Tag: Job Control in urbiscript

Jean-Christophe Baillie Akim Demaille Quentin Hocquet
Matthieu Nottale

May 7, 2010

Abstract

The Urbi software platform for the development of applications for robots relies heavily on concurrency. While concurrent programs have existed for decades, there is still no universal consensus for the proper means to control the complexity of parallelism. This paper presents some innovating features of Urbi: syntactic constructs to support concurrent jobs, and *tags*, used to control them.

Urbi SDK is a software platform used to develop portable applications for robotics and artificial intelligence (Baillie, 2005). It is based on UObjects, a distributed component architecture, and on urbiscript, a parallel and event-driven scripting language. It's current architecture has already described elsewhere (Baillie et al., 2008), this paper focuses on one specific feature of urbiscript: job control.

Today, any computer runs many programs concurrently. The Operating System is in charge of providing each process with the illusion of a fresh machine for it only, and of scheduling all these concurrent processes. In robots, jobs not only run concurrently, they *collaborate*. Each process is a piece of a complex behavior that emerges from the coordination/orchestration of these jobs.

The Urbi platform aims to provide the developers with a convenient framework to orchestrate concurrent, collaborating, tasks.

Outline Section 1 introduces some of the features of urbiscript for parallel programming; it concludes on the common need for a high-level job control. Section 2 presents *tags*, the cornerstone of job control in Urbi. A real world example is presented in Section 3: safe interruption of the Nao walk. Section 4 concludes.

1 Concurrency in urbiscript

Internally, to support concurrency, Urbi relies on coroutines. *Coroutines* are light threads (aka green threads, or fibers) with their own stack and execution pointer implemented as a library. They are commonly used in modern interpreted languages such as Io (Dekorte, 2005). They allow the use of the Urbi kernel on architectures (e.g., small robots) whose operating system does not provide support for multiple processes or kernel threads.

This section presents some urbiscript constructs for concurrency. These features are also available to user code written in some low-level compiled language such as C++. It requires to use the UObject API, and, since coroutines are not preemptive, some collaboration with the Urbi scheduler. Alternatively, UObject can be launched in their own thread.

1.1 Flow Control

Classical sequential programming languages, such as C or C++, provide a single statement separator, '`;`': `a; b` stands for "run `a` and then `b`". urbiscript provides additional connectors. The '`,`' connector launches the first statement in background, and immediately proceeds to executing the next statements. Scopes

(statements enclosed in curly braces: `{ s1; s2, ...}`), are boundaries: a compound statement "ends" when all its components did. The following example demonstrates these points[1].

```
{
  { sleep(2s); echo(2) },
  { sleep(1s); echo(1) },
};
echo(3);
[00001451] *** 1
[00002447] *** 2
[00002447] *** 3
```

The '`&`' connector is similar to '`,`' in that it launches jobs in parallel, yet, contrary to '`,`', it "waits" for all the statements to be known to start their concurrent execution. The difference is virtually insignificant in scripts, but it is very noticeable in interactive sessions: entering `echo(1),` will immediately print `1`, while `echo(1)&` will wait for the right-hand side statement to be complete (i.e., to actually end with either a '`;`' or a '`,`') to fire the execution of the compound statement.

Once concurrency-oriented statement-separators are introduced, some control flow constructs naturally follow the same extension pattern. In particular loops, sequential by default, can be generalized to become parallel. For instance, iterating over a collection comes in several flavors: **`for`** is sequential while **`for&`** launches all the iterations concurrently (see below).

```
for  (var i : [2, 1, 0])
{
  echo("%s: start" % i);
  sleep(i);
  echo("%s: done" % i)
};
echo("done");
[00125189] *** 2: start
[00127190] *** 2: done
[00127190] *** 1: start
[00128192] *** 1: done
[00128192] *** 0: start
[00128193] *** 0: done
[00128194] *** done
```

```
for& (var i : [2, 1, 0])
{
  echo("%s: start" % i);
  sleep(i);
  echo("%s: done" % i)
};
echo("done");
[00105789] *** 2: start
[00105789] *** 1: start
[00105789] *** 0: start
[00105793] *** 0: done
[00106793] *** 1: done
[00107793] *** 2: done
[00107795] *** done
```

These operators cover a large part of common needs to compose computations with sequential and/or concurrent components. Yet they fall short when jobs must be created dynamically, or if, for instance, their lifetime must escape the simple rules of scope nesting. To this end, urbiscript provides means to create new jobs.

