1 Introduction

An LGI law can be expressed with Event-Condition-Action (ECA) Blocks. Any language that can express ECA blocks can potentially be developed into a Domain-Specific Language (DSL) for writing LGI laws. Such a DSL is called a LawScript.

Currently, 4 variants of LawScript are developed, respectively based on Prolog, Java, JavaScript, and CoffeeScript. The variants based on Prolog and Java are described in the manual of LGI.

This document introduces CoffeeScript-based LawScript.

2 CoffeeScript

CoffeeScript is a script language that is semantically equivalent to JavaScript. It’s created by Jeremy Ashkenas in 2009. It provides much syntactic sugar to JavaScript, which makes it much easier to read and write than JavaScript. Its syntactic sugar about anonymous function, operations on complex data structures, and indent-based block control makes it an ideal script language for writing LGI Laws.

This document will not cover the syntax of CoffeeScript, because its homepage covers the syntax in detail with many good examples, making it the best resource for learning. So before moving forward in this document, if you are not familiar with CoffeeScript, please first visit its homepage at: http://coffeescript.org.

3 The Structure of Law File

A Law file is a plain text file written in LawScript, conventionally with the extension of “.law”. It consists of 3 parts:

1. Law signature
2. Global variables and functions
3. ECA blocks

3.1 Law Signature

*Law signature* states the name of the law and the LawScript variant in which the law is written. Its syntax is:

```plaintext
law(<law name>,language(<variant name>))
```

For CoffeeScript-based LawScript, the variant name is *coffeescript*.

For example, if a law written in Coffeescript-based LawScript has the name *trivial*, then its signature is:

```plaintext
law(trivial,language(coffeescript))
```

3.2 Global Variables and Functions

In the *Global variables and functions* part, we can define some global variables or functions for later use, such as helper functions and shared variables.

There is one restriction on naming: the name of a variable or a function should never be all-uppercase, i.e., names like *FOO* or *BAR* are forbidden.

3.3 ECA Blocks

*ECA blocks* are the main parts of a Law file. It describes when an event occurs, under a specific condition, what should be done. In CoffeeScript-based LawScript, the syntax of an ECA block is:

```plaintext
UPON "<event name>", ->
<handler>
```

The `<handler>` of an event is essentially an anonymous function that emits regulated operations of LGI.

An event can have multiple ECA blocks, and they will be executed one-by-one, until the handling logic of one of them returns *true*.

Here is an actual example of an ECA block:

```plaintext
UPON "arrived", ->
  if @message is "probe"
    DO "deliver"
    return true
```
This ECA block means, when an arrived event occurs (which means the agent has an incoming message), if the message is probe, then perform a deliver operation (which means to deliver the message to the actor), then return true to prevent other arrived handlers from being evaluated.

The details of events and operations will be discussed below.

4 Events

Currently all LGI events described in the manual are supported.

An event usually carries several arguments. For example, an arrived event usually carries the name of its sender, the name of its destination, and the message itself. The event handling logic usually determines what to do by checking the arguments of the event and the control state of the controller. The syntax for accessing an event’s argument value is:

@<argument name>

For example, @sender will return the value of sender argument of the incoming event, indicating the sender’s identification.

All events have 4 special arguments:

- **self**: the name of the agent on which the event occurs;
- **self_law_name**: the name of the law which the agent is following;
- **self_law_hash**: the hash of the law;
- **timestamp**: the local timestamp when the event occurs.

The following is an incomplete list of events and their arguments:

- **sent**: triggered when the agent is about to send a message. Arguments:
  - **sender**: the name of the sender agent;
  - **receiver**: the name of the receiver agent;
  - **peer_law_name**: the law name of the receiver agent;
  - **message**: the message that is being sent. If message is a valid JSON string, then it will be parsed into a JSON object, otherwise it will keep its original string form. This feature holds for all message arguments below.

- **arrived**: triggered when the agent has an incoming message. Arguments:
  - **sender**: the name of the sender agent;
  - **receiver**: the name of the receiver agent;
  - **peer_law_name**: the law name of the sender agent;
  - **message**: the incoming message.
• **submitted**: triggered when the agent has an incoming message from a host that is not regulated by any LGI law. Arguments:
  
  – **sender**: the hostname of the sender;
  – **port**: the port of the sender;
  – **receiver**: the name of the receiver agent;
  – **message**: the incoming message.

• **obligation_due**: triggered when an obligation dues. Argument:
  
  – **type**: the obligation type.

• **exception**: triggered when an exception is raised. Arguments:
  
  – **sender**: the name of the sender agent;
  – **receiver**: the name of the receiver agent;
  – **message**: the incoming message;
  – **law_name**: the law name;
  – **cause**: the cause of the exception.

• **adopted**: triggered when a controller is adopted by an actor, i.e., when the agent is formed. Arguments:
  
  – **arguments**: the initial arguments of the agent;
  – **issuer**: the issuer of the certificate;
  – **subject**: the subject of the certificate;
  – **attribute**: the attribute of the certificate.

• **disconnected**: triggered when the actor disconnects with the controller, i.e., when the agent is dissolved.

• **reconnected**: triggered when the actor reconnects to the controller, i.e., when the agent is revived.

## 5 Operations

One of the main ideas of LGI is: when an event occurs and the conditions are met, the law interpreter returns a list of **regulated operations** to the controller, and the controller executes them in a standardized, atomic way. Currently all LGI operations described in the manual are supported.

