Testing Physics Engines with Live Robot Programming

Johan Fabry
PLEIAD and RyCh labs,
Computer Science Department (DCC),
University of Chile

Stephen Sinclair
Inria Chile

June 10, 2016

Abstract

Existing tests for physics engines in robotics simulators are not sufficient to address the concerns of roboticists: is the simulation physically faithful to the real world? Would a change of use of the engine yield better results? To answer these questions, unit tests of real world situations that can be performed on different engines are needed. To help to quickly and efficiently develop such tests, we developed a methodology and tool we call Live Tests for Robotics (LT4R). It consists of the use of a live programming language for developing state machines while interacting with simulation recordings. In this text we describe LT4R, show test examples for simple dynamical system behaviours as well as a plain gripper simulation, and argue why such tests could as easily be developed by a roboticist user, as by a physics engine implementor.

1 Introduction

The availability of good tests is a key factor for high-quality software [20], yet typically developers spend a low amount of time on writing and maintaining tests [1]. Software testing is also key in the robotics domain since there is a wide variety of software in use in robotics. One such kind of software is physics engines in robotic simulation software, such as the SimSpark simulation used in the Robocup Simulation league[1] or the Open Dynamics Engine (ODE) physics engine for the DARPA Virtual Robotics Challenge [11].

Noted conspicuously in the DARPA challenge rules [5] is that physics engines should be physically faithful to real world behaviour. Yet empirical evidence shows that this is not always the case. For example, we found that using the current version of the Gazebo robotics simulator [14] (version 7) with default settings, if we drop a ball its behaviour is radically different depending on the

[1] `http://wiki.robocup.org/wiki/Soccer_Simulation_League`

physics engine used. In the ball example, the same setup will, with ODE[2], cause the ball to bounce when it hits the floor, while with Bullet[3] the ball does not bounce at all. Whether it is fundamental to the engine or just a question of settings, without drawing judgement we can nonetheless conclude that at least one of the two engine behaves incorrectly.

Writing tests for physics engines is however not straightforward. This is because most scenarios cannot easily be compared to an ideal solution, and real-world behaviour is expected to further diverge from simulation. As a result, typically, physics simulation tests are tailored to specifics of the simulation software under test. A testament to this is the scarce availability of research results quantitatively comparing different physics engines [2, 6, 12].

The roboticist however is not interested in the specifics of a given simulator. Instead, the question is if the simulation of a scenario that he or she is working on is physically faithful to the real world, or if the use of a different physics engine would yield better behaviour. Yet existing tests do not reflect this need.

Instead, what is required are unit tests of real world situations to establish overall correct behaviours, and it should be possible to carry out these tests on several physics engines. Moreover, such tests should be easy to write, to help overcome developer aversion to writing them and ideally to also allow the roboticist to define the tests required for the scenario being worked on.

The above are therefore the goals for the methodology and toolset that we present in this paper, called Live Tests for Robotics (LT4R). The research contributions of LT4R are as follows:

- We propose the use of unit tests, even multiple tests for the same simulation data, each testing a specific characteristic of an overall behaviour.
- We define a programming language and environment for efficiently building such unit tests.
- We treat different physics engines as a black box and only consider the output, so that a roboticist can apply the test to different physics engines.
- We propose the paradigms of live programming and state machines to significantly ease the writing of tests, the extension of tests and the experimentation with test variants.

2 Physics Engines in Robotics

Robotics is a multidisciplinary field distributed between challenges in mechanical and electrical design, manufacturing, and algorithm design at various levels of control, from PID design to path planning, decision making, and much more.

While design and construction of real robots that perform work is generally thought of as the end goal of research in this field, it has long been considered

[2] `http://ode.org`
[3] `http://www.bulletphysics.org`

that in many cases computer simulation can provide a convenient proxy for working with real mechanical devices. This approach comes with many advantages, since it decouples the development of control-related solutions from the design and development of real machines, which has its own and mostly disjoint set of engineering challenges.

Simulation can be useful in offline and online scenarios:

- *Offline simulation* can be used to develop and optimise control strategies, to study empirical relations between physical quantities, and to evaluate mechanical configurations during design stage (e.g. evolutionary robotics, gait generation, etc.)

- *Online simulation* further makes possible applications in predictive control, testing of devices in hazardous or difficult scenarios such as in space applications or nuclear facilities, or observing simulated self-driving vehicles with a faster and cheaper development-test cycle as compared to testing real or model vehicles [9].

- *Real time interactive simulation*, an extension of online methods, allows to extend real-time controls of simulated mechanisms to a user interface, allowing virtual telerobotics for hands-on testing of scenarios such as manipulation tasks [16]. This can be used in training applications, e.g. for medical professionals, avoiding danger to patients [18].

Some other reasons why simulation can be a prefered strategy are to avoid incurring large costs that might be associated with risking real hardware during testing, and similarly, lowering the cost of entry to robotics research by opening the doors to algorithm design for teams that cannot afford expensive robots.

The role of simulation has been acknowledged widely, notably in the form of the Virtual Robotics Challenge issued by DARPA in 2013, for which the Gazebo simulator running the Open Dynamics Engine (ODE) was selected as the competition physics engine [11]. Tasks included controlling a walking robot, having it sit in a car and drive, and grasping and manipulating a fire hose.