1.2 Dynamic Jobs

The `detach` function takes a single argument: a block of code that will be run in its own thread of execution. It returns a handle to this job, which can be used to query the status of the job.

```
//#line 1 "foo.u"
var job = detach( { sleep(1s); echo(1) } );
[00000214] Job<shell_5>

job.dumpState;
[00000219] *** Job: shell_5
[00000219] ***   State: sleeping
[00000219] ***   Tags:
[00000219] ***     Tag<U0x1004043e0>
[00000221] ***   Backtrace:
```

[1] In urbiscript interactive sessions there is no prompt: lines starting with a time stamp such as `[00001451]` are output by Urbi. The other lines were entered by the user.

```
[00000221] ***     foo.u:1.21-28: sleep
[00000221] ***     foo.u:1.11-41: detach

sleep(1s);
[00001219] *** 1

job.dumpState;
[00001222] *** Job: shell_5
[00001222] ***   State: terminated
[00001222] ***   Tags:
[00001222] ***     Tag<U0x1004043e0>
[00001225] ***   Backtrace:
[00001225] ***     foo.u:1.11-41: detach
```

This handle provides a first simple means to control the job.

```
//#line 1 "bar.u"
var job = detach( { sleep(1s); echo(1) } );
[00000214] Job<shell_9>
job.terminate;
job.status;
[00008206] "terminated"
sleep(1s);
// Nothing printed, the job was killed.
```

But handles are not different from POSIX thread handles for instance. They are low-level, error prone (Ousterhout, 1996), and provide few features.

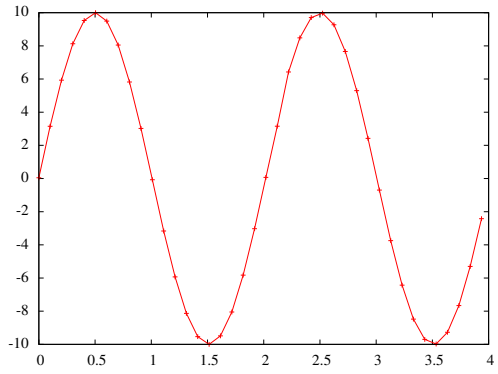
1.3 Trajectory Generators

It is occasionally useful to continuously update a variable, sort of binding the variable to a function whose value depend on time. This can be used to assign trajectories to motors, for instance to make the neck of the robot oscillate to browse the world to look for a specific object, or simply to program a movement.

To this end, urbiscript provides trajectory generators. They look like "continuous" assignments:

```
var y = 0;
y = 0 sin:2s ampli:10,
```

means "repeatedly set `x` to the value of a sine centered on 0, with a period of 2 seconds, and an amplitude of 10". Since it is "continuous", we use the '`,`' separator in order to background this statement (see the figure on the right-hand side).



Of course, some means must be provided to control this assignment, for instance to interrupt the oscillation of the neck once the object of interest is found.

1.4 Event Watchers

urbiscript code —and actually even C++ user code managed by Urbi— supports event-driven programming. Some specific constructs allow to arm event-watchers that remain idle until some condition is verified, in which case they fire their attached code.

For instance, the **`at`** construct waits for some *event* to be triggered (using the `!` operator in urbiscript) to execute their body. Lengthy body executions can be concurrent (see the left-hand side of Figure 1).

In other frameworks, similar features are provided by callbacks, but their interface is cumbersome, and they are typically poorly extensible: callbacks cannot be added freely from anywhere in the code, and

```
var e = Event.new;
[00094489] Event_0x109354cd0
at (e?(var x))
{
  echo("%s: start" % x);
  sleep(x);
  echo("%s: done" % x);
};
e!(2);
[00094490] *** 2: start
e!(1);
[00094501] *** 1: start
sleep(2s);
[00095511] *** 1: done
[00096510] *** 2: done
```

```
var x = 0;
[00248475] 0
at (x == 1)
  echo("x is one!");
x = 1;
[00266201] 1
[00266201] *** x is one!
x = 2;
[00272634] 2
x = 1;
[00275081] 1
[00275081] *** x is one!
```

Figure 1: Watching changes: events and expressions

they hardly can be attached to conditions that were not declared to be "events" beforehand. In urbiscript, expressions can be watched too, without having to bless the used variables in any particular way (see the right-hand side of Figure 1).