As aforementioned, `<handler>` emits regulated operations of LGI. The syntax of emitting an operation is:

```
DO "<operation name>" [ <arguments> ]
```

For example,

```
DO "set", key: "token", value: true
```
means to emit a set operation, with the arguments “key: "token", value: true”.

The arguments are actually key-value pairs, in which a key is the name of the argument, and the value is the value of the argument. The order of the arguments doesn’t matter.

The following is an incomplete list of operations and their arguments:

• **forward**: forward a message to another agent. Arguments:
  – sender: the name of the sender agent, default as the value of current event’s sender argument;
  – receiver: the name of the receiver agent, default as the value of current event’s receiver argument;
  – law_name: the receiver agent’s law name, default as the current agent’s law name.
  – message: the message to forward, default as the value of current event’s message argument. message can be any type of Javascript object, not only string, and will be parsed into a JSON string before sending. This feature holds for all message arguments below.

• **deliver**: deliver a message to the actor of this agent. Arguments:
  – sender: the name of the sender agent, default as the value of current event’s sender argument;
  – receiver: the name of the receiver agent, default as the value of current event’s receiver argument;
  – message: the message to deliver, default as the value of current event’s message argument.

• **release**: send a message to a host that is not governed by LGI. Arguments:
  – sender: the name of the sender agent;
  – receiver: the hostname of the receiver;
  – port: the port of the receiver;
  – message: the message to release.

• **impose_obligation**: impose an obligation. Arguments:
  – type: the type of the obligation;
  – time: the time interval of the obligation;
  – unit: the time unit, default as “second”.

• **repeal_obligation**: repeal an obligation. Argument:
  – type: the type of the obligation to repeal.

• **set**: set a control state item. Arguments:
  – key: the name of the control state item;
  – value: the value of the control state item.

• **unset**: remove a control state item. Argument:
  – key: the name of the control state item.

• **quit**: gracefully dissolve the agent.
6 Control State

We can use set operation to create or update a control state item, and unset to remove a control state item. A control state item can have any CoffeeScript type, e.g., numbers, string, boolean, array, and object.

The syntax to get the value of an existing control state item is:

CS("<item name>")

For example, CS("foo") will return the value of the control state item foo.

Note that since the change of control state is a regulated operation, it will be executed atomically with other operations emitted by current event’s handler, therefore reading a control state after setting it in the same event handler will only get undefined.

For example, consider the code:

```coffee
DO "set", key: "foo", value: "bar"
if CS("foo") is "bar"
  DO "forward"
```

Here the statement `DO "forward"` will not be executed, because `CS("foo")` returns undefined now.

If you want to refer to a control state right after setting it, you should do it in this way:

```coffee
foo = "bar"
DO "set", key: "foo", value: foo
if foo is "bar"
  DO "forward"
```

7 Conformance Hierarchy

Conformance Hierarchy is a tree-like structure, which is formed within the law signature of the involved Law files. When an agent decide to follow a Conformance Hierarchy, it should choose one of its leaf laws, then the agent is actually following a law path from the root law to this leaf law.

If the law is in a Conformance Hierarchy and is not a root law, then its law signature should include a pointer to its parent law. The syntax to construct the pointer is:

```
portal(<parent law name>,lawURL(<parent law url>))
```

And the law signature should be:
law(<name>,language(coffeescript)) refined <parent law name>

For example, if a law state has a parent law named federal, then its law signature should be:

law(state,language(coffeescript)) refined federal
portal(federal,lawURL(http://moses.rutgers.edu/federal.law))

When an event occurs, a law may consult its direct subordinate law in the law path about the ruling of this event, in the form of an ordered list of operations, then makes its own ruling based on this advice. This procedure runs recursively, and finally, the root law makes its final ruling.

In CoffeeScript-based LawScript, this is done with such a syntax:

DELEGATE @event

This will return a list of regulated operations emitted by the subordinate law.

A regulated operation is represented by a list of key value pairs about the operation’s arguments, which contains a special key value pair “operation: <operation name>” indicating the operation’s name. Then the law may use this list to make its final ruling.

8 Ping-Pong Law: An Example

Now we examine a concrete example of an LGI law written in CoffeeScript-based LawScript called “Ping-Pong law”, which enforces the protocol of polite conversation:

• Every message must be of a type of ping or pong;
• Once an agent x sends a ping message to another agent y, x will not send other ping messages to y until it receives a pong message from y.
• y can only send one pong message to x for every ping message it gets from x.

Figure 1 shows how Ping-Pong law is written in CoffeeScript-based LawScript.
UPON "sent", ->
  if @message.type is "ping" and not CS("pingTo:#@receiver")?
    DO "set", key: "pingTo:#@receiver", value: true
    DO "forward"
    return true

UPON "arrived", ->
  if @message.type is "ping"
    DO "set", key: "pingFrom:#@sender", value: true
    DO "deliver"
    return true

UPON "sent", ->
  if @message.type is "pong" and CS("pingFrom:#@receiver")?
    DO "unset", key: "pingFrom:#@receiver"
    DO "forward"
    return true

UPON "arrived", ->
  if @message.type is "pong"
    DO "unset", key: "pingTo:#@sender"
    DO "deliver"
    return true

UPON "disconnected", ->
  DO "quit"

Figure 1: Ping-Pong Law