Resource Arbiter Safety in Plexil 4

Albert Kutsyy

June 25, 2020

Purpose

This analysis aims to identify and propose solutions to unsafe features in the Plexil 4 resource arbiter by analyzing the implementation and documentation, as well as running experiments. It does not examine the resource hierarchy feature, but does consider both commands which release resources and those that don't, as well as the possibility of commands which generate resources.

Current Implementation

Currently, command nodes can specify resources that the command consumes, and include a lower bound, upper bound, a priority, and whether the resource should be released when the command terminates. It is not clear what the lower bound is intended to be (a minimum possible resource usage wouldn't be useful because in order to execute the plan safely we need to allow for the possibility that each command will require its maximum resource consumption) and the implementation does not use this.

Each command node can specify multiple of these resource requirements.

Currently, only the priority of the first resource to be specified by the node is used and the rest are discarded.

The resource arbiter receives a list of all commands sent by a quiescence cycle (that is, all commands that can run as a result of an external signal), and does the following:

```
1. (Stably) Sort commands by the priority of the first resource in each, leaving
   equal-priority commands in the order they were sent
2. Iterate through the sorted list and attempt to subtract (or add) the requested amount of
   each resource from a log of how much of each resource is in use
3.  If each resource transaction results in a non-negative value, accept the command, else
   reject and rollback the modifications to the log
```

The implementation of this can be found in `plexil-4/src/intfc/ResourceArbiterInterface.cc`

Immediate issues

1. Having different priorities for different resources doesn't make much sense - how can a command have a low memory priority but high electrical priority (for example). Priority should be assigned to a command, not a resource.

2. Allowing different priorities for different resources but only caring about the first is dangerous - it implies that every priority matters and can easily lead to users accidentally assigning high priority commands a low priority.

3. Can the lower bound of a resource usage be usefully utilized - there is no documentation about what it is supposed to mean and in order to safely execute commands we have to assume that they take their maximum possible resources.

4. Allowing and then ignoring the lower bound is dangerous as users may attempt to use the lower bound, but it is entirely ignored.

5. Allowing the renewal of resources to counteract the use of resources within the same quiescence is dangerous. For example, if one command is `chargeBatteriesFromSolar()--> +10 power` and another is `activateDrill()--> -5 power`, both commands would be executed even we only have 2 power and the chargeBatteries command may charge after the drill consumes 5 power. Accepting these commands implicitly assumes that resource renewal commands take place before resource consumption ones.

6. Similarly, there are accepted sequences that assume resource consumption takes place before resource renewal, so if there is currently 10 power out of a maximum of 15 and `runGenerator()` gives 10 power before `activateDrill()` consumes 5 we will in fact only have 10 power (10 –> 15 (capped at 15) –> 10), but the resource arbiter will report 15.

7. Cannot use resources with synchronous commands - results in compiler error

Starvation

A plan, `Starvation.ple` (appendix A) was written to test if the resource arbiter may cause starvation - that is, have a low-priority processes prevent a high-priority process from running. In the example, three low-priority procesees each use a small portion (4.0) of a resource, running for 0.5 seconds before releasing the resource and requesting to run again. These represent small, continuous operations such as taking pictures, organizing files, and measuring temperature.

In addition, there is concurrently a high-priority process that requires the entirety (10.0) of the resource.

If there is a small command running and the high priority process attempts to execute its command, it will be rejected since there isn't enough of the resource left for the big command. However, if a small command is executed, the arbiter will allow it through since there is no higher priority command which can run and there are sufficient resources for the small command.

This leads to the high priority command being starved with the small commands cycling between each other and the big one unable to run. `StarvationFixedSequence.ple` (appendix A) demonstrates that this starvation is avoidable since there is a schedule of command execution that results in all commands able to run.

Note, this is NOT priority inversion since PLEXIL commands are not preemptive.

Deadlock

After initial experiments, it has become apparent that the constraint that only Command nodes can request resources prevents most sources of deadlock. If a command depends on another command to run before it completes, this could cause deadlock, but there is no way to specify this dependency in Plexil.

However, the restriction that only Command nodes can request resources does prevent users from implementing locks using the resource arbiter, instead needing to fallback on manual locking using variables.

Assessments and recommendations

In its current form, PLEXIL has poor concurrency support - locks have to be manually implemented, resources requirements can't be applied to a sequence of commands, and the resource arbiter can't prevent starvation.

For example, if a user wants only one of a few concurrent nodes (each with a sequence of commands) to use the arm at a time, they must implement their own locks rather than use the resource arbiter, even though the resource arbiter would be an elegant solution if each node was a single Command.

The resource support has multiple instances of syntax which it allows and then ignores - specifically different priorities for different resources and resource lower bounds.

