Amazon States Language

This document describes a JSON-based language used to describe state machines declaratively. The state machines thus defined may be executed by software. In this document, the software is referred to as "the interpreter".

Table of Contents

Structure of a State Machine

A State Machine is represented by a JSON Object.

Example: Hello World

The operation of a state machine is specified by states, which are represented by JSON objects, fields in the top-level “States” object. In this example, there is one state named “Hello World”.

```
{
    "Comment": "A simple minimal example of the States language",
    "StartAt": "Hello World",
    "States": {
    "Hello World": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",
      "End": true
    }
  }
}
```

When this state machine is launched, the interpreter begins execution by identifying the Start State. It executes that state, and then checks to see if the state is marked as an End State. If it is, the machine terminates and returns a result. If the state is not an End State, the interpreter looks for a “Next” field to determine what state to run next; it repeats this process until it reaches a Terminal State (Succeed, Fail, or an End State) or a runtime error occurs.

In this example, the machine contains a single state named “Hello World”. Because “Hello World” is a Task State, the interpreter tries to execute it. Examining the value of the “Resource” field shows that it points to a Lambda function, so the interpreter attempts to invoke that function. Assuming the Lambda function executes successfully, the machine will terminate successfully.

A State Machine is represented by a JSON object.

Top-level fields

A State Machine MUST have an object field named “States”, whose fields represent the states.

A State Machine MUST have a string field named “StartAt”, whose value MUST exactly match one of names of the “States” fields. The interpreter starts running the the machine at the named state.

A State Machine MAY have a string field named “Comment”, provided for human-readable description of the machine.

A State Machine MAY have a string field named “Version”, which gives the version of the States language used in the machine. This document describes version 1.0, and if omitted, the default value of “Version” is the string “1.0”.

A State Machine MAY have an integer field named “TimeoutSeconds”. If provided, it provides the maximum number of seconds the machine is allowed to run. If the machine runs longer than the specified time, then the interpreter fails the machine with a `States.Timeout` Error Name.

Concepts

States

States are represented as fields of the top-level “States” object. The state name, whose length MUST BE less than or equal to 128 Unicode characters, is the field name; state names MUST be unique within the scope of the whole state machine. States describe tasks (units of work), or specify flow control (e.g. Choice).

Here is an example state that executes a Lambda function:

```
"HelloWorld": {
  "Type": "Task",
  "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",
  "Next": "NextState",
  "Comment": "Executes the HelloWorld Lambda function"
}
```

Note that:

1. All states MUST have a “Type” field. This document refers to the values of this field as a state’s *type*, and to a state such as the one in the example above as a Task State.

2. Any state MAY have a “Comment” field, to hold a human-readable comment or description.

3. Most state types require additional fields as specified in this document.

4. Any state except for Choice, Succeed, and Fail MAY have a field named "End" whose value MUST be a boolean. The term “Terminal State” means a state with with `{ "End": true }`, or a state with `{ "Type": "Succeed" }`, or a state with `{ "Type": "Fail" }`.

Transitions

Transitions link states together, defining the control flow for the state machine. After executing a non-terminal state, the interpreter follows a transition to the next state. For most state types, transitions are unconditional and specified through the state's “Next” field.

All non-terminal states MUST have a “Next” field, except for the Choice state. The value of the “Next” field MUST exactly and case-sensitively match the name of the another state.

States can have multiple incoming transitions from other states.

Timestamps

The Choice and Wait states deal with JSON field values which represent timestamps. These are strings which MUST conform to the RFC3339 profile of ISO 8601, with the further restrictions that an uppercase “T” character MUST be used to separate date and time, and an uppercase “Z” character MUST be present in the absence of a numeric time zone offset, for example “2016-03-14T01:59:00Z”.

Data

The interpreter passes data between states to perform calculations or to dynamically control the state machine’s flow. All such data MUST be expressed in JSON.

