NASA Ames Research Center
Autonomous Systems and Robotics

PLEXIL Workshop

An Introduction to PLEXIL and the PLEXIL Executive

Part 2: Plexil Language

This presentation was developed in Open Office Impress. Template courtesy of Chih-Hao Tsai (technology.cstsai.org/impress).

Outline

- Introduction
- Actions
- Action Attributes
  - Variables
  - Conditions
  - Interface
  - Library Nodes
- Action Types
  - Core PLEXIL
    - Empty Node
    - Assignment Node
    - Command Node
    - Update Node
    - Library Call Node
    - List Node
- Action Types (continued)
  - Sequence and Concurrence
  - Conditional (If-Then-Else)
  - Loops
  - Message Passing
- Data Types and Expressions
  - The UNKNOWN value
  - Numeric Expressions
  - Logical Expressions
  - String Expressions
  - Arrays
- World State (lookups)
- Action State
- Translating Plexil into XML

Introduction

- Standard programming syntax for PLEXIL
- Example

```
SimpleAssignment:
{
  Integer foo = 0;
  PostCondition: foo == 3;
  Assignment: foo = 3;
}
```

- Translated into PLEXIL XML for execution
  - XML format described by XML schema
    - See directory `plexil/schema`

Actions

- *Actions* specify a kind of behavior
- General format:

```
<Action name>:
{
  <action attributes>
  <action body>
}
```

- Action name, attributes, and body are all optional. E.g. an empty action is:

```
{ }
```

Action Attributes

- Action Attributes
  - Variables
  - Conditions
  - Interface
  - Library Nodes

Variables

- An action may declare local variables.
  - Visible to the action and its descendants (lexical scope)
  - Of type Boolean, integer, real, string, or array
- Examples

```
Boolean isReset = true;
Integer n = 123;
Real pi = 3.14159;
String message = "hello there";
Integer scores[100];
Real defaults[10] = #(1.3 2.0 3.5);
```

Conditions

- An action's conditions are logical expressions.
  - If omitted, defaults apply
  - Up to one clause for each condition type:
    - Start, end, pre, post, invariant, skip, and repeat condition
- Examples

```
StartCondition: Node1.outcome == SUCCESS;
EndCondition: SignalEndOfPlan.state == FINISHED ||
                        SendAbortUpdate.state == FINISHED;
PostCondition: AtGoal;
InvariantCondition: LookupOnChange(ZZZZCWEC5520J) == 1;
RepeatCondition: Count < 10;
```

Interface

- Interfaces control variable visibility and access
- Example

```
{
  Integer x = 2;
  String message = “Enter number:” ;
  NodeList:
    {
      InOut Integer x;
      In String message;
      Integer y = 5;
      NodeList:
        { Assignment: x = y + 2; }
        { Command print(message); }
    }
}
```

Library Actions

- Library actions are actions you “call” in other actions.
  - They are invoked by *Library Call Nodes*.
- Any action can be a library action.
  - Library actions often have Interface clauses
    - These are the action's formal parameters.
- Library actions are top level (i.e. not nested) actions.
  - Exactly one top level action per file is required.
- Upcoming slide on library nodes has examples.

Action Types

- Core PLEXIL
  - Empty Node
  - Assignment Node
  - Command Node
  - Function Call Node
  - Update Node
  - Library Call Node
  - List Node
- Concurrence
- Sequences
- Conditional (If-Then-Else)
- While and For loops
- Inter-executive communication

- The type of the action is determined by its **body**, which comes after its (optional) **attributes**.

Core PLEXIL

- Core PLEXIL is a subset of PLEXIL.
- All PLEXIL translates into Core PLEXIL
- Basis for execution
- Basis for formal semantics
- Consists of **nodes**
  - Empty Node
  - Assignment Node
  - Command Node
  - Update Node
  - Library Call Node
  - List Node
- Nodes are kinds of actions

Empty Node

- Empty nodes have no body. They may contain only attributes.
- Example:

```
VerifyTemp:
{
 PostCondition: LookupNow("engine_temperature") > 100.0;
}
```

- Common uses for empty nodes:
  - Verification of a state (as in above example)
  - Stubs (for testing or incremental development)

Assignment Node

- Identified by an Assignment clause, e.g.

```
// A simple assignment node

IncrementCounter:
{
  Assignment: ExecutionCount = 1 + ExecutionCount;
}
```

- The assigned variable must be writable.
- The source (RHS) of the assignment is an expression whose type must match that of the variable.

Command Node

- Identified by a Command clause, e.g.

```
// A simple command node
ConfirmProceed:
{
  Boolean result;
  EndCondition: isKnown(result);
  PostCondition: result;
  Command: result = QueryYesNo("Proceed with instructions?");
}
```

- The assigned variable is optional and must be writable.

