Continuous Integration for Testing Full Robotic Behaviours in a GUI-stripped Simulation

Vladimir Estivill-Castro[1] and René Hexel[1] and Carl Lusty[1]

MiPal, Griffith University, Brisbane, QLD, Australia

Abstract. Today, behaviour models are incorporated to perform a sequence of tasks, involving motion planning, object recognition, localisation, manipulation and collaboration, and potentially even learning. Testing models in isolation is insufficient; entire missions that integrate several models should be thoroughly validated before deployment. This paper provides two new contributions: first, to profit from the development of the simulator as an extension to Model-View-Controller (MVC) where the view can be optionally incorporated. Second, to use the simulator with the stripped-GUI to massively scale up the testing the integration of models of behaviour that fulfil missions performed under the paradigm of continuous integration. We explore the challenging aspects of this testing context and illustrate it with a case study where software models of behaviour are parameterized.

Keywords: Behaviour-Based Software, Model-View Controller, Continuous Integration, Test-Driven Development, Headless Simulation

1 Introduction

Continuous Integration (CI) automates the build and testing of source code every time a developer incorporates changes through a version control system [1]. Testing the sophisticated software models that control the behaviour of robots transcends the validation of each model through unit testing [2]. The interaction of elements composing a sophisticated behaviour is of increasingly crucial importance to the reliability of robotics systems. Consider, for example, one software module implementing tracking behaviour of cameras in a vision system mounted on a robot's head. The development and testing of such a module may be focused on handling different camera resolutions, lighting conditions, and target objects. Naturally, the software development of the head tracker may initially ignore any interactions with the locomotion module of the robot. If the robot is legged, the software control for planning trajectories, controlling all leg articulations, and ensuring manoeuvrability, path and trajectory compliance (position and speed of the body's motion) will be the concern of a second team of developers. However, when the modules are placed together, head motions while tracking objects will, e.g., affect the robot's centre of gravity in ways not necessarily predicted by locomotion control. Testing the interactions of the modules in simulation enables the evaluation of such potential interactions. Such simulations offer many benefits, from accelerated testing times to minimising risks to actual hardware [3, 4].

CI favours the deployment of robotic software components with a higher level of robustness and portability [5]. First, other developers become consumers of

the software module, acting as testers or code inspectors. But also, as mentioned earlier, Robotic Unit Testing [2] in combination with frameworks for CI [5] will perform the unit test on the merged changes that have been recently incorporated into the version control system. The framework highlight (and not include) modules whose new release does not satisfactorily pass its tests. CI also facilitates immediate validation as each committed set of changes triggers a recompilation of modified modules, a recompilation of applications composed of multiple modules (where modules have been altered) and the execution of validation tests.

CI is currently considered best practice as software developers often work in large projects without face-to-face contact with all those involved (e.g., Care-O-bot® expands over 100 developers and more than 50 packages [5].) Moreover, the ubiquity today of agile approaches to software development implies that integrations occur regularly. Often improved functionality is integrated at least on a daily basis. The practice of regular, frequent integration has proven superior to the isolated development and lengthy merge of integration conflicts, usually resulting in hard to fix issues and diverging code branches. In robotics, testing the integrated application on the physical robot is infeasible for each new version of a module. Therefore, using a simulator is much more cost-effective to validate the mission-critical performance of new or updated modules [6].

The options available in modern version control systems (such as `git`) encourage a practice of creating short-lived feature branches. This fast incorporation of features certainly places time limitations on the extent to which testing can be performed on a physical robot and encourages the evaluation of behaviours on simulators. The practice of work isolation to a branch that does not pollute the master, until desired quality criteria are met puts pressure on the completion of tests in time. Even in simulators, robotic missions may be lengthy, especially if the simulator is forced to reflect the reality of the model through a Graphical User Interface (GUI). In the literature of testing robotic missions at the system-level, tests had to be limited to 20s [6]. Some simulators for machine-learning, multi-agents systems and swarm robotic emphasize detached visualizations mainly to attach several views [7]; however, such literature does not address Continuous Integration of Behaviour-based control and the resulting testing benefits.