When a state machine is started, the caller can provide an initial JSON text as input, which is passed to the machine's start state as input. If no input is provided, the default is an empty JSON object, {}. As each state is executed, it receives a JSON text as input and can produce arbitrary output, which MUST be a JSON text. When two states are linked by a transition, the output from the first state is passed as input to the second state. The output from the machine's terminal state is treated as its output.

For example, consider a simple state machine that adds two numbers together:

```
{
  "StartAt": "Add",
  "States": {
    "Add": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Add",
      "End": true
    }
  }
}
```

Suppose the "Add" Lambda function is defined as:

```
exports.handler = function(event, context) {
  context.succeed(event.val1 + event.val2);
};
```

Then if this state machine was started with the input `{ "val1": 3, "val2": 4 }`, then the output would be the JSON text consisting of the number 7.

The usual constraints applying to JSON-encoded data apply. In particular, note that:

1. Numbers in JSON generally conform to JavaScript semantics, typically corresponding to double-precision IEEE-854 values. For this and other interoperability concerns, see RFC 7159.
2. Standalone "-delimited strings, booleans, and numbers are valid JSON texts.

Paths

A Path is a string, beginning with "$", used to identify components with a JSON text. The syntax is that of JsonPath.

Reference Paths

A Reference Path is a Path with syntax limited in such a way that it can only identify a single node in a JSON structure: The operators “@”, “,”, “:”, and “?” are not supported - all Reference Paths MUST be unambiguous references to a single value, array, or object (subtree).

For example, if state input data contained the values:

```
{
    "foo": 123,
    "bar": ["a", "b", "c"],
    "car": {
        "cdr": true
    }
}
```

Then the following Reference Paths would return:

```
$.foo => 123
$.bar => ["a", "b", "c"]
$.car.cdr => true
```

Paths and Reference Paths are used by certain states, as specified later in this document, to control the flow of a state machine or to configure a state's settings or options.

Here are some examples of acceptable Reference Path syntax:

```
$.store.book
$.store\.book
$.\stor\e.boo\k
$.store.book.title
$.foo.\.bar
$.foo\@bar.baz\[\[.\?pretty
$.&Ж中.\uD800\uDF46
$.ledgers.branch[0].pending.count
$.ledgers.branch[0]
```

```
$.ledgers[0][22][315].foo
$['store']['book']
$['store'][0]['book']
```

Input and Output Processing

As described above, data is passed between states as JSON texts. However, a state may want to process only a subset of its input data, and may want that data structured differently from the way it appears in the input. Similarly, it may want to control the format and content of the data that it passes on as output.

Fields named “InputPath”, “Parameters”, “OutputPath”, and “ResultPath” exist to support this. Any state except for a Fail State MAY have “InputPath” and “OutputPath”. States which potentially generate results MAY have “ResultPath” and “Parameters”: Pass State, Task State, and Parallel State.

In this discussion, “raw input” means the JSON text that is the input to a state. “Result” means the JSON text that a state generates, for example from external code invoked by a Task State, the combined result of the branches in a Parallel State, or the value of the “Result” field in a Pass state. “Effective input” means the input after the application of InputPath and Parameters, and “effective output” means the final state output after processing the result with ResultPath and OutputPath.

InputPath, Parameters, OutputPath, DefaultPath

1. The value of “InputPath” MUST be a Path, which is applied to a State’s raw input to select some or all of it; that selection is used by the state, for example in passing to Resources in Task States and Choices selectors in Choice States.

2. “Parameters” may have any value. Certain conventions described below allow values to be extracted from the effective input and embedded in the Parameters structure. If the “Parameters” field is provided, its value, after the extraction and embedding, becomes the effective input.

3. The value of “ResultPath” MUST be a Reference Path, which specifies the raw input’s combination with or replacement by the state’s result.

4. The value of "OutputPath" MUST be a Path, which is applied to the state's output after the application of ResultPath, producing the effective output which serves as the raw input for the next state.