- Call to command immediately returns a *handle,* finishing the node. (Plan's execution is not blocked).
  - This is independent of the returned *value, which is lost if the node finishes before the value is returned.*

Update Node

- Updates reflect data to an external system (e.g. planner)
  - Data represented as name/value bindings
- Identified by an Update clause

```
// A simple update node
SendAbortUpdate:
{
  StartCondition: MonitorAbortSignal.state == FINISHED;
  Update: taskId = taskTypeAndId[1], result = -2;
}
```

- Any number of name/value bindings are allowed.

Library Call Node

- Identified by a LibraryCall clause
  - Example library node:

```
F:
{
  In Integer i;
  InOut Integer j;
  Assignment: j = j * j + i;
}
```

  - Example call to above library node (note declaration):

```
LibraryNode F(In Integer i, InOut Integer j);

LibraryCallTest:
{
  Integer k = 2;
  LibraryCall: F(i=12, j=k);
}
```

List Node

- Identified by a NodeList clause. Example:

```
Root:
{
  NodeList:
    Increment: { Assignment: count = count + 1; }
    Detect: {
      StartCondition: LookupOnChange("button-pressed");
    }
    React: {
      StartCondition: Detect.State == FINISHED;
      Command: activate_device()
    }
 }
```

  - The first node , `Increment`, is unconstrained.
  - The second node, `Detect`, is empty.
  - The third node, `React`, runs after `Detect` finishes.

Concurrence

- A **Concurrence** specifies parallel execution.

```
StartSystems: {
  Concurrence:
    TurnOnLights: { Command: activateLights(); }
    TurnOnCamera: { Command: activateCamera(); }
}
```

- A Concurrence is essentially a List Node.

Sequences

- A **Sequence** specifies sequential execution.

```
StartSystems: {
  Sequence:
    TurnOnLights: { Command: activateLights(); }
    TurnOnCamera: { Command: activateCamera(); }
}
```

- An **UncheckedSequence** is like a Sequence, except success of each action is not checked.

- A **Try** is like a Sequence, except that each action is executed *until* one succeeds, then it terminates.

Conditional

- The **If-Then-Else** specifies conditional execution.

```
Camera: {
  If (Lookup(PowerOn)) Then
    activateCamera: { Command: activateCamera(); }
  Else Warn: { Command: warn("No power ..."); }
}
```

- The Else is optional.
- Conditionals may be nested (use brackets accordingly).

For Loop

- The **For Loop** repeats an action over a range of numbers

```
pins: {
  For (Integer I = 0; I < 10; I + 1)
    checkAndInform: {
      Boolean state;
      Sequence:
        { EndCondition: isKnown(state);
          Command: state = checkPinState(I); }
        { Command: informPinState(I, state); }
    } // checkAndInform
}
```

While Loop

- The **While Loop** repeats an action while its expression holds

```
processItems: {
  Boolean continue = true;
  While (continue)
    processItem: {
      …
      If (…) Then { Assignment: continue = false; }
    }
}
```

- Loops can contain, or be nested within, other loops (or any other kind of action).

Inter-Executive Communication

- Multiple PLEXIL executives can communicate with each other:
  - By sending messages (strings)
  - By issuing commands

- The OnMessage action specifies an action to respond to a message.

```
HandleFinished: {
  OnMessage (“finished”)
    { Command: shutDown; }
}
```

Inter-Executive Communication (cont.)

- The **OnCommand** action specifies an action to respond to a *command* from another executive.

```
AdjustSpeed : {
  Command: speed = adjustSpeed(45.0); }

HandleAdjustSpeed : {
  OnCommand adjustSpeed (Real incr) {
    Sequence:
      { Assignment: CurSpeed = CurSpeed + incr;
      { Command: SendReturnValue (CurSpeed); }
  }
}
```

- Assume these actions are in different plans/executives.
- **SendReturnValue** is optional; default return is **true**.

Wrapup

- Data types and expressions
  - The UNKNOWN value
  - Numeric Expressions
  - Boolean Expressions
  - String Expressions
  - Arrays
- World State (lookups)
- Node State
- Translating into XML

The UNKNOWN value

- Every type includes the UNKNOWN value.
- Default initial value for variables and array elements
- Results when a lookup fails
- Results when a requested node timepoint is invalid
- Part of PLEXIL's *three-valued* logic
  - True, False, Unknown
- Not a literal – cannot be used in a plan
  - Instead, queried through **isKnown** operator

Numeric Expressions

- Evaluate to numbers (integer or real)
- Literals
  - Integers
  - Reals
- Variables of type integer or real
- Lookups
- Node timepoint values
- Arithmetic operations
  - Add, subtract, multiply, divide
  - Square root, absolute value
- Arrays: size, element index, elements (for numeric arrays)

Numeric Expressions (cont.)