We propose that simulators should follow the Model-View-Controller software pattern and potentially be stripped of their GUI. Fundamentally, the software of a simulator does not have to operate the virtual model in real-time. Robotic Unit Testing requires that the simulator shall "substitute real sensors and actuators *sufficiently* well" [2]. We argue further that what is needed is that the simulator can maintain the same physical effects, control flow, and event ordering. This way, it should then be possible for a simulator to manage all of the objects in a simulation at much higher speed (than real-time), limited only by processing capabilities. The simulator should consequently scale up tremendously, without altering or biasing the final result. We integrate model-driven development of behaviours because these are developed as executable logic-labelled finite-state machines [29, 31].

2 The MVC Pattern for a Simulation

Model View Controller (MVC) is perhaps the most popular design pattern because of the massive proliferation of devices that enable Graphical User Interfaces. Some scholars [8] consider MVC an architectural-style pattern, and it is complementary to tiered architectures for web-applications [9–11] [1]. The MVC pattern separates the organisation of data and knowledge tasks (the model) from the user interface and presentation (the view). MVC introduces a controller to mediate between user-generated events that request operations on the data, and to update the presentation of model elements for the view to render appropriate visualisations. In this paper, we only require the presence of such mediation and use MVC broadly to include similar architectures and refinements (such as *Model, View, View-Model*, MVVM) that are often distinguished from MVC.

In the design and implementation of simulators, the MVC pattern has been applied, especially for virtual reality (simulators) [3]. In this space, researchers consider USARSim [12], Gazebo [13], ROAMS [14] and Webots [15] the leaders because these simulators feature physics simulations and rigid body dynamics, as well as simulation of various sensors. We also consider V-REP [16] as another simulator in this category. The first challenges faced by the design of these simulators is semantic world modelling and semantic data rendering to represent complex information about the environment and the robots. The application of MVC to the architecture of robotic simulators could be attributed to Tobler [17] for separating the semantics of the simulation from the rendering. The model encapsulates all there is to know and represent about the environment, objects and the robots, while the view is in charge of its visualisation. Thus, the model absorbs the earlier semantic graphs. While some rules of behaviour are placed in the controller. The focus of simulators has followed a path similar to the computer gaming industry, even being nicknamed *eRobotics* [3]. It has enhanced the way simulators have become holistic tools, aiming to assist in several fields including education and research, improving realism of virtual environments and semantic data interpretation. With respect to testing, this effort has resulted in Virtual Testbeds [3]. Such "simulation-in-the-loop" testing [5] is mainly performed through visualisation of robot performance in a mission [18]. Since running missions visually on the simulator takes a long time, testing can easily consume a large amount of time during the development process. This lengthy test potentially leads to several anti-patterns of continuous integration [1], most notably not to *categorise tests* and not to *automate tests*. The robotics community is favouring ROS with Gazebo for development of modules for localisation and navigation under a simulation process that can be directly implemented on the real robot without modifications [2, 5, 19]. On the other hand, Wienke and Wrede argue for the opposite. Rather than more complex integration and testing, they favour performance testing on a per-component basis, because mission testing in robotics is "much harder to set up and maintain due to the complex interactions of robots with the real world and the non-standard interfaces" [20].

[1] For example, *Rails* codifies SaaS application structure as MVC.

3 Continuous Integration

Software quality can be more effectively achieved with testing and validation from the early stages by applying continuous integration. One fundamental claim of MVC is that, when the pattern is used correctly, model classes are reusable without modification. The second fundamental claim is that MVC can also make the view reusable without modification. These two claims strongly suggest that there is a significant decoupling between the model and the view.

But such decoupling seems uncommon among robotics simulators. Gazebo is part of the *Player* project that claims to be "*probably the most widely used robot control interface in the world.*"[2] Although Gazebo decoupling into a client-server architecture enables high-performance physics engines like *Open Dynamics Engine (ODE)*, *Bullet*, *Simbody*, and the *Dynamic Animation and Robotics Toolkit (DART)*, its tight integration makