1")
```

4.4.3 setFunction

```
avio.setFunction(functionName,channel,[channel1],[...])
```

Dieser Befehl definiert eine Funktion die aufgerufen wird, wenn sich der Wert eines Channels ändert. Die übergebene Funktion muss so viele Aufrufparameter enthalten wie Channels übergeben werden, bei deren Änderung die Funktion aufgerufen wird.

Parameter:

- *#string function*: Der Name der Funktion, die aufgerufen werden soll, wenn sich einer der nachfolgend übergebenen Channels ändert.
- *#string channel1...channelN*: Beliebig viele Channels, bei deren Wertänderung die vorhergehende Funktion aufgerufen werden soll

Rückgabewerte: keine

Beispiel:

Listing 13: Beispiel SetFunction

```
avio.setFunction("addFunc","input1","input2")
function addFunc(input1,input2)
    res = input1+input2;
end
```

4.4.4 setChannel

```
avio.setChannel(channel,value)
```

Dieser Befehl weist einem Channel einen Wert zu. Es können nur Channels verwendet werden, die im selben Skript angelegt wurden. Wurde der gleiche Name in einem anderen Skript ebenfalls angelegt, wird also der Channel aus dem gleichen Skript verwendet.

Parameter:

- *#string channel*: Der Name des Channels, dessen Wert gesetzt werden soll
- *#string, #number channel*: Der Wert, den der Channel bekommen soll.

Rückgabewerte:

- –

Beispiel:

```
avio.setChannel("output",5)
avio.setChannel("output","hello world")
```

4.4.5 getChannel

```
res = avio.getChannel(channel)
```

Dieser Befehl liest den Wert eines Channels aus. Auch hier wird analog zu Kapitel 4.4.4 der Name eines Channels verwendet, der im selben Skript angelegt wurde.

Parameter:

- *#string channel*: Der Name des Channels, dessen Wert gelesen werden soll

Rückgabewerte:

- *#number res*: Der Wert des ausgelesenen Channels

Beispiel:

```
res = avio.getChannel("input1")
```

4.4.6 sleep

```
avio.sleep(time)
```

Dieser Befehl unterbricht das ausgeführte Skript für eine bestimmte Zeitspanne. Während dieser Zeit können keine Funktionen durch das Ändern von Werten eines Channels (siehe Kapitel 4.4.3) im gleichen Skript oder in einem anderen Skript, dass sich im gleichen Slot befindet getriggert werden.

Parameter:

- *#number time*: Zeit in Millisekunden, die die Ausführung verzögert werden soll.

Rückgabewerte:

- –

Beispiel:

```
avio.sleep(2000)--Pauses execution of script for 2000 ms (=2 s)
```

4.4.7 setPeriodicFunction

```
avio.setPeriodicFunction(functionname,time)
```

Dieser Befehl dient dazu, eine vorhandene Funktion innerhalb des Skripts in einem definierbaren Intervall periodisch aufzurufen.

Parameter:

- *#string function*: Der Name der Funktion, welche periodisch aufgerufen werden soll
- *#number time*: Zeitspanne in Millisekunden, in der der die Funktion periodisch aufgerufen werden soll.

Rückgabewerte:

- –

Beispiel:

```
avio.setPeriodicFunction("calc",100);
```

4.5 Grundgerüst eines Avio Scripts

Jedes Avio Script benötigt eine Funktion mit dem Namen *init*. Diese Funktion wird vom jeweiligen Avio-Knoten beim Starten des Skripts aufgerufen. Diese Funktion initialisiert das Skript und erstellt die Ports und Channels, mit denen das Skript mit dem Avio-System kommunizieren kann (siehe Hello World-Listing 4.3 als Minimalbeispiel).
Wie in Kapitel 2 beschrieben, können einzelne Skripts parametrisiert werden und die Beschreibung in der Weboberfläche des Avio-Knotens angezeigt werden. Die Informationen über das Skript und die Parametrisierung werden in der Datei des Skripts zu Beginn angegeben. Die Beschreibung erfolgt im Xml-Format, sie wird später beim Hinzufügen und beim Ändern des Skripts vom jeweiligen Avio-Knoten interpretiert. Da es sich bei dieser Beschreibung nicht um Lua-Code handelt, wird sie für Lua mit `--` auskommentiert.

Die Beschreibung des Skripts kommt in den Tag `<summary>` , der Name des Skripts in den Tag `<name>` .
Einzelne Parameter werden mit dem Tag `<param>` definiert. Jeder Parameter-Tag enthält als Argument einen Namen (Argumentname *name*) und einen Standardwert (Argumentname *default*), mit dem der Parameter gesetzt wird. Der Param-Tag enthält die Beschreibung des Parameters. Es ist sinnvoll, dass jedes Skript zumindest einen Parameter mit dem Namen des Ports enthält, welcher beim Initialisieren erstellt wird.
Beim Aufruf der Init-Funktion werden die im Header definierten Parameter übergeben. Es ist wichtig, dass die Init-Funktion genauso viele Parameter enthält, wie in der *Xml* Beschreibung des Skripts angegeben wurden. Die Übergabe der Parameter an die Init-Funktion erfolgt in der Reihenfolge, in der sie definiert wurden. Der Name des Parameters ist unerheblich. So können z.B. besser lesbare Parameternamen in der Konfiguration verwendet werden, ohne durch die Limitierungen eines Parameternamens des Skripts beeinträchtigt zu werden. Beispiel Parametername Beschreibung: „Number of Loops“, Beispiel dazugehöriger Parametername der Init-Funktion: „nrOfLoops“. Leerzeichen wären als Variablennamen in Lua nicht möglich.

In Listing 14 ist als Beispiel die Beschreibung des Skripts „Add“ abgebildet.

Listing 14: Die *Xml* Beschreibung des Skripts Add am Anfang des Skripts

```
1 -- <summary>
2 --    This script adds 2 values. 1 value can be set constant.
3 -- </summary>
4 -- <name>Add</name>
5 -- <param name="name" default="Add">Name of the port</param>
6 -- <param name="default" default="0">Default add value of input 2</param>
7 -- <author>David Malzner, AV Stumpfl GmbH</author>
```

5 Debuggen von Avio Scripts in den „Lua Development Tools“

Die „Lua Development Tools“ enthalten neben einem Editor auch einen leistungsfähigen Debugger. Damit ist es möglich, ein selbst geschriebenes Skript Zeile für Zeile durchzugehen und die Variablenwerte zu beobachten. Für Avio Script werden zwei Arten des Debuggens unterschieden. Einerseits das „Offline“ Debuggen, bei dem das Avio Script nur innerhalb der Entwicklungsumgebung ausgeführt wird. Um das Skript realitätsnah zu testen, wurden alle Lua-Bibliotheken, inklusive der Avio-Bibliothek (siehe 4.4) dem Lua-Interpreter der Entwicklungsumgebung hinzugefügt. Da es sich bei der Entwicklungsumgebung selbst nicht um einen Avio-Knoten handelt, kann das Skript beim „Offline“ debuggen nicht mit dem Avio-System kommunizieren und ist nach dem Ausführen daher auch nicht im Avio Manager sichtbar. Wird nun eine Funktion aus der Avio-Bibliothek aufgerufen, wird per Textausgabe in das Konsolenfenster der Entwicklungsumgebung darauf hingewiesen, was geschehen würde. Der Vorteil dieser Methode ist, dass Änderungen sofort übernommen werden können, bevor sie auf den Avio-Knoten übertragen werden müssen und sofort wieder getestet werden können.

5.1 „Offline“ ohne Avio-Knoten

5.1.1 Erstellen des Testtreibers

Nach dem Start der „Lua Development Tools“ befinden sich bereits zwei Dateien im Workspace. Eine Datei mit dem Namen „script1.lua“, welches ein Beispielskript zeigt und eine Datei mit dem Namen „Testdriver.lua“, welches dazu dient andere Skripts zu testen.

In unserem Beispiel werden wir das bereits mitgelieferte Skript „Add“ debuggen. Wir ziehen die Datei *Add.lua* aus dem Skript Verzeichnis *C:\ProgramData\AV Stumpfl\Scripts* per Drag&Drop in den Workspace (wie in Kapitel 4.2.1 beschrieben).
Nun wollen wir das Skript debuggen. Wir werden die gleichen Funktionen aufrufen, die Avio im Skript aufrufen wird. Diese Funktionsaufrufe werden wir in der Datei „Testdriver.lua“ erstellen. Der Dateiname ist beispielhaft gewählt, der Name könnte beliebig sein.
In der 1. Zeile des Testtreibers werden wir die Datei *Add.lua* sichtbar machen. Dies geschieht mit der Zeile `require("Add")` (siehe Listing 16, Zeile 2).
Beim Laden des Skripts wird Avio die Funktion *init* mit den dazugehörigen Parametern aufrufen. In unserem Testdriver passiert das in Zeile 3. Als Parameter werden der Name „Add“ und der Standardwert „5“ für die Addition angegeben. Im Avio System kann dies in der Weboberfläche konfiguriert werden (siehe Kapitel 2).
Als nächstes wird im Testdriver die Funktion *add* aufgerufen. Wie in Listing 15, Zeile 14 definiert, wird diese Funktion von Avio aufgerufen, wenn sich einer der Eingabewerte verändert.

Listing 15: Das Skript add das getestet werden soll