- Examples

```
234
12.9
X (where X was declared Integer)
Bar (where Bar was declared Real)
LookupNow ("ExternalTemperature")
TakePicture.EXECUTING.START   (a node timepoint)
Bar + 4.5
X - (30 + LookupNow("x") )
3 * X
(3 * X)/(X - 20)
sqrt(X)
abs(X)
Entries[X] (where Entries is an array of integers)
```

Logical Expressions

- Boolean literals
  - `true, false`
- Boolean-typed variables
  - `Boolean flag = false;`
  - `StartCondition: flag;`
- Lookups that return a Boolean-valued state
- Array elements (of Boolean arrays)

Logical Expressions (cont.)

- Comparison
  - Equal, not equal

```
Postcondition: attempts == successes;
Precondition: arm_status != engaged;
```

  - Less than, greater than (or equal)

```
StartCondition: temperature < 70;
InvariantCondition: altitude > 4000;
PreCondition: LookupOnChange(“score”) >= 10;
Precondition: LookupNow(“tachometer”) < 6500;
```

Boolean Expressions (cont.)

- Operations
  - Negation (not): `!`
  - Disjunction (or): `!!`
  - Conjunction (and): `&&`
  - Exclusive Or: `XOR`
- Examples

```
StartCondition: ! Lookup("engine_on");
StartCondition: temp > 100 || rpm > 6000
StartCondition: score < 10 && my_turn
Assignment: result = (x > 10) XOR (y > 10)
```

Logical Expressions (cont.)

- Examples

```
True
False
CommandReceived (where CommandReceived was declared Boolean)
Lookup("Rover:initialized")
count <= 30 (where count was declared Integer)
Lookup("Rover:batteryCharge") > 120.0
! CommandReceived
Lookup("Rover:initialized") || CommandReceived
Flags[3] (where Flags is an array of Booleans)
isKnown(val) (where val is any variable)
node3.state == FINISHED && node3.outcome == SUCCESS
```

String Expressions

- Evaluate to strings
  - Literal strings (double quoted, as in “hello”)
  - Variables of type string
  - Lookups
  - String concatenation (+)
- Examples

```
"foo"
"Would you like to continue?"
Username (where Username was declared string)
Lookup("username")
"Hello, " + "Fred"     => "Hello, Fred"
"Hello, " + Username
```

- Example

```
{
  // array of 10 Booleans
  Boolean flags[10];

  // array of 6 integers, with X[0]=1, X[1]=3, X[2] = 5.
  // X[3] through X[5] are UNKNOWN.
  Integer X[6] = #(1 3 5);

  Assignment: X[3] = X[2] + 1;
}
```

Standard Plexil – World State

- Obtained through *lookups*
- Syntax: **`Lookup (<state_name>, <tolerance>)`**
  - Tolerance optional, defaults to 0
- Two execution contexts
  - “Fetch” : value immediately returned
    - Action bodies, check conditions
    - Tolerance ignored
  - “Subscribe” : current value returned, then subscribed in plan for future changes
    - Gate conditions

World State (cont.)

- Example

```
HeatRoom:
{
  StartCondition: Lookup("Temperature") < 70;
  Postcondition: Lookup("Temperature") > 70 &&
                 Lookup("Temperature") < 74
  Command: RunHeaterCycle();
}
```

Action State

- Consists of:
  - Current execution state
  - Start and end times of each execution state
  - Outcome of finished actions
  - Failure type of failed actions
  - Last command handle, for command nodes
- Accessible only for current node and its parent, children, and siblings.

Action State (cont.)

```
Root:
{
  EndCondition: Bar.state == FINISHED;
  PostCondition: Bar.outcome == SUCCESS ||
                 Foo.failure != INVARIANT_CONDITION_FAILED;
  NodeList:
    Foo: { ... }
    Bar:
    {
      StartCondition:
        Foo.command_handle == “COMMAND_ACCEPTED” &&
        Foo.EXECUTING.START > 300.0;
    }
}
```

This says: Root ends when Bar is finished; Root is successful if Bar is successful, or Foo failed while maintaining its invariant; Bar starts when Foo's command has been accepted, and Foo started executing sometime after time 300.

Translating into XML

- By convention, Plexil files have extension `.ple`
- Files must contain a *single* plan (top level action).
- Plexil files are translated into XML with the `plexilc` command

```
plexilc foo.ple
```

  - The resulting file is `foo.plx`
  - Errors and warnings will get printed if there are problems. Fix them and try again!