Note that JsonPath can yield multiple values when applied to an input JSON text. For example, given the text:

```
{ "a": [1, 2, 3, 4] }
```

Then if the JsonPath `$.a[0,1]` is appplied, the result will be two JSON texts, 1 and 2. When this happens, to produce the effective input, the interpreter gathers the texts into an array, so in this example the state would see the input:

```
[ 1, 2 ]
```

The same rule applies to OutputPath processing; if the OutputPath result contains multiple values, the effective output is a JSON array containing all of them.

The ResultPath field's value is a Reference Path that specifies where to place the result, relative to the raw input. If the input has a field which matches the ResultPath value, then in the output, that field is discarded and overwritten by the state output. Otherwise, a new field is created in the state output.

If the value of InputPath is `null`, that means that the raw input is discarded, and the effective input for the state is an empty JSON object, {}. Note that having a value of `null` is different from the InputPath field being absent.

If the value of of ResultPath is `null`, that means that the state's own raw output is discarded and its raw input becomes its result.

If the value of OutputPath is `null`, that means the input and result are discarded, and the effective output from the state is an empty JSON object, {}.

Defaults

Each of InputPath, Parameters, ResultPath, and OutputPath are optional. The default value of InputPath is “$”, so by default the effective input is just the raw input. The default value of ResultPath is “$”, so by default a state’s result overwrites and replaces the input. The default value of OutputPath is “$”, so by default a state’s effective output is the result of processing ResultPath.

Parameters has no default value. If it is absent, it has no effect on the effective input.

Therefore, if none of InputPath, Parameters, ResultPath, or OutputPath are supplied, a state consumes the raw input as provided and passes its result to the next state.

Input/Output Processing Examples

Consider the example given above, of a Lambda task that sums a pair of numbers. As presented, its input is: `{ "val1": 3, "val2": 4 }` and its output is: 7.

Suppose the input is little more complex:

```
{
  "title": "Numbers to add",
  "numbers": { "val1": 3, "val2": 4 }
}
```

Then suppose we modify the state definition by adding:

```
"InputPath": "$.numbers",
"ResultPath": "$.sum"
```

And finally,suppose we simplify Line 4 of the Lambda function to read as follows: `return JSON.stringify(total)`. This is probably a better form of the function, which should really only care about doing math and not care how its result is labeled.

In this case, the output would be:

```
{
  "title": "Numbers to add",
  "numbers": { "val1": 3, "val2": 4 },
  "sum": 7
}
```

The interpreter might need to construct multiple levels of JSON object to achieve the desired effect. Suppose the input to some Task state is:

```
{ "a": 1 }
```

Suppose the output from the Task is “Hi!”, and the value of the “ResultPath” field is “$.b.greeting”. Then the output from the state would be:

```
{
  "a": 1,
  "b": {
    "greeting": "Hi!"
  }
}
```

Parameters

The value of the “Parameters” field (after processing described below) becomes the effective input. Consider the following Task state:

```
"X": {
  "Type": "Task",
  "Resource": "arn:aws:swf:us-east-1:123456789012:task:X",
  "Next": "Y",
  "Parameters": {
    "first": 88,
```

```
      "second": 99
    }
  }
```

In this case, the effective input to the code identified in the Resource field would be the object with “first” and “second” fields which is the value of the “Parameters” field.

Values from the effective input can be inserted into the “Parameters” field with a combination of a field-naming convention and JsonPath.

If any JSON object within the value of Parameters (however deeply nested) has a field whose name ends with the characters “.$”, its value MUST be a Path. In this case, the Path is applied to the effective input and the result is called the Extracted Value.

If the path is legal but cannot be applied successfully the Interpreter fails the machine execution with an Error Name of “States.ParameterPathFailure”.

When a field name ends with “.$” and its value can be used to generate an Extracted Value as described above, the field is replaced within the Parameters value by another field whose name is the original name minus the “.$” suffix, and whose value is the Extracted Value.