However, for 'in-simu' development to be transferable to real-world robotics, a simulation engine must of course be physically faithful to real world behaviour [5]. This goes both for simulation of the robot mechanism itself, as well as for the simulated environment with which it interacts [14]. In one survey of robotics simulation engines, physical accuracy was the most requested feature [13]. In the Open Source Robotics Foundation's 2014 online survey on Gazebo, it was similarly found that "physics validation" was the highest-voted topic [8].

Despite many advances in the area of physics simulation, there remain nonetheless some serious challenges even in simulating rigid body interaction with contact and Coulomb friction, leaving aside questions related to more complex physical phenomena such as fluid mechanics and non-ideal sensors. Although there are many reasons for this, one that can be summarized briefly is that apart from a small set of known closed-form solutions, most problems involving contact and friction cannot, or cannot easily, be tested against an

ideal solution. Despite this, there is often an expected behaviour that one can validate by 'eye-balling' the results.

It is for this reason that in this work we propose a methodology to flexibly develop tests for evaluation and comparison of physics simulations. We hope that this will allow for the application of testing methodology to physics engine development that can take into account characteristics of behavioural results designed on a per-case basis, instead of (or to complement) the often-used method of comparing RMS error along a motion path. We suggest that the design of a large number of simple and minimal tests based on behaviour and outcomes can help to evaluate the robustness and adequacy of a simulation to a specific benchmark or task.

One problem with the motion path error approach, apart from the lack of ground-truth, (which is often simply generated by setting the timestep to a very small value, providing only an "internal consistency" approach [6]), is that due to the integration process, small differences early in the simulation may lead to an accumulation of error, skewing the evaluation when one is more interested in whether the overall behaviour is correct—i.e., did the hand grasp the object; did contact restitution conserve energy; did the car steer towards the goal.

There are some examples of this approach in previous literature. For example, in their comparison of 5 simulation engines, Erez et al. proposed using short-time motion path error, in itself a unique and interesting idea, but ultimately for their grasping task, measured simply whether the object was successfully grasped for the duration of the simulation [6]. They proposed the development of a series of standard tests in a basic description format such as their own MJCF, or the URDF[4] format used by the Player simulator.

In another example, Gowal et al. evaluated a vehicle simulator based on a simple measure of how far the ending position was from the desired position, ignoring details of the full motion path [9]. Castillo et al. suggested taking advantage of the aforementioned error accumulation by only looking at the final position of actors [4].

Peters and Hsu proposed the development of a series of standardized tests for physical properties, such as angular momentum or friction, using arrays of boxes instantiated with a range of parameters [15]. However, in this case, they used a motion path metric to compare accuracy of several physics engines. They argue that small scale tests such as this can complement larger simulations such as a full walking robot, isolating individual strengths and weaknesses of simulators.

Although these small tests in URDF format (for example) provide a good starting point, they only contain a scene description and initial parameters—evaluation is still left up to the implementer. Relatedly, the principle disadvantage with our proposed method, as compared to the motion path approach, is that custom metrics would be needed for individual tests. Thus, in this work, we propose one methodology that we believe can be used to quickly develop such tests with a minimum of fuss. This is thanks to the specification of the tests as state machines in a live programming language.

[4]Unified Robot Description Format

3 Live Robot Programming

Live Robot Programming (LRP) is a programming language for the specification of behaviours as nested state machines [7]. The language follows the paradigm of live programming [19]: allowing for the direct construction, visualization and manipulation of the program's run-time state. As a result of this directness, changes in program code are immediately reflected in the running machines. This results in the development cycle being reduced to the minimum since the time gap between programming a behaviour and observing it in action has been removed. Also, if the machine is in an inappropriate state to see the effect of an update, the environment allows to explicitly set the currently active state, i.e. 'jump' to a specific state at the command of the programmer. Al this support allows for rapid development and modification of behaviours, enabling fast adaptation of existing robot behaviours to new contexts as well as cheap experimentation with new behaviour (variants).

Live programming is not a new idea. The first work on live programming was by Tanimoto on Viva [19], a dataflow language for image processing. In the context of robotics, the only other work using live programming is the Flogo language by Hancock [10]. It is focused on teaching programming of robots to children, and also follows the dataflow paradigm. In contrast, the focus of LRP is autonomous robots in research and industry, and it uses the state machine paradigm. To the best of our knowledge LRP is the only live programming language using the state machine paradigm.

LRP is not tied to a specific robot API: it currently supports programming behaviours in ROS [17], the Lego Mindstorms EV3[5], and the Parrot AR.Drone[6]. Moreover, LRP is not fundamentally restricted to the development of robot behaviours: any type of behaviour that can be specified as a nested state machine can be programmed in LRP. The only requirement for interoperation of LRP with other systems is the availability of an API for Pharo Smalltalk, as LRP is implemented in Pharo.