```
1  -- <summary>
2  --    This script adds 2 values. 1 value can be set constant.
3  -- </summary>
4  -- <name>Add</name>
5  -- <param name="name" default="Add">Name of the port</param>
6  -- <param name="default" default="0">Default add value of input 2</param>
7  -- <author>David Malzner, AV Stumpfl GmbH</author>
8  require("avio")
9  function init(name,default)
10     avio.addPort(name,"This port offers an add operation");
11     avio.addChannel(name,"input1");
12     avio.addChannel(name,"input2");
13     avio.addChannel(name,"output");
14     avio.setFunction("add","input1","input2");
15     avio.setChannel("input2",default);
16     print("initialized script add");
17 end
18
19 function add(input1,input2)
20     res = input1+input2;
21     avio.setChannel("output",res);
22 end
```

Listing 16: Der Testtreiber welcher die zu testenden Funktionen im Skript add aufruft

```
1  --Testdriver for add
2  require("Add")
3  init("Add",5);
4  add(5,6);
```

5.1.2 Testtreiber debuggen

Da nun ein Testtreiber erstellt ist, welcher die Funktionen des Skripts auf gleiche Weise aufruft wie später das Avio System, kann das Skript jetzt debuggt werden. Damit der Debugger später in der ersten Zeile stehenbleibt, wird hier ein Breakpoint durch Doppelklick außerhalb des Textbereiches in der 1. Zeile erstellt. Zum Debuggen wird mit der rechten Maustaste auf die Datei „Testdriver.lua“ geklickt und „Debug As - Lua Application“ ausgewählt (siehe Abbildung 7).
Danach kommt eine Abfrage, ob in die Debugperspektive des Editors gewechselt werden soll. Dieser Dialog soll positiv beantwortet werden. Die Entwicklungsumgebung befindet sich nun im Debugmodus.

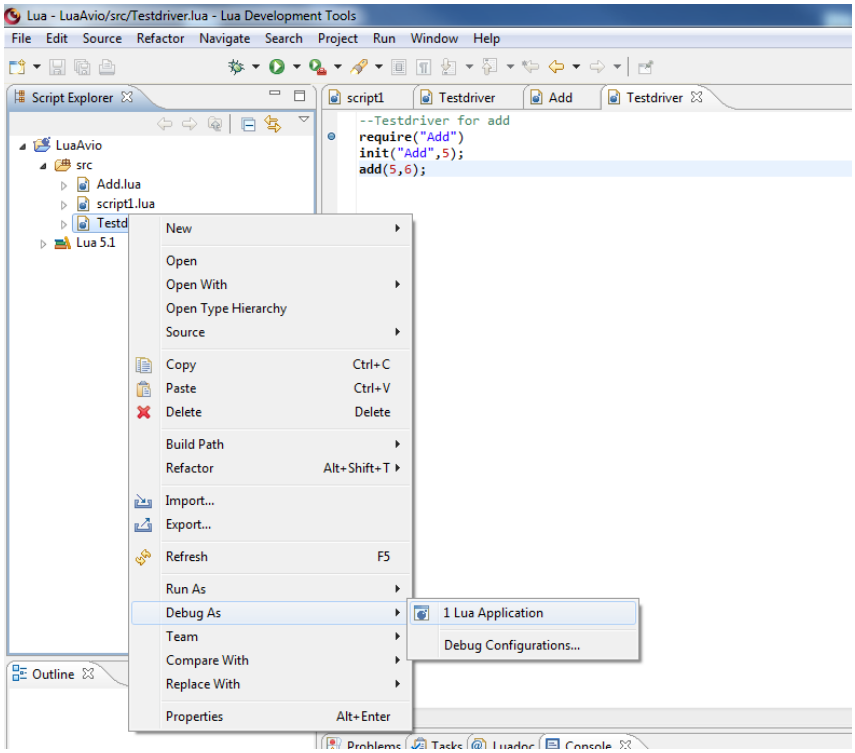

Abbildung 7: Debuggen des Skripts über den Testtreiber

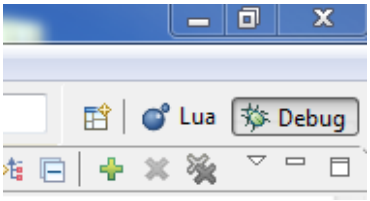

Abbildung 8: Wechseln zwischen Debug- und Entwicklungsperspektive

Über die Buttons „Lua“ und „Debug“ rechts oben (siehe Abbildung 8) kann zwischen den Perspektiven hin- und hergewechselt werden.