Consider this example:

```
"X": {
  "Type": "Task",
  "Resource": "arn:aws:swf:us-east-1:123456789012:task:X",
  "Next": "Y",
  "Parameters": {
    "flagged": true,
    "parts": {
      "first.$": "$.vals[0]",
      "last3.$": "$.vals[3:]"
    }
  }
}
```

Suppose that the input to the state is as follows:

```
{
  "flagged": 7,
  "vals": [0, 10, 20, 30, 40, 50]
}
```

In this case, the effective input to the code identified in the Resource field would be as follows:

```
{
  "flagged": true,
  "parts": {
    "first": 0,
    "last3": [30, 40, 50]
  }
}
```

Runtime Errors

Suppose a state's input is the string `"foo"`, and its "ResultPath" field has the value "$.x". Then ResultPath cannot apply and the Interpreter fails the machine with Error Name of "States.ResultPathMatchFailure".

Errors

Any state can encounter runtime errors. Errors can arise because of state machine definition issues (e.g. the "ResultPath" problem discussed immediately above), task failures (e.g. an exception thrown by a Lambda function) or because of transient issues, such as network partition events.

When a state reports an error, the default course of action for the interpreter is to fail the whole state machine.

Error representation

Errors are identified by case-sensitive strings, called Error Names. The States language defines a set of built-in strings naming well-known errors, all of which begin with the prefix “States.”; see Appendix A.

States MAY report errors with other names, which MUST NOT begin with the prefix “States.”.

Retrying after error

Task States and Parallel States MAY have a field named “Retry”, whose value MUST be an array of objects, called Retriers.

Each Retrier MUST contain a field named “ErrorEquals” whose value MUST be a non-empty array of Strings, which match Error Names.

When a state reports an error, the interpreter scans through the Retriers and, when the Error Name appears in the value of of a Retrier’s “ErrorEquals” field, implements the retry policy described in that Retrier.

An individual Retrier represents a certain number of retries, usually at increasing time intervals.

A Retrier MAY contain a field named “IntervalSeconds”, whose value MUST be a positive integer, representing the number of seconds before the first retry attempt (default value: 1); a field named “MaxAttempts” whose value MUST be a non-negative integer, representing the maximum number of retry attempts (default: 3); and a field named “BackoffRate”, a number which is the multiplier that increases the retry interval on each attempt (default: 2.0). The value of BackoffRate MUST be greater than or equal to 1.0.

Note that a “MaxAttempts” field whose value is 0 is legal, specifying that some error or errors should never be retried.

Here is an example of a Retry field which will make 2 retry attempts after waits of 3 and 4.5 seconds:

```
"Retry" : [
    {
      "ErrorEquals": [ "States.Timeout" ],
      "IntervalSeconds": 3,
      "MaxAttempts": 2,
```

```
      "BackoffRate": 1.5
    }
  ]
```

The reserved name “States.ALL” in a Retrier’s “ErrorEquals” field is a wild-card and matches any Error Name. Such a value MUST appear alone in the “ErrorEquals” array and MUST appear in the last Retrier in the “Retry” array.

Here is an example of a Retry field which will retry any error except for “States.Timeout”, using the default retry parameters.

```
  "Retry" : [
    {
      "ErrorEquals": [ "States.Timeout" ],
      "MaxAttempts": 0
    },
    {
      "ErrorEquals": [ "States.ALL" ]
    }
  ]
```

If the error recurs more times than allowed for by the “MaxAttempts” field, retries cease and normal error handling resumes.