We now give a brief overview of the language by constructing a small example which will be useful later in this text: a state machine that represents a bouncing ball. The machine is called bouncingBall, and has two states: rising and falling, one transition between each state, and two events: goingUp and goingDown. The code for this machine is straightforward. A commented example is shown below, and we describe some notable points next.

```
1  (machine bouncingBall                ; Represents a bouncing ball
2    (state falling)                    ; The ball is falling
3    (state rising)                     ; The ball is rising
4    (on goingUp falling->rising)       ; When to transition between
5    (on goingDown rising->falling)     ;    the different states
6    (event goingUp [velocity > 0])     ; What it means to go up
7    (event goingDown [velocity < 0]))  ;    or to go down
8  (spawn bouncingBall falling)         ; Start by falling
```

[5] https://education.lego.com/mindstorms

[6] http://developer.parrot.com/

In this code, lines 6 and 7 define *events*. These serve as triggers for the *transitions*, resp. in lines 4 and 5. When the block of code of the event, e.g. [velocity > 0] returns true, the event is said to *occur* which allows the transition to *trigger*. Such blocks of code in LRP are called *actions*. Actions are allowed in three different places in LRP code, and we will show the other two places in our next example. Actions are written in Smalltalk and are the means for LRP to communicate with the outside world through an API. For this example, we suppose a minimal API that just exposes the vertical velocity of the ball as a global variable velocity. Hence, if velocity > 0, the ball is going up, and if < 0, the ball is going down. Lastly, on line 7 we state that the machine should start in the falling state. This is needed as LRP has no special start or end states.

We now change the bouncingBall machine to add a counter of the number of times the ball has bounced. This counter is kept in a count variable and initialized to 0 by an action. Variables in LRP are untyped, can be global or defined inside a machine, and have lexical scope. To increase the variable on each bounce, we add an on-entry action to the rising state. In LRP, states can define actions that are executed atomically when entering a state (onentry), when leaving a state (onexit), and when being in a state (running). For the latter actions, the interpreter executes them in a loop with a default rate of 10Hz, which is modifiable by the user.

```
1  (var count :=[0])                          ; Define a counter with value 0
2  (machine bounceCounter
3    (state falling)
4    (state rising
5      (onentry [count := count + 1]))  ; Increase the counter on entry
6    (on goingUp falling->rising)
7    (on goingDown rising->falling)
8    (event goingUp [velocity > 0])
9    (event goingDown [velocity < 0]))
10 (spawn bounceCounter falling)
```

Our bouncing ball example, for sake of brevity, does not include the following features of LRP:

Nested Machines A state can include the definition of a machine, instead of a running statement. This nested machine is started by specifying a spawn statement as the body of the onentry, i.e. instead of the action block. Also, exit transitions go from a state of a nested machine to a state of the parent machine, effectively exiting from the nested machine.

Other Transition Types There are three more types of transitions: eps transitions have no event and always trigger. ontime takes a number or variable name as argument, and the transition occurs after this timeout, given in milliseconds. Wildcard transitions (*->) have no source state and instead consider all states of the machine as a possible source.

Part of the power of LRP lies in its ability to interact with a wide variety of APIs, as actions can contain any Smalltalk code. Hence, if the API is available for Pharo Smalltalk, it can be used by LRP. The example API we have used

until now is extremely simple, so as to provide an idea of what is possible, but we now briefly illustrate the use of the ROS API. ROS is fundamentally a publish/subscribe system over topics, so the core of the ROS API for LRP is focused on this: providing a means to subscribe to a topic as well as a means to publish messages. A top-level variable robot holds the connection to the ROS system, and ROS topics are reified as variables on that object. A user interface allows the programmer to specify subscriptions to topics, with a given variable name, and announcing on which topics the program will publish, also giving a variable name. Reading the variable of a subscription obtains the last message published on that topic, and writing the variable of an announcement causes that message to be published. For example, the code below shows a state that makes a robot move forward:

```
1 (state fwd (running [robot moveBase: [:msg | msg linear x: 0.2]]))
```

We assume that in the user interface, the programmer specifies that Twist messages will be sent to the /movebase/command topic when the moveBase variable is written. While in the fwd state, a twist with linear x value of 0.2 will be published at a rate of 10 Hz, making the robot go forward.

Note that due to the live programming nature of LRP, this value can be changed **while the program is running** and this immediately takes effect, i.e. without needing to save the code, build it, and deploy it to the robot. This allows for extremely cheap experimentation of behaviour variants: just change the value and the robot immediately uses the updated speed.

4 Live Tests for Robotics (LT4R)

As mentioned in the introduction, for transferability, physics simulation must be realistic, yet in the current state of the art it is not difficult to find non-ideal behaviour, c.f. the simple bouncing ball. There is therefore a need for testing. In this section we present our solution: Live Tests for Robotics.

4.1 The Design of LT4R

Providing tests for physics engines is not straightforward, as ideal solutions are often not available, and properly comparing real-world behaviour for a large number of situations is non-trivial. Current testing approaches often concentrate on motion paths, but do not help to identify to what degree different aspects of the overall behaviour are correct. To address these issues, LT4R allows the rapid design of minimal unit tests using a state-based model of expected behaviour.

As a first part of LT4R, we propose the use of state machines to encode the world state, i.e. "what is happening now." For example, this was shown in Section 3 where the state of the world is a ball that is either rising or falling. Such an encoding allows to define different aspects of the required behaviour as different state machines: one for each test. Furthermore, it is arguably straightforward to do since there is a natural mapping of the state of the world

to a state in the machine. Last and not least, it also allows tests to abstract over parts of the world that are not relevant for the test, e.g. the bouncing ball may also be moving forward, but this is not encoded in the state machine as it is not relevant for the test. Consequently, the code of the test only concerns itself with the task at hand, making it easier to write and understand.