Many Urbi features, including the ones presented in this section — concurrent flow control constructs, dynamic jobs, trajectories, and event watchers — allow to launch concurrent tasks. Some features, such as events or scopes, suffice to control them in common cases. Yet, similarly to exceptions that allow to disrupt the execution flow of a sequential program, one needs high-level features to manage jobs. Urbi introduces the concept of *tag* to this end.

2 Job Control in urbiscript

Section 1 introduced some of the means to perform concurrent tasks. This section describes how urbiscript proposes to manage them.

2.1 Tags

2.1.1 Simple Tags

Tags are regular urbiscript objects: they must be created before being used: `var t = Tag.new("sample")`. Statements can be tagged: `t : stm`. Then tags can be used to manage the tasks; for instance `t.stop` interrupts any running piece of code tagged by `t`. This is not unlike signals in Unix, but from the job point of view, this corresponds to the raising of an exception.

```
var t = Tag.new;
[00000010] Tag<tag_8>

t : every (1s) echo("tick!"),
[00000019] *** tick!
sleep(0.5s);
t : every (1s) echo("tack!"),
[00000020] *** tack!
sleep(2s);
[00000021] *** tick!
[00000022] *** tack!
[00000023] *** tick!
[00000024] *** tack!
t.stop;
sleep(2s);
// Nothing else runs.
```

Tags can also be used on expressions (i.e., constructs that have a value), in which case, `stop` takes the forced result as argument. Stopped tags can be used to label new jobs: the following example reuses the previous `t`.

```
// A very long, erroneous, computation.
var large = { t: { sleep(2**1024); 51 } },
// A simpler computation.
var simple = { t: { sleep(0s); 6 * 7 } },

// Wait for the first of 'simple' and 'large' that has a value.
waituntil (!simple.isVoid || !large.isVoid);

// Force the result of all the still-running computations.
t.stop({if (!simple.isVoid) simple else large});
echo(large);
[00000222] *** 42
```

2.1.2 Hierarchical Tags

Tags can also be hierarchical: stopping a tag also stops the children tags. This enable to create hierarchical behaviors, where you can control sub behaviors individually, or the whole behavior directly.

```
var party = Tag.new;
party.addChild("dance");
party.addChild("sing");

function robot.party()
{
  party.dance: { /* dance behavior */ }
  &
  party.sing: { /* sing behavior */ }
};

party.dance.stop; // Stop dancing
party.sing.stop;  // Stop singing
party.stop;       // Stop both
```

2.2 Enter/leave

Another useful aspect of events is how they enable to run code upon entering and leaving the tagged block.

2.2.1 Usage

Every tag features the *enter* and *leave* events, fired when the "tag is entered or left".

```
var tag = Tag.new;
at (tag.enter?)
  echo("I'm in!");
at (tag.leave?)
  echo("I'm out!");

tag: echo("I'm there!");
[00000000] *** I'm in!
[00000000] *** I'm there!
[00000000] *** I'm out!
```

This feature has two main pros over typing directly the enter and leave code around the tagged block. First, it enables you to deport and factor a scoped behavior and reuse it:

```
var withGas = Tag.new;
at (tag.enter?)
  turnGasOn();
at (tag.leave?)
```

```
  turnGasOff();

withGas: bakeMeat();
withGas: cookPasta();
```

The other major advantage is that it *guarantees* that the exit code is run, in an exception, stop, flow control, ... safe way. In the following code, if we exit the function through the exception or the `return` statement, we won't shut the gas off as expected:

```
function cook()
{
  turnGasOn();
  {
    if (!recipe)
      throw Exception.new("Missing recipe!");
    if (alreadyCooked)
      return true;
    doStuff();
    return true;
  };
  turnGasOff();
};
```