Complex retry scenarios

A Retrier’s parameters apply across all visits to that Retrier in the context of a single state execution. This is best illustrated by example; consider the following Task State:

```
  "X": {
    "Type": "Task",
    "Resource": "arn:aws:swf:us-east-1:123456789012:task:X",
    "Next": "Y",
    "Retry": [
```

```
      {
        "ErrorEquals": [ "ErrorA", "ErrorB" ],
        "IntervalSeconds": 1,
        "BackoffRate": 2,
        "MaxAttempts": 2
      },
      {
        "ErrorEquals": [ "ErrorC" ],
        "IntervalSeconds": 5
      }
    ],
    "Catch": [
      {
        "ErrorEquals": [ "States.ALL" ],
        "Next": "Z"
      }
    ]
  }
```

Suppose that this task fails four successive times, throwing Error Names “ErrorA”, “ErrorB”, “ErrorC”, and “ErrorB”. The first two errors match the first retrier and cause waits of one and two seconds. The third error matches the second retrier and causes a wait of five seconds. The fourth error would match the first retrier but its “MaxAttempts” ceiling of two retries has already been reached, so that Retrier fails, and execution is redirected to the “Z” state via the “Catch” field.

Note that once the interpreter transitions to another state in any way, all the Retrier parameters reset.

Fallback states

Task States and Parallel States MAY have a field named “Catch”, whose value MUST be an array of objects, called Catchers.

Each Catcher MUST contain a field named “ErrorEquals”, specified exactly as with the Retrier “ErrorEquals” field, and a field named “Next” whose value MUST be a string exactly matching a State Name.

When a state reports an error and either there is no Retry field, or retries have failed to resolve the error, the interpreter scans through the Catchers in array order, and when the Error Name appears in the value of a Catcher's "ErrorEquals" field, transitions the machine to the state named in the value of the "Next" field.

The reserved name "States.ALL" appearing in a Retrier's "ErrorEquals" field is a wild-card and matches any Error Name. Such a value MUST appear alone in the "ErrorEquals" array and MUST appear in the last Catcher in the "Catch" array.

Error output

When a state reports an error and it matches a Catcher, causing a transfer to another state, the state's result (and thus the input to the state identified in the Catcher's "Next" field) is a JSON object, called the Error Output. The Error Output MUST have a string-valued field named "Error", containing the Error Name. It SHOULD have a string-valued field named "Cause", containing human-readable text about the error.

A Catcher MAY have an "ResultPath" field, which works exactly like a state's top-level "ResultPath", and may be used to inject the Error Output into the state's original input to create the input for the Catcher's "Next" state. The default value, if the "ResultPath" field is not provided, is "$", meaning that the output consists entirely of the Error Output.

Here is an example of a Catch field that will transition to the state named "RecoveryState" when a Lambda function throws an unhandled Java Exception, and otherwise to the "EndMachine" state, which is presumably Terminal.

Also in this example, if the first Catcher matches the Error Name, the input to "RecoveryState" will be the original state input, with the Error Output as the value of the top-level "error-info" field. For any other error, the input to "EndMachine" will just be the Error Output.

```
"Catch": [
  {
    "ErrorEquals": [ "java.lang.Exception" ],
    "ResultPath": "$.error-info",
    "Next": "RecoveryState"
  },
  {
```

```
    "ErrorEquals": [ "States.ALL" ],
    "Next": "EndMachine"
  }
]
```

Each Catcher can specifiy multiple errors to handle.

When a state has both Retry and Catch fields, the interpreter uses any appropriate Retriers first and only applies the a matching Catcher transition if the retry policy fails to resove the error.

State Types

As a reminder, the state type is given by the value of the “Type” field, which MUST appear in every State object.

Table of State Types and Fields

Many fields can appear in more than one state type. The table below summarizes which fields can appear in which states. It excludes fields that are specific to one state type.

	States						
	Pass	**Task**	**Choice**	**Wait**	**Succeed**	**Fail**	**Parallel**
Type	Required	Required	Required	Required	Required	Required	Required
Comment	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
InputPath, OutputPath	Allowed	Allowed	Allowed	Allowed	Allowed		Allowed
Parameters	Allowed	Allowed					Allowed
ResultPath	Allowed	Allowed					Allowed
One of: Next *or* "End":true	Required	Required		Required			Required
Retry, Catch		Allowed					Allowed

Pass State

The Pass State (identified by "Type":"Pass") simply passes its input to its output, performing no work. Pass States are useful when constructing and debugging state machines.

A Pass State MAY have a field named “Result”. If present, its value is treated as the output of a virtual task, and placed as prescribed by the “ResultPath” field, if any, to be passed on to the next state. If “Result” is not provided, the output is the input. Thus if neither “Result” nor “ResultPath” are provided, the Pass state copies its input through to its output.

Here is an example of a Pass State that injects some fixed data into the state machine, probably for testing purposes.

```
"No-op": {
  "Type": "Pass",
  "Result": {
    "x-datum": 0.381018,
    "y-datum": 622.2269926397355
  },
  "ResultPath": "$.coords",
  "Next": "End"
}
```

Suppose the input to this state were as follows:

```
{
  "georefOf": "Home"
}
```

Then the output would be:

```
{
  "georefOf": "Home",
  "coords": {
```

```
      "x-datum": 0.381018,
      "y-datum": 622.2269926397355
    }
  }
```

Task State

The Task State (identified by "Type":"Task") causes the interpreter to execute the work identified by the state's "Resource" field.

Here is an example:

```
  "TaskState": {
    "Comment": "Task State example",
    "Type": "Task",
    "Resource": "arn:aws:swf:us-east-1:123456789012:task:HelloWorld",
    "Next": "NextState",
    "TimeoutSeconds": 300,
    "HeartbeatSeconds": 60
  }
```

A Task State MUST include a "Resource" field, whose value MUST be a URI that uniquely identifies the specific task to execute. The States language does not constrain the URI scheme nor any other part of the URI.

Tasks can optionally specify timeouts. Timeouts (the "TimeoutSeconds" and "HeartbeatSeconds" fields) are specified in seconds and MUST be positive integers. If provided, the "HeartbeatSeconds" interval MUST be smaller than the "TimeoutSeconds" value.

If not provided, the default value of "TimeoutSeconds" is 60.

If the state runs longer than the specified timeout, or if more time than the specified heartbeat elapses between heartbeats from the task, then the interpreter fails the state with a `States.Timeout` Error Name.

Choice State

A Choice state (identified by "Type":"Choice") adds branching logic to a state machine.

A Choice state state MUST have a “Choices” field whose value is a non-empty array. Each element of the array is called a Choice Rule - an object containing a comparison operation and a “Next” field, whose value MUST match a state name.

The interpreter attempts pattern-matches against the Choice Rules in array order and transitions to the state specified in the “Next” field on the first Choice Rule where there is an exact match between the input value and a member of the comparison-operator array.

Here is an example of a Choice state, with some other states that it transitions to.

```
  "ChoiceStateX": {
    "Type" : "Choice",
    "Choices": [
      {
          "Not": {
            "Variable": "$.type",
            "StringEquals": "Private"
          },
          "Next": "Public"
      },
      {
        "And": [
          {
            "Variable": "$.value",
            "NumericGreaterThanEquals": 20
          },
          {
            "Variable": "$.value",
            "NumericLessThan": 30
          }
        ],
        "Next": "ValueInTwenties"
```

```
      }
    ],
    "Default": "DefaultState"
  },

  "Public": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Foo",
    "Next": "NextState"
  },

  "ValueInTwenties": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Bar",
    "Next": "NextState"
  },

  "DefaultState": {
    "Type": "Fail",
    "Cause": "No Matches!"
  }
```

In this example, suppose the machine is started with an input value of:

```
{
  "type": "private",
  "value": 22
}
```

Then the interpreter will transition to the “ValueInTwenties” state, based on the “value” field.