In LT4R, the physics engines are tested off-line: we use Gazebo non-interactively to run the simulation and produce a log of motion paths of all the simulated objects as a set of *trajectories*. The first advantage of this approach is that these trajectories can then be consumed by several of our tests, only requiring Gazebo to be run once for the execution of all these tests. Second, generation of such logs can be fully automated which then enables all tests to be automated, independent of engine choice and parameters. Note that we use Gazebo in this work because its support for multiple physics engine makes contrasting behaviours of different engines under identical conditions very easy.

The second part of our solution, interactivity, encompasses the idea that using LRP, the state machine is directly and interactively constructed. This makes the development cycle minimal because a change to a machine can be immediately and interactively tried out on the trajectory under test. Moreover, LRP itself can be tailored to the task at hand. Specifically, for LT4R, we tailored the API for the action blocks, which are used to perform measurements on the trajectories, such that their code is clear and concise.

4.2 The LT4R Implementation

Recall that for LRP to interoperate with external systems, an API must be available for that system. In our case, we need to have some API that allows for the state machines to reason over the recorded trajectories of a physics simulation. The LT4R implementation is exactly this, and we discuss it here.

The conceptual model of the API is a replay of the trajectories: Action blocks of LRP only have access to the physical properties of the objects at the current point in time, called a *snapshot*. The running of a test then consists of replaying the trajectory snapshot by snapshot, where at each step in time the state machine can react appropriately to that snapshot.

This conceptual model yields two advantages: First it results in state machines that encode the current state of the simulated world in a state of the machine, which is arguably a natural mapping. The second advantage is that it restricts action blocks in the state machine to only reason about the present. Consequently we avoid the need for using temporary logic expressions and thus arguably have simpler test code.

LT4R exposes the object trajectories as global variables, in effect making the name of the object in Gazebo a global variable in LRP. This global variable has two methods: `pose` and `velocity`, resp. for the object's pose and velocity (in the current snapshot). Each of these is a six-dimensional vector with methods `x`, `y`, `z` and `rx`, `ry`, `rz` that give the scalar values for the linear and angular components, respectively. Thus, for example, to get the linear z velocity of the bouncing ball

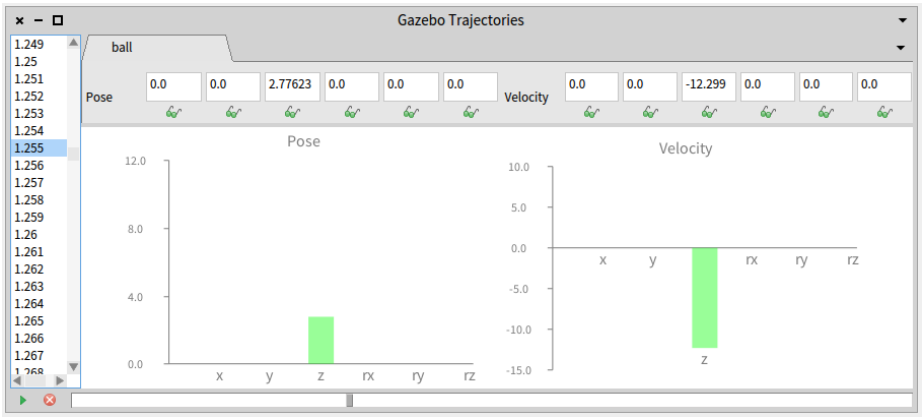


Figure 1: The physics testing API UI showing a snapshot of the bouncing ball trajectory. The left lists all snapshots in order, the right shows pose and velocity graphs for the current snapshot. Each component of these vectors can be plotted over time. At the bottom are controls and a slider for trajectory playback.

example, if the object is called ball in Gazebo the expression in the action block would be: ball velocity z.[7]

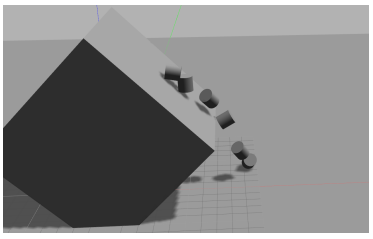
All objects also have the time method, which returns the simulation time of the current snapshot, and there are two global variables: startTime and stopTime that return the time of the first and the last snapshot, respectively.

LT4R also provides a UI for the exploration of object trajectories and control of playback, shown in Figure 1. The current snapshot can then be manually picked from the list and playback controls allow for moving through simulation time. For the selected snapshot, for each object trajectory a tab shows pose and velocity vectors as bar charts, together with the exact values of their scalar components. Each component can be plotted over time for the entire simulation run, examples of which are shown in Figures 2, 3 and 4.

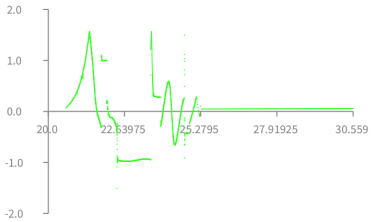
This UI effectively allows for the interactive and live construction of the state machines that encode the test. This development experience is achieved by letting the user interactively experiment with the passing of time, e.g. by using the slider to scrub through all snapshots. He or she then sees the effects on the state machine, and can change this state machine on the fly when needed.