Durch den erstellten Breakpoint befinden wir uns nun in der 1. Zeile des Testdrivers. Über den Menüeintrag „Run - Step Over“ bzw. der Taste *F6* kann die aktuelle Zeile ausgeführt werden, mit „Run - Step Into“ bzw. der Taste *F5* kann in die Funktion der aktuellen Zeile hineingegangen werden.
Die 1. Zeile mit `require("Add")` dient dazu das zu testende Skript sichtbar zu machen (siehe 5.1.1) und kann mit „Step Over“ übersprungen werden.
Nun wird als nächstes die Funktion *init* aufgerufen, die beim Laden des Skripts aufgerufen wird und die nötigen Ports und Channels erstellt (siehe Kapitel 4.3). Diese Funktion kann mit „Step Into“ angezeigt werden. Befindet man sich nun in dieser Funktion, kann das Anlegen der einzelnen Ports und Channels beim Durchsteppen mit „Step Over“ der einzelnen Funktionen beobachtet werden. In der Konsolenausgabe ist die Textausgabe der einzelnen Funktionen ersichtlich (siehe Abbildung 9).

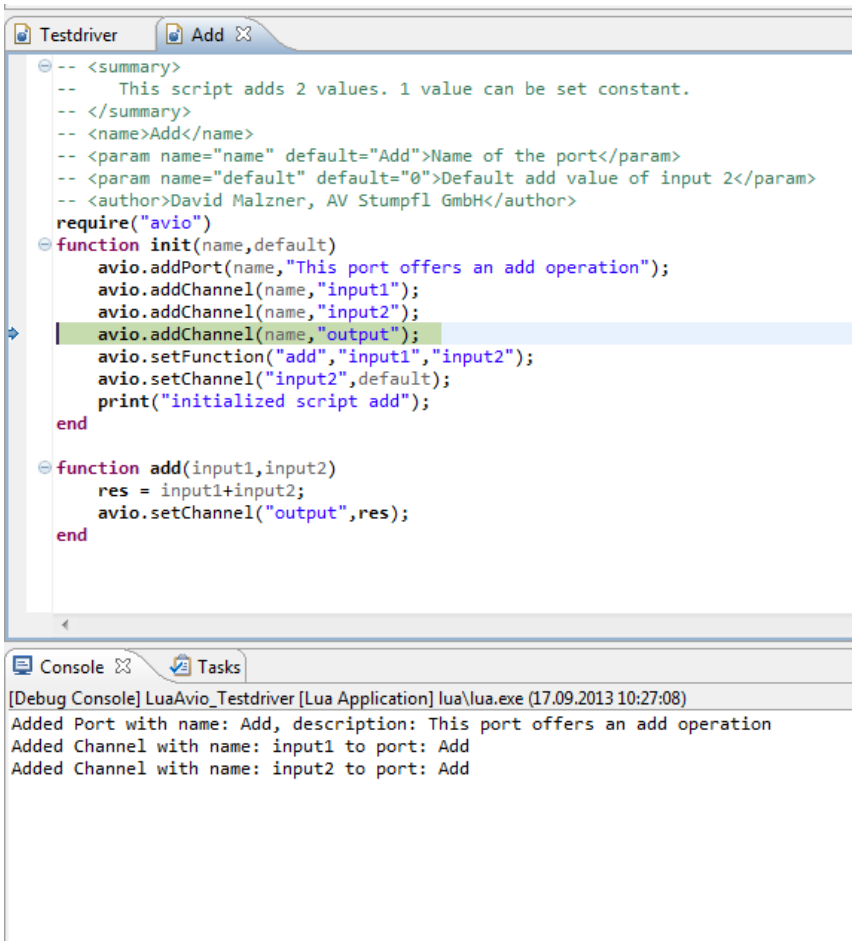

Abbildung 9: Durchsteppen der Init Funktion

Wurde die Funktion *init* nun Schritt für Schritt durchgegangen, kann mit „Step Into“ die Funktion *add* aufgerufen werden, welche später im Avio aufgerufen wird, wenn sich einer der zu addierenden Eingangswerte ändert. Befindet man sich in der Funktion, können die Werte der Variablen ausgelesen werden. Dazu wird der Name der Variable in ein leeres Feld im „Watch Window“ unter dem Reiter „Expressions“ geschrieben. Daneben erscheint dann der Wert (siehe Abbildung 10). Es können auch komplexere Ausdrücke ausgewertet werden, in unserem Fall werden an dieser Stelle die beiden Eingangswerte *input1* und *input2* addiert.

5.2 „Online“ direkt auf einem Avio-Knoten

Im Gegensatz zum „OfflineDebuggen kann ein Skript auch debuggt werden, wenn es bereits auf einem Avio-Knoten läuft. Der Vorteil dabei ist, dass es unter realen Bedingungen getestet werden kann und kein Testtreiber geschrieben werden muss. Der Nachteil dabei ist, dass eventuell nötige Änderungen im Skript etwas länger dauern.