On the other hand, using our tag prevents this problem. Leaving the scope through an exception or any flow control construct will still run the *`leave`* behavior.

```
function cook()
{
  withGas: // We're safe
  {
    if (!recipe)
      throw Exception.new("Missing recipe!");
    if (alreadyCooked)
      return true;
    doStuff();
    return true;
  };
};
```

Even stopping a piece of code will run the *`leave`* behaviors. Note that the *`stop`* statement will synchronously block until all tagged code has stopped, and thus until all *`leave`* behavior have been run.

```
at (tag.leave?)
{
  sleep(2s);
  echo("leave");
  sleep(2s);
};

tag: sleep(10s);
echo("before");
tag.stop;
// The leave behavior is finished before reaching this line.
echo("stopped");
// See the timestamps.
[00000000] *** before
[00002000] *** leave
[00004000] *** stopped
```

Therefore we can safely stop our cooking process:

```
tag: cook(), // Note the comma: cook in the background.

sleep(10ms);
tag.stop;
// Here, we're sure we're done cooking and the gas is off.
```

2.2.2 An application: Mutual Exclusion

An application of this feature is the definition of easy to use, safe mutexes. In Urbi, mutexes are actually tags (that use a semaphore internally). Tagging a block will lock/unlock the mutex when entering/leaving it.

```
var mtx = Mutex.new;

// Ensure critical sections are mutually exclusive.
every (200ms)
  mtx: criticalSection1();
at (someEvent?)
  mtx: criticalSection2();
```

This provides an elegant and concise way to use mutexes. Moreover, it ensures the mutex is unlocked when leaving the critical section, even through exceptions or flow control, thus avoiding stray locks and resulting dead locks. As an additional feature, since mutexes are tags, anyone can use `mtx.stop` to make any code currently holding the mutex release it.

3 Application: Controlling Nao Walk

An application of this concept is used in Urbi for Naoqi: the Urbi interface provides a `robot.walk` function, which makes the robot walk the given distance forward. The call is *synchronous*: it returns only when the robot is done moving. This enables to easily chain actions:

```
// Walk 10 meters, then say 'Hi'.
robot.walk(10);
robot.say("Hi!");
```

However, in the Urbian way, you should be able to change your mind and stop any behavior, including a currently running walk. However, the walk should not be abruptly stopped, leaving the robot in an unstable position, since it would certainly fall. To address this, we can use tags to run a behavior when the `walk` function is stopped.

```
{
  var movement = Tag.new;
  at (movement.leave?)
  {
    // Ask Naoqi backend to reset current movement.
    motion.ClearFootSteps;
    // Wait for Nao to be immobile.
    motion.WaitUntilWalkIsFinished;
  }

  function robot.walk(distance)
  {
    movement: {
      // Ask Naoqi backend to move forward.
      motion.walkTo(distance, 0, 0);
      motion.WaitUntilWalkIsFinished;
    }
  }
}

tag: robot.walk(1000),
sleep(10s);
tag.stop;

// When this line is reached, we have the guaranty Nao is stopped, and
// stable on its feet.
```

Another question is, what to do if two incompatible orders are given? For instance:

```
robot.walk(1),
sleep(500ms);
robot.walk(-1);
```

If we want the second order to wait until any previous movement is finished, we can simply change our `movement` tag into a Mutex. If we want the second order to override any current movement, we can simply call `movement.stop` at the beginning of our `walk` function, thus stopping any currently running movement.

4 Conclusion

This paper presented some urbiscript constructs for concurrency. It describes how controlling concurrency is achieved using *tags*, which are far more than job ids, or job groups: they are a high-level means to specific how jobs can interact (for instance to implement mutual exclusion), in addition to providing simple job control.

References

Baillie, J.-C. (2005). URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825.

Baillie, J.-C., Demaille, A., Hocquet, Q., Nottale, M., and Tardieu, S. (2008). The Urbi universal platform for robotics. In Noda, I., editor, *First International Workshop on Standards and Common Platform for Robotics*.

Dekorte, S. (2005). Io: a small programming language. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 166–167, New York, NY, USA. ACM.

Ousterhout, J. K. (1996). Why threads are a bad idea (for most purposes).