Lastly, LT4R is able to handle intervals on the pose and velocity vectors, since we desire to have tests that are robust to small differences (i.e. 'fuzzy' tests). Tests should reject wildly wrong behaviour but can determine whether the current state is close to an expected value. The comparison interval may differ depending on the test, and therefore should be possible to be specified on a per-case basis. To enable this, both vectors can be turned into intervals in all of their six dimensions (uniformly) by using the '~' operator, e.g. ball pose ~

[7]This is Smalltalk syntax equivalent to ball.velocity.z() in Java.



(a) Stop-motion of the cylinder rotating around the y-axis as it tumbles down a plane.



(b) Plot of the rotational velocity of the cylinder around the y-axis over time.

Figure 2: The tumbling cylinder in Gazebo when using Bullet.

0.5. The center of the interval is the original value and its size is twice the size of the argument to ~. Logic comparisons on the scalars of these vectors will then be comparisons on the interval around these scalars. For example, if the ball is at the origin, the following comparison is true: (ball pose ~ 1) x = -0.75.

5 Writing Unit Tests in LT4R

5.1 Sliding cylinder

We take a break from the bouncing ball example we have been using to show a first simple test: a cylinder that slides down an inclined plane. Again using Gazebo's default parameters, perhaps due to differences in the friction settings or implementation, we found that when using ODE, the cylinder slides down the plane in a straight line, and remains standing on the inclined surface, while using Bullet, the cylinder tips over and quickly starts tumbling end-over-end in the x, y and z axes. This tumbling can be seen in the test data, for example in Figure 2b where the rotational velocity on the y axis is plotted over time.

For the sake of the example, let us suppose that the ODE behaviour is the intended behaviour, and the Bullet behaviour is not. Verifying correctness of behaviour can then be straightforwardly encoded in a test, by asserting that the rotational velocity of the cylinder remains 0. The code is as follows,

```
1  (machine slide
2    (state green)
3    (state red)
4    (on tumbling green -> red)
5    (event tumbling [(unit_cylinder_1 velocity ~ 0.001) ry  != 0 ]))
6  (spawn slide green)
```

By convention, if a test ends in a state named green we consider the test passed, if it ends in any other state, we consider the test failed. If a test reaches a state named red during any moment of its execution it is an immediate fail. The code above thus defines both a red and a green state (in lines 2 and 3) and

declares (in line 4) that when the cylinder tumbles the machine goes to the red state. Line 7 starts the machine in the green state. Hence if at the end of the replay no tumbling occurred, the test passes, and if the cylinder starts tumbling the test immediately fails. Line 6 then declares what it means for the cylinder to tumble: the rotational velocity of y (e.g. as shown in Figure 2b) is non-zero.

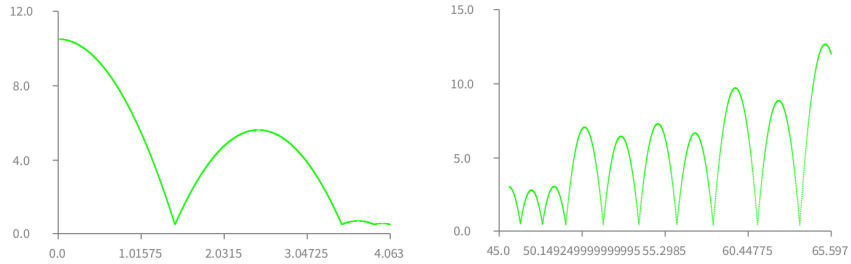
The use of an interval of 0.001 is required here because without it the ODE tests would also fail—the y rotational velocity does not remain exactly at 0. The comparison operator in line 6 filters out noise that we observed while interacting with the data, which makes the test pass. We would stress that this is not a 'fudge factor,' but rather a way to express the desired level of precision.

Note that the state machine for this test effectively abstracts over a large amount of properties that are not relevant to the test, reducing the state of the world to 'acceptable' (green) and 'not acceptable' (red). The resulting test code is concise and very simple to write. It is certainly within the abilities of a simulation designer and could be a test that is written by 'just a user' of a physics engine.

5.2 Bouncing ball

Returning to the bouncing ball example, we now present three different tests that can be made on the same trajectory of a dropped ball. This is to illustrate the different features of our solution as well as to show that there are different kinds of simulation failures that can be identified by different unit tests.

We see that dropping a ball from a fixed height does not always make it bounce correctly, depending on the physics engine used and on its simulation parameters. We therefore implement three successively precise definitions of what is 'correct' as unit tests and we discuss these next: the number of times it bounces; that bounce height should decrease; and that it should decrease in a sensible fashion.



(a) Default ODE parameters.

(b) Solver iterations restricted to 1.

Figure 3: The z position of the ball over time, for two configurations of ODE.