Each choice rule MUST contain exactly one field containing a comparison operator. The following comparison operators are supported:

1. StringEquals
2. StringLessThan
3. StringGreaterThan
4. StringLessThanEquals
5. StringGreaterThanEquals
6. NumericEquals
7. NumericLessThan
8. NumericGreaterThan
9. NumericLessThanEquals
10. NumericGreaterThanEquals
11. BooleanEquals
12. TimestampEquals
13. TimestampLessThan
14. TimestampGreaterThan
15. TimestampLessThanEquals
16. TimestampGreaterThanEquals
17. And
18. Or
19. Not

For each of these operators, the field's value MUST be a value of the appropriate type: String, number, boolean, or Timestamp.

The interpreter scans through the Choice Rules in a type-sensitive way, and will not attempt to match a numeric field to a string value. However, since Timestamp fields are logically strings, it is possible that a field which is thought of as a time-stamp could be matched by a "StringEquals" comparator.

Note that for interoperability, numeric comparisons should not be assumed to work with values outside the magnitude or precision representable using the IEEE 754-2008 "binary64" data type. In particular, integers outside of the range $[-(2^{53})+1, (2^{53})-1]$ might fail to compare in the expected way.

The values of the "And" and "Or" operators MUST be non-empty arrays of Choice Rules that MUST NOT contain "Next" fields; the "Next" field can only appear in a top-level Choice Rule.

The value of a "Not" operator MUST be a single Choice Rule, that MUST NOT contain "Next" fields; the "Next" field can only appear in a top-level Choice Rule.

Choice states MAY have a "Default" field, which will execute if none of the Choice Rules match. The interpreter will raise a run-time States.NoChoiceMatched error if a "Choice" state fails to match a Choice Rule and no "Default" transition was specified.

Choice states MUST NOT be End states.

Wait State

A Wait state (identified by `"Type":"Wait"`) causes the interpreter to delay the machine from continuing for a specified time. The time can be specified as a wait duration, specified in seconds, or an absolute expiry time, specified as an ISO-8601 extended offset date-time format string.

For example, the following Wait state introduces a ten-second delay into a state machine:

```
"wait_ten_seconds" : {
  "Type" : "Wait",
  "Seconds" : 10,
  "Next": "NextState"
}
```

This waits until an absolute time:

```
"wait_until" : {
  "Type": "Wait",
  "Timestamp": "2016-03-14T01:59:00Z",
  "Next": "NextState"
}
```

The wait duration does not need to be hardcoded. Here is the same example, reworked to look up the timestamp time using a Reference Path to the data, which might look like `{ "expirydate": "2016-03-14T01:59:00Z" }`:

```
"wait_until" : {
    "Type": "Wait",
    "TimestampPath": "$.expirydate",
    "Next": "NextState"
}
```

A Wait state MUST contain exactly one of ”Seconds”, “SecondsPath”, “Timestamp”, or “TimestampPath”.

Succeed State

The Succeed State (identified by `"Type":"Succeed"`) terminates a state machine successfully. The Succeed State is a useful target for Choice-state branches that don't do anything but terminate the machine.

Here is an example:

```
"SuccessState": {
  "Type": "Succeed"
}
```

Because Succeed States are terminal states, they have no “Next” field.

Fail State

The Fail State (identified by "Type":"Fail") terminates the machine and marks it as a failure.

Here is an example:

```
  "FailState": {
            "Type": "Fail",
            "Error": "ErrorA",
            "Cause": "Kaiju attack"
  }
```

A Fail State MUST have a string field named “Error”, used to provide an error name that can be used for error handling (Retry/Catch), operational, or diagnostic purposes. A Fail State MUST have a string field named “Cause”, used to provide a human-readable message.

Because Fail States are terminal states, they have no “Next” field.

Parallel State

The Parallel State (identified by "Type":"Parallel") causes parallel execution of "branches".