Finally, the resource arbiter is unsafe because critical commands can fail to execute in environments with frequently executing low priority commands (starvation), which is a major safety concern.

The recommendations are as follows:

1. Different priorities are possible for each resource, all but first are ignored [High priority]:
   (a) Rather than taking the priority of the first resource, take the maximum priority (lowest value) of the resources. This doesn't resolve the issue that many priorities are allowed but only one is used, but resolves one potential source of bugs (users thinking a command is high priority while the resource arbiter assigns it a lower priority base off of the first resource). [Quick fix]
   (b) On compile time, throw an error if the priorities for the resources are different. This imposes seemingly unnecessary syntactic restrictions (one needs to write out a priority for each resourcem but they must be the same), but is clear and explicit. [Quick fix]
   (c) Rather than associating a priority with each resource requested, require that a Priority statement be included in any Command node using resources, such as

```
Execute: {
    Resource Name = "sys_memory",UpperBound=4.0;
    Resource Name = "camera_control";
    Priority 15;
    out = ExecuteCommand123();
  }
```

This is the cleanest fix, but requires a change in the syntax. [Best solution]

2. Lower bound is allowed and ignored [Low priority]:

   (a) Don't allow lower bounds - there is no reason they should exist, since using them for resource consumption estimation is unsafe. [Best Solution]

   (b) If there is a specific reason to leave lower bounds - perhaps some sort of execution tracing - explicitly detail this on the documentation while still warning users the lower bound has no impact on resource allocation.

3. Resource arbiter makes implicit assumptions that concurrent resource consumption and generation can't exceed limits if their sum can't (that an operation consuming 5 can occur concurrently with an operaton producing 5 and the value of the resource will never change) [High priority]:

   (a) Rather than simply keeping track of the "current" amount of a resource, maintain three counters per resource - the current value, the utilization, and the generation.

   Mandate that current-utilization ≥ 0 and current+generation $\leq$ maximum. When a new command comes in, check that adding the utilization/generation for each resource would cause no resource to exceed these bounds. That is, if the command consumes 5 of A and generates 10 of B, check that current_A - utilization_A - $5 \geq 0$ and current_B + generation_B + $10 \leq$ max_B. If so, accept the command and update the utilization and generation values accordingly (but NOT the current value).

   This, of course, assumes that we should simply reject any command which would exceed the maximum amount of the resource. If different behavior is desired, this algorithm should be adjusted accordingly.

   When a command succeeds, fails, or is aborted, this must be forwarded to the resource arbiter. If the command releases resources on termination, the arbiter should simply remove the relevant amounts of each resource from the utilization/generation counts. If the resource does not release on termination, the amount should be removed from the utilization/generation counts and subtracted/added to the current value, respectively. This assumes that all commands will consume resources in the worst possible order, ensuring safety. [Best Solution]

      i. Note that extensive clear documentation and examples would be necessary to make this behaviour clear to users, since it does not cleanly fall into a mental model of "consumption comes before generation" or vice versa.

4. Cannot use resources with synchronous commands (a major use case of resources) [High priority].

(a) Copy the resource clauses into the generated Command node at compile time. [Best solution]

5. Lack of straightforward concurrency support for non-command nodes [Medium priority]

   (a) This is a large undertaking, since the resource arbiter is only built to take commands. However, if the same resource syntax could be used for sequences (with appropriate deadlock prevention such as Two-Phase-Locking), this would make PLEXIL much more usable for safe concurrency.

   This could be done by compiling a node with resources into a sequence node that begins with a Command node that executes a dummy command with the required resources, and ends with another Command that signals to complete the first command.

6. Starvation [High Priority]

   (a) While this is unavoidable if we use the syntax that commands are immediately accepted or rejected by the resource arbiter, it can be avoided by adding the option (like the proposed Priority option) wait_for_resources. For commands with this option specified, rather than immediately rejecting the command if the resources aren't available, the resource arbiter will add the command to a queue (ordered by priority, then arrival time) of waiting commands. The command would then be rejected under only two circumstances - the requested amount of resources is more than the maximum possible, or the timeout expires.

   Arriving commands which have a lower priority than the command at the head of the queue and would result in there not being sufficient amounts of *any* resource for the command to run are rejected. When there are sufficient resources for the command at the head of the queue to execute, it is executed and removed from the queue. Note, this algorithm would not permit commands which consume a small number of requested resources to generate large amounts of others (potentially creating deadlock), so alternatives should be considered.

   In the starvation example, first GetTemperature() would be proposed to the resource arbiter and accepted, then CalculateLanding() and OrganizeFiles() would be proposed. If the wait_for_resources option is specified for CalculateLanding(), it would be enqueued, and OrganizeFiles() would be rejected since if all running commands were to finish and OrganizeFiles() were to continue to run, it would stop CalculateLanding() from executing. Similarly, TakePicture() would be rejected in the next quiescence. When GetTemperature() terminates, CalculateLanding() would be dequeued and run since it is the highest priority operation able to execute.

   (b) Alternatively, we can add a "block_interfering" option to a command which would specify that when this command is being considered by the Resource Arbiter, block other commands which would interfer with it, in the manner described in (a).

   Combined with retrying every cycle, this would stop starvation while letting the user have more visibility and control. However, this runs into the same deadlock issues as (a) and would require more work on the user's end, since even rarely failing to retry could result in starvation.

      i. This design pattern could be included in a macro, similarly to the SynchronousCommand macro.

(c) Please note that both suggestions introduce semantic concerns which are beyond the scope of this report, please see the following comment by Dr. Jeremy Frank:
The 'semantic interpretation' of 'command wasn't issued because not enough resources available because higher priority commands were granted the resource' requires some more thought. This is why the rest of the PLEXIL team should weigh in.

I'm not sure what to think about rejecting (failing) a command because other commands of higher priority took the resources away. On one hand, it's better than bailing out of the whole plan. The problem of diagnosing 'why did the command start so much later than expected' is similar. In both cases, the explanation is resource contention at excution time. On the other, it may strike users as somewhat cryptic if a command fails once in a thousand (or less) times.

Handling either semantics correctly requires looking in more detail at the condition evolution and node execution state diagrams.

- If the resource requirement really is some kind of start-condition, then it's important to update the node's start_condition internal variable only when the Resource Arbiter says so. If the intended semantics of not enough resource means the start condition isn't satisfied, then you do nothing, but the command node's other conditions need to be updatable during the next quiescence cycle, and if this command node fails or is aborted or whatever, then you have to remove this node from the Resource Arbiter's queue.

- If instead insufficient resource leads to node failure, you would update node state to Failing. (In other words, it's not a failure of the Start Condition at all.)

See the relevant state transition diagrams linked below.
http://plexil.sourceforge.net/wiki/index.php/Detailed_Semantics#Node_Transitions
http://plexil.sourceforge.net/wiki/index.php/Node_State_Transition_Diagrams#FAILING_state

Should we make a more significant change to PLEXIL syntax and allow Start_Conditions and possibly other conditions to explicitly reference resource availability? The idea is to let those conditions dictate the two semantic interpretations we are batting around above for resources. The Precondition of a PLEXIL node is an instantaneous check prior to execution; if it is false, the node does not execute (instead going to Failure). So if you wanted the command to 'persistently retry' until resource becomes available you would make a StartCondition(resource); if you wanted it to fail if resource wasn't available, you would instead say PreCondition(resource). Such a change is probably too significant. Since Resource declarations are only in Commands, more syntax checking would be needed to ensure the resource-centric conditions are only in Command nodes. However, the syntax of the Resource declaration in the Command node could reflect how lack of resource availability should be handled. An example: use a Boolean flag, FailIfDeferred. If this flag is set to True, then low priority commands will fail if the resource is used by higher priority commands; otherwise, it will wait and try again.

```
Resource
  // required
  Name = <String expr>,
  Priority = <Integer expr> // smallest value = first priority
  Bound = <Real expr> // default=1.0
  FailIfDeferred = <Boolean expr> // default = false
  //optional
  , ReleaseAtTermination = <Boolean expr> // default = true
  ;
}
```

Appendix A

Starvation.ple

```
// This plan models several small operations with low priority that starve a high priority
   process
Integer Command GetTemperature();
String Command OrganizeFiles();
String Command TakePicture();
Command SendMessage(...);

String Command CalculateLanding();
Command pprint(...);

Starvation:
Concurrence
{
  Boolean continue = true;
  Integer temperature;
  Small1: {
    Integer returnValue;
    RepeatCondition continue;
    SkipCondition !continue;
    Retrieve: {
      EndCondition isKnown(returnValue);
      Resource Name = "sys_memory",UpperBound=4.0, Priority = 30;
      returnValue = GetTemperature();
    }
    AssignTemp: {
      temperature = returnValue;
    }
  }
  Small2: {
    String out;
    StartCondition isKnown(temperature);
    RepeatCondition continue;
    InvariantCondition continue;
    SkipCondition !continue;
    EndCondition isKnown(out);
    Resource Name = "sys_memory",UpperBound=4.0, Priority = 50;
    out = OrganizeFiles();
  }
  Small3: {
    String out;
    RepeatCondition continue;
    InvariantCondition continue;
```

```
    SkipCondition !continue;
    EndCondition isKnown(out);
    Resource Name = "sys_memory",UpperBound=4.0, Priority = 40;
    out = TakePicture();
    }

  Big: {
    Execute:{
      String out;
      StartCondition isKnown(temperature); // Don't immediately start
      EndCondition isKnown(out);
      RepeatCondition Self.command_handle == COMMAND_DENIED || Self.command_handle ==
          COMMAND_FAILED;
      Resource Name = "sys_memory",UpperBound=10.0, Priority = 1;
      out = CalculateLanding();
    }
    QuitSimulator:
    {
      SendMessage("Quit");
      continue = false;
    }
  }
}
```

StarvationFixedSequence.ple

```
// This plan models several small operations with low priority that starve a high priority
    process
Integer Command GetTemperature();
String Command OrganizeFiles();
String Command TakePicture();
Command SendMessage(...);

String Command CalculateLanding();
Command pprint(...);

Starvation:
Sequence
{
  Boolean continue = true;
  Integer temperature;
  SmallTasks: {
    Small1: {
      Integer returnValue;
      Retrieve: {
        EndCondition isKnown(returnValue);
        Resource Name = "sys_memory",UpperBound=4.0, Priority = 30;
        returnValue = GetTemperature();
      }
      AssignTemp: {
        temperature = returnValue;
      }
    }
    Small2: {
      String out;
      StartCondition isKnown(temperature);
      EndCondition isKnown(out);
      Resource Name = "sys_memory",UpperBound=4.0, Priority = 50;
      out = OrganizeFiles();
    }
    Small3: {
      String out;
      EndCondition isKnown(out);
      Resource Name = "sys_memory",UpperBound=4.0, Priority = 40;
      out = TakePicture();
    }
  }



  Big: {
```

```
    Execute:{
      String out;
      StartCondition isKnown(temperature); // Don't immediately start
      EndCondition isKnown(out);
      RepeatCondition Self.command_handle == COMMAND_DENIED || Self.command_handle ==
          COMMAND_FAILED;
      Resource Name = "sys_memory",UpperBound=10.0, Priority = 1;
      out = CalculateLanding();
    }
    QuitSimulator:
    {
      SendMessage("Quit");
      continue = false;
    }
  }

  //Never executes, but necessary to demonstrate equivelance with Starvation.ple
  SmallTasks2: {
  SkipCondition !continue;
    Small12: {
      Integer returnValue;
      RepeatCondition continue;
      Retrieve: {
        EndCondition isKnown(returnValue);
      Resource Name = "sys_memory",UpperBound=1, Priority = 30;
      returnValue = GetTemperature();
      }
    AssignTemp: {
   temperature = returnValue;
      }
      }
    Small2: {
      String out;
      RepeatCondition continue;
      EndCondition isKnown(out);
      Resource Name = "sys_memory",UpperBound=1, Priority = 50;
      out = OrganizeFiles();
    }
    Small3: {
      String out;
      RepeatCondition continue;
      EndCondition isKnown(out);
      Resource Name = "sys_memory",UpperBound=1, Priority = 40;
      out = TakePicture();
    }
  }
}
```

Simulator.ple

```
// This plan emulates a simulator that handles:
// Integer GetTemperature()
// TakePicture()
// OrganizeFiles()
// CalculateLanding()
// - the message 'Quit', which closes the simulator (CAVEAT: not always cleanly)

Command UpdateLookup (...);
Command SendReturnValue (...);
Command pprint(...);

Integer Lookup time;

Simulator: Concurrence
{
  Integer temperature = 1;
  Boolean continue = true;
  ExitCondition !continue;

  HandleTemperature:
  {
    RepeatCondition continue;
    ReceiveTemperature: OnCommand "GetTemperature" ()
    {
      RespondTemperature: SendReturnValue(temperature);
      Wait 0.5; //Wait half a second so commands aren't instantanious
    }
  }

  HandlePicture:
  {
    RepeatCondition continue;
    ReceiveTakePicture: OnCommand "TakePicture" ()
    {
      String s = "nothing";
      SendReturnValue(s);
      Wait 0.5;
    }
  }

 HandleOrganizeFiles:
  {
    RepeatCondition continue;
    ReceiveOrganizeFiles: OnCommand "OrganizeFiles" ()
    {
```

```
      RespondOrganizeFiles: SendReturnValue("nothing");
      Wait 0.5;
    }
  }

  HandleLanding:
  {
    RepeatCondition continue;
    ReceiveLanding: OnCommand "CalculateLanding" ()
    {
      RespondLanding: SendReturnValue("nothing");
    }
  }

  HandleQuit:
  {
    RepeatCondition continue;
    ReceiveQuit: OnMessage "Quit" Set: continue = false;
  }
}
```