5.2.1 Three Bounces

Using Bullet, dropping the ball does not cause it to bounce, conversely in ODE it bounces three times, as shown in Figure 3a. Given the physical properties of the ball, it should bounce a non-zero number of times. Also, under the same starting conditions this number should always be the same. Let us therefore encode this in a test, supposing that the ball should bounce exactly three times, as observed in the ODE simulation.

```
1  (var count :=[0])
2  (machine bounceCounter
3    (state falling)
4    (state rising (onentry [count := count + 1]))
5    (on goingUp falling->rising)
6    (on goingDown rising->falling)
7    (event goingUp [ball velocity z  > 0])
8    (event goingDown [ball velocity z < 0])
9    (event endWell [ball time = stopTime and: [count = 3]])
10   (state green)
11   (on endWell falling -> green)
12   (on endWell rising -> green))
13 (spawn bounceCounter falling)
```

This test is a straightforward extension of the code we have seen in Section 3, with the code in the events on lines 7 and 8 updated to use the API we defined in Section 4.2. The extensions consist of the definition of a correct ending of the test in lines 9 through 12. Line 9 defines the event that causes the machine to go to green (line 10), which can happen irrespective of whether the ball is falling (line 11) or rising (line 12). The code of the event checks if the time of this snapshot is equal than the last time recorded (stopTime). If this is the case, we are at the last snapshot of the simulation and should therefore decide if the number of bounces is correct. Hence, if the test went well, the machine will end in the green state. Recall that if the machine ends in any other state than green, the test is still considered to fail.

5.2.2 Decreasing Bounce Height

To check whether each bounce results in a successively lower bounce, we invert the problem and fail (enter the red state) if a higher bounce was found as compared to the previous top height.

Here we make use of a nested state machine, in which an inner bouncer machine runs while the system is bouncing correctly. It switches back and forth between rising and falling states, but exits to an outside red state if it ever bounces higher than the previous step. We will see in the next section how packaging the bouncer machine into a nested machine makes development of a derivative test quick and easy.

```
1  (machine bounceLower
2    (var top := [ball pose z])          ; Memorize the initial top
3    (var higher := [false])             ; A boolean tracks if we
4    (state bouncing                     ;  are higher than previous
5      (machine bouncer                  ; Machine nested in bouncing
```

```
6         (state rising)
7         (state falling
8           (onentry [higher := ball pose z > top. ; Check higher first
9                     top := ball pose z]))        ;   then set top
10        (on goingUp falling->rising)
11        (on goingDown rising->falling)
12        (event goingUp [ball velocity z > 0])
13        (event goingDown [ball velocity z < 0])
14        (event badBounce [higher])               ; If 'higher' then
15        (exit  badBounce falling -> red)         ; leave either state and
16        (exit  badBounce rising -> red))         ; go to red in the parent
17      (onentry (spawn bouncer falling)))
18    (state red)
19    (state green)
20    (on eof bouncing -> green)                   ; If we reach the end,
21    (event eof [ball time >= stopTime]))         ; transition to green
22  (spawn bounceLower bouncing)
```

Since we wish to explicitly signal a green state, which is defined here as never reaching the red state, the parent machine watches the current time, and if it reaches the last time of the simulation, transitions from bouncing to green on the eof event. Note that this is a general technique that can be applied to 'negate' any similar negative results-oriented test.

Again, the code is straightforward and arguably easily understandable. The live nature of LT4R significantly aids in development because it allows to build this test as a variant of the previous test. A loaded trajectory is used as a basis for interactive experimentation and the machine is 'grown' step by step, forcing it in specific states as needed and playing back (parts of) the trajectory. Note that this test can be used for different physics engines, or with different configurations of a single engine. This allows the roboticist to ensure that a change of engine or the tweaking of simulation parameters does not break relevant overall behaviour.

5.2.3 Bounce Height Descends Sensibly

In order to develop a more refined test for bouncing ball behaviour, we defer to an example from the Siconos non-smooth dynamical system simulator[8], developed by the BeBop group at INRIA, since we knew it to contain an accurate simulation of the bouncing ball as compared to the closed-form solution [3]. An image of an example trajectory from the Siconos simulation can be seen in Figure 4.

Here instead of only checking that each successive bounce is lower than the previous, we wish to verify that energy loss is consistent between bounces, and thus the ratio of the top of any bounce to the previous should remain the same within some comparison interval.

We make use of an almost identical bouncer machine as in the previous example, with a modification that the **onentry** action for the falling state becomes:

[8] `http://siconos.gforge.inria.fr`

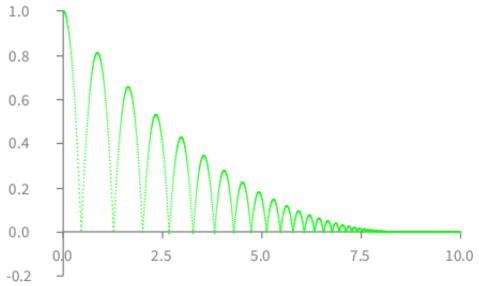


Figure 4: The z position of the ball over time, for simulation in Siconos.

```
24  (onentry [
25    higher := (ball pose ~ 0.02) z > top .
26    correctRatio :=
27      ((ball pose ~ 0.02) z) = (((top - bottom) * ratio) + bottom) .
28    top := ball pose z])
```

Moreover, the definition of badBounce becomes:

```
31    (event badBounce [higher or: [correctRatio not]])
```

To save space we will not reproduce the rest of the bouncer machine here. However, it is clear that all that is needed is a way to determine correctRatio and bottom. For this, we have another nested machine, measureRatio, defined starting on line 7 below, which is entered when the parent machine starts in the measuring state.