Here is an example:

```
  "LookupCustomerInfo": {
    "Type": "Parallel",
    "Branches": [
      {
        "StartAt": "LookupAddress",
        "States": {
          "LookupAddress": {
            "Type": "Task",
            "Resource":
              "arn:aws:lambda:us-east-1:123456789012:function:AddressFinder",
            "End": true
```

```
        }
      }
    },
    {
      "StartAt": "LookupPhone",
      "States": {
        "LookupPhone": {
          "Type": "Task",
          "Resource":
            "arn:aws:lambda:us-east-1:123456789012:function:PhoneFinder",
          "End": true
        }
      }
    }
  ],
  "Next": "NextState"
}
```

A Parallel state causes the interpreter to execute each branch starting with the state named in its “StartAt” field, as concurrently as possible, and wait until each branch terminates (reaches a terminal state) before processing the Parallel state's “Next” field. In the above example, this means the interpreter waits for “LookupAddress” and “LookupPhoneNumber” to both finish before transitioning to “NextState”.

In the example above, the LookupAddress and LookupPhoneNumber branches are executed in parallel.

A Parallel State MUST contain a field named “Branches” which is an array whose elements MUST be objects. Each object MUST contain fields named “States” and “StartAt” whose meanings are exactly like those in the top level of a State Machine.

A state in a Parallel state branch “States” field MUST NOT have a “Next” field that targets a field outside of that “States” field. A state MUST NOT have a “Next” field which matches a state name inside a Parallel state branch’s “States” field unless it is also inside the same “States” field.

Put another way, states in a branch’s “States” field can transition only to each other, and no state outside of that “States” field can transition into it.

If any branch fails, due to an unhandled error or by transitioning to a Fail state, the entire Parallel state is considered to have failed and all the branches are terminated. If the error is not handled by the Parallel State, the interpreter should terminate the machine execution with an error.

The Parallel state passes its input (potentially as filtered by the “InputPath” field) as the input to each branch’s “StartAt” state. it generates output which is an array with one element for each branch containing the output from that branch. There is no requirement that all elements be of the same type.

The output array can be inserted into the input data using the state’s “ResultPath” field in the usual way.

For example, consider the following Parallel State:

```
  "FunWithMath": {
    "Type": "Parallel",
    "Branches": [
      {
        "StartAt": "Add",
        "States": {
          "Add": {
            "Type": "Task",
            "Resource": "arn:aws:swf:::task:Add",
            "End": true
          }
        }
      },
      {
        "StartAt": "Subtract",
        "States": {
          "Subtract": {
            "Type": "Task",
            "Resource": "arn:aws:swf:::task:Subtract",
            "End": true
          }
        }
```

```
      }
    ],
    "Next": "NextState"
  }
```

If the “FunWithMath” state was given the JSON array `[3, 2]` as input, then both the “Add” and “Subtract” states would receive that array as input. The output of “Add” would be 5, that of “Subtract” would be 1, and the output of the Parallel State would be a JSON array:

```
[ 5, 1 ]
```

If any branch fails, due to an unhandled error or by transitioning to a Fail state, the entire Parallel state is considered to have failed and all the branches are terminated. If the error is not handled by the Parallel State, the interpreter should terminate the machine execution with an error.

Appendices

Appendix A: Predefined Error Codes

Code	Description
States.ALL	A wild-card which matches any Error Name.
States.Timeout	A Task State either ran longer than the “TimeoutSeconds” value, or failed to heartbeat for a time longer than the “HeartbeatSeconds” value.
States.TaskFailed	A Task State failed during the execution.
States.Permissions	A Task State failed because it had insufficient privileges to execute the specified code.
States.ResultPathMatchFailure	A state’s “ResultPath” field cannot be applied to the input the state received.
States.ParameterPathFailure	Within a state’s “Parameters” field, the attempt to replace a field whose name ends in “.$” using a Path failed.

States.BranchFailed	A branch of a Parallel state failed.
States.NoChoiceMatched	A Choice state failed to find a match for the condition field extracted from its input.