In dieser Anleitung wird das „Online“-Debuggen anhand des Skripts „Add“ erklärt, das zuvor in Kapitel 5.1 behandelt wurde.
Um das Skript „Add“ nun „online“ debuggen zu können, wird es über die Weboberfläche gestartet (siehe Kapitel 2). Über den Rechtsklick auf den erstellen Port des Skripts im Avio Manager

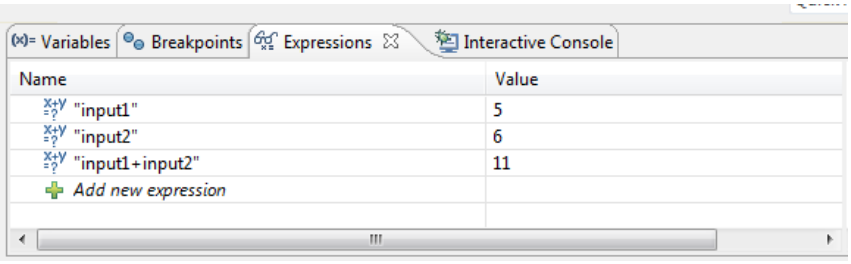

Abbildung 10: Anzeigen der Variablenwerte

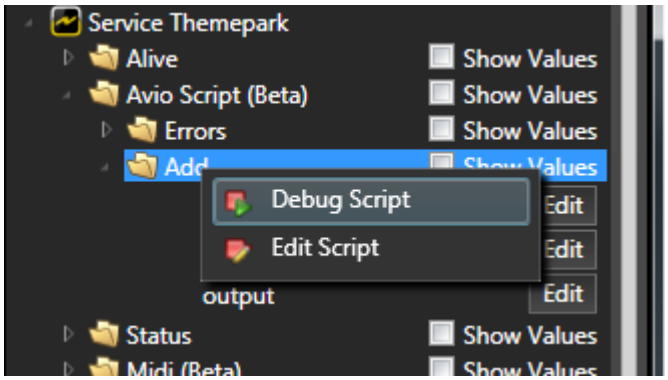

Abbildung 11: Skript aus Manager Debuggen

erscheint ein Kontextmenü. Hier kann „Debug Skript“ ausgewählt werden (siehe Abbildung 11).

Das zu debuggende Skript kann sich übrigends auf einem beliebigen Knoten im Netzwerk befinden, es muss nicht auf dem Service des lokalen PCs gestartet sein. Nach der Auswahl startet eine neue Instanz der „Lua Development Tools“ mit den dazugehörigen Einstellungen für das „Online“-Debuggen. Zu beachten ist, dass alle laufenden Instanzen der Entwicklungsumgebung an dieser Stelle beendet werden und Änderungen nicht gespeichert werden.
Nach dem Starten des Debuggers der Entwicklungsumgebung befindet man sich in der 1. Zeile der Init-Funktion. Diese kann wie im vorherigen Kapitel beschrieben Zeile für Zeile mit „Step Into“ und „Step Over“ durchgegangen werden oder gleich komplett mit „Run - Resume“ bzw. *F8* fertig durchgelaufen werden. Ebenfalls können hier wie beim „Offline“-Debuggen beschrieben im „Watch Window“ einzelne Variablen und Ausdrücke beobachtet werden.
Da uns auch interessiert was passiert, wenn die *add*-Funktion aufgerufen wird, wird per Doppelklick ein Breakpoint erstellt neben die 1. Zeile der Funktion außerhalb des Textbereichs. Der Breakpoint ist dann durch einen blauen Punkt sichtbar (siehe Abbildung 12).

Diese Funktion wird nun aufgerufen, wenn sich ein Eingangswert ändert (siehe Listing 15,

```
function add(input1,input2)
    res = input1+input2;
    avio.setChannel("output",res);
end
```

Abbildung 12: Breakpoint erstellen

Zeile 14 bzw. Kapitel 4.4.3). Um dies durchzuführen, wird z.B. im Avio Manager der Wert des Eingangschannels des Skripts geändert. Dies ist im Debugger sichtbar, indem sich die aktuelle ausgeführte Zeile beim erstellten Breakpoint befindet. Nun können die aktuellen Variablen im „Watch Window“ wieder beobachtet werden und die Funktion Zeile für Zeile durchgeangen werden.

6 Abschluss

Diese Dokumentation sollte die Einbindung und den empfohlenen Workflow für das Erstellen und Debuggen von Avio Scripts erklären. Weiters wurde eine kurze Einführung in die Skriptsprache Lua gegeben. Abseits der grundlegenden Einführung ist mit Lua jedoch viel mehr möglich. So ist es mit der Standardbibliotheken *os* für „Operating System“ möglich auf das Dateisystem zuzugreifen, Programme zu starten und auf die Uhrzeit zuzugreifen. Mit der Bibliothek *string* sind komplexe Manipulationen von Zeichenketten möglich. Die Bibliothek *math* bietet alle denkbaren mathematischen Operationen an.

Neben den Standardbibliotheken werden wie in Kapitel 3.6 angegeben im Avio-System die Bibliotheken *LuaXML* und *Socket* mitgeliefert. Die Bibliothek *LuaXML* unterstützt das Parsen und Auslesen von *XML* Dateien. Diese Bibliothek findet im mitgelieferten Standardskript *Weather* Verwendung, um die über das Internet abgerufenen Wetterinformationen einer XML Datei auszulesen.
Die Bibliothek *Socket* bietet vollen Zugriff auf das Netzwerk mittels *TCP* oder *UDP* Protokoll. Als höhere Funktionen werden hier das *SMTP* Protokoll zum Senden von E-Mails unterstützt und das *HTTP* Protokoll, um Webseiten zu laden. Diese Bibliothek findet in den Standardskripts *Email* und *Weather* Verwendung. Die Dokumentation dieser Bibliotheken würde den Rahmen dieses Dokuments sprengen, hier wird auf die Internetseite des jeweiligen Projekts verwiesen (siehe Kapitel 3.6).

Diese Anleitung sollte zeigen, wie es mit einfach zu erlernenden Konstrukten möglich ist, nahezu alle typisch auftretenden Logikaufgaben lösen zu können.
Für eine tiefere Auseinandersetzung mit der Skriptsprache Lua kann neben der zahlrichen Online Literatur auch das Buch *Programming in Lua* von *Roberto Ierusalimschy*, einem der Chefentwickler der Programmiersprache, empfohlen werden. Es wird nochmals darauf hingewiesen, dass die Möglichkeit geboten wird, sich tiefer mit der Materie zu beschäftigen, dies jedoch die Grundfunktion von Avio Script sprengt und seitens AV Stumpfl daher kein Support dafür geleistet werden kann.