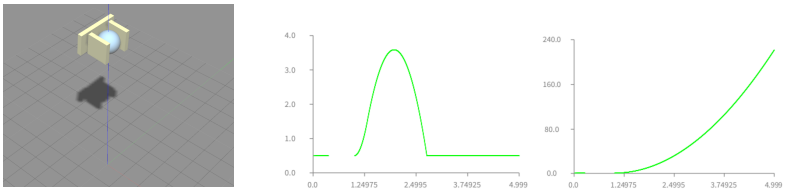
```
1   (machine bounceLowerWithRatioCheck
2     (var top := [ball pose z])          ; Initialize variables assuming
3     (var bottom := [nil])               ;  everything is as expected,
4     (var higher := [false])             ;  but with measured values
5     (var ratio := [nil])                ;  set to 'nil'.
6     (var correctRatio := [true])
7     (state measuring
8       (machine measureRatio             ; Measurements performed on
9         (state falling                  ;  entry to certain states.
10          (onentry [ratio := (ball pose z - bottom) / (top - bottom)])))
11        (state rising
12          (onentry [bottom := ball pose z]))
13        (on goingUp falling->rising)          ; Similar to 'bouncer'
14        (on goingDown rising->falling)
15        (event goingUp [ball velocity z > 0])
16        (event goingDown [ball velocity z < 0])
17        (event measured [ratio isNumber]))    ; Exit when ratio
18        (exit measured falling -> bouncing)   ;  is non-nil
19      (onentry (spawn measureRatio falling)))
20    (state bouncing

...

35    )
36    (state red)                       ; Similar to previous examples
37    (state green)
```

```
38    (on eof bouncing -> green)
39    (on isHigher falling ->red)
40    (event eof [ball time >= stopTime]))
41  (spawn bounceLowerWithRatioCheck measuring)  ; Start in 'measuring'
```

We note that in Figure 4, it can be seen that the ball keeps bouncing to very small values, which eventually cannot be seen. A human cannot tell if it continues to bounce or if it stops. This test uses an interval of 0.02, and therefore bounces are not considered once they get smaller than this interval. This is done explicitly, and considered a feature rather than a bug: a more specific test would need to be defined for examining this small-scale behaviour, which likely changes due to approaching limit effects of the numerical integrator. One possibility could be to specify the comparison interval as a proportion of the ball height, but we leave that as an exercise to the reader. In any case, concentrating on only the large-scale bounces allows to focus the test on behaviour, while ignoring the kind of precision numerical work that is needed to consider boundary effects.

5.3 Unit Testing a Gripper

Throughout this section we have given some minimal physics examples, but to motivate application to robotics, here we give a very short test for a gripper simulation. The simulation, visualized in Figure 5a, is composed of 2 blocks on a horizontal prismatic joint that move with some specified force into a ball weighing 0.1 grams, with contact friction. After 1 second, forces are applied in the z axis to move the mechanism upwards. The expected behaviour is for the ball to move with the gripper and not drop. Trajectories of the ball in the z axis can be found in Figure 5b, for 3 and 4 Newtons of continous force.



(a) A gripper with a ball in Gazebo, moving upwards along the z axis.

(b) Ball trajectory in z for grip force of 3 N (left) and 4 N (right).

Figure 5: Model and recorded trajectories of the gripper simulation.

Depending on the grip force and the upwards velocity, the ball either does not move, moves upward and drops, or continues to move upwards. Some very light-weight tests can be written in LRP to test whether the gripper ever lets the ball drop, but we go as far as to split this into two tests: 1) does the ball move upwards at all; and 2) does the ball continue to move upwards. The first can be expressed very briefly:

```
1  (machine gripLift
2    (state green)                                  ; Success and failure
3    (state red)
4    (state immobile)                               ; Initial state
5    (on goingUp immobile->green)                   ; On move up, success
6    (event goingUp [(ball velocity ~ 0.1) z > 0]); Criteria for rising
7    (event goingDown [(ball velocity ~ 0.1) z < 0]);  and falling
8    (on eof immobile -> red)                       ; If we end without
9    (event eof [ball time >= stopTime]))           ;   moving, fail
10 (spawn gripLift immobile)
```

We start off with an immobile state, where the grippers are approaching the ball. Next, there is a transition to green only if the ball's z velocity goes positive, with an interval of 0.1 to allow any small movements that don't result in a true grip to be formed. If the simulation ends still in the immobile state, we transition to red to declare failure.

For the second test, we have a couple of options. We can check whether the ball passes a certain point, or we can simply check if it is still moving up. For the threshold method, we add,

```
3    (state rising)
4    (on goingUp immobile->rising)          ; Replaces immobile->green
5    (on goingDown rising->red)                    ; Fail if it falls
6    (on passedThreshold rising->green)            ; Success
7    (event passedThreshold [ball pose z > 4])     ; Define threshold
```

The alternative is essentially to check if we are still rising when we get to the end of the file, while the other states transition to failure at the end of the file:

```
3    (on goingDown rising->falling)          ; Replaces rising->red
4    (on eof rising -> green)                ; Success if still rising
5    (on eof immobile -> red)                ; Otherise, fail
6    (on eof falling -> red)
7    (event eof [ball time >= stopTime])
```

It is notable that while the code of these tests is very small, they effectively test real-world behaviour that is relevant to robotics. It is certainly feasible for a simulation user to define these tests. Furthermore, in our experience the live nature of LT4R makes it straightforward to produce these test variants. This is thanks to the support for interactively building and modifying the state machine. For example, to observe the effects of a modification the programmer can place the machine in a specific state and replay parts of the trajectory.

6 Conclusion

Existing tests for physics engines are not sufficient to address the concerns of roboticists: is the simulation that he or she is working on physically faithful to the real world? Would the use of a different engine, or a different parametrization of the same engine yield better results? To answer these questions, unit tests of real world situations that can be performed on different engines are needed. Moreover, such tests should be easy to write, to increase the motiva-

tion of the physics engine developer to write them and ideally to allow roboticists themselves to define tests suited to the scenario being worked on.

In this text we proposed a methodology and tool that addresses these issues, called Live Tests for Robotics (LT4R). LT4R can be used to develop small unit tests that robustly encapsulate desired behaviours without being sensitive to small perturbations. It uses state machines to straightforwardly and succinctly encode the state of the simulated world in test logic. The replaying of recorded motion paths then causes state changes in the machine, leading to eventual success or failure of the test. Moreover, thanks to the use of live programming these machines are developed and manipulated interactively while working with example recordings, which lowers the barrier to test creation even further.

We have shown some minimal examples of simple dynamical system behaviours that could be tested, and also provided tests for a basic gripper simulation. All tests are concise and understandable, and many of these could conceivably have been created by users while simply examining the output of their simulation in Gazebo and LT4R.

7 Acknowledgements

We would like to thank Sebastian Maass for identifying the tumbling/sliding cylinder scenario while testing Gazebo's physics engines at Inria Chile.

References

[1] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM.

[2] Adrian Boeing and Thomas Bräunl. Evaluation of real-time physics simulation systems. In *Conf. Computer graphics and interactive techniques in Australia and Southeast Asia (GRAPHITE)*, pages 281–288. ACM, 2007.

[3] Bernard Brogliato and Vincent Acary. Numerical methods for nonsmooth dynamical systems. *Lecture Notes in Applied and Computational Mechanics*, 35, 2008.

[4] Patricio Castillo-Pizarro, Tomás V Arredondo, and Miguel Torres-Torriti. Introductory survey to open-source mobile robot simulation software. In *Latin American Robotics Symposium and Intelligent Robotic Meeting (LARS)*, pages 150–155. IEEE, 2010.

[5] DARPA. DARPA Robotics Challenge: Virtual robotics challenge rules, March 2013. DISTAR Case 21064. Accessed online 08/06/2016: `http://archive.darpa.mil/roboticschallengetrialsarchive/files/VRC_Rules_Release_2_DISTAR_Case_21064.pdf`.

[6] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *Conf. Robotics and Automation (ICRA)*, pages 4397–4404. IEEE, 2015.

[7] Johan Fabry and Miguel Campusano. Live robot programming. In Ana Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence – IBERAMIA 2014*, number 8864 in LNCS, pages 445–456. Springer-Verlag, 2014.

[8] Open Source Robotics Foundation. Gazebo blog: Gazebo survey results. `http://gazebosim.org/blog/survey_2014`, 2014. Online. Accessed: 27-05-2016.

[9] Sven Gowal, Yizhen Zhang, and Alcherio Martinoli. A realistic simulator for the design and evaluation of intelligent vehicles. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1039–1044. IEEE, 2010.

[10] Christopher Michael Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[11] John M. Hsu and Steven C. Peters. Extending open dynamics engine for the DARPA virtual robotics challenge. In Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 8810 of *Lecture Notes in Computer Science*, pages 37–48. Springer, 2014.

[12] Johannes Hummel, Robin Wolff, Tobias Stein, Andreas Gerndt, and Torsten Kuhlen. An evaluation of open source physics engines for use in virtual reality assembly simulations. In *Advances in visual computing*, pages 346–357. Springer, 2012.

[13] Serena Ivaldi, Jan Peters, Vincent Padois, and Francesco Nori. Tools for simulating humanoid robot dynamics: a survey based on user feedback. In *Conf. Humanoid Robots (Humanoids)*, pages 842–849. IEEE, 2014.

[14] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *Conf. Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[15] Steven Peters and John Hsu. Simple benchmarks for speed and accuracy of rigid body dynamic simulators. `http://www.osrfoundation.org/wordpress2/wp-content/uploads/2015/12/multibody2015_scpeters.pdf`, 2015. Presentation slides. Accessed: 27-05-2016.

[16] Wei Qian, Zeyang Xia, Jing Xiong, Yangzhou Gan, Yangchao Guo, Shaokui Weng, Hao Deng, Ying Hu, and Jianwei Zhang. Manipulation task simulation using ROS and Gazebo. In *Conf. Robotics and Biomimetics (ROBIO)*, pages 2594–2598. IEEE, 2014.

[17] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, number 3.2, page 5, 2009.

[18] R. M. Satava. Accomplishments and challenges of surgical simulation. *Surgical Endoscopy*, 15(3):232–241, 2001.

[19] Steven Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990.

[20] G Tassey. Economic impacts of inadequate infrastructure for software testing, planning report 02-3. Technical report, 2002. Prepared by RTI for the National Institute of Standards and Technology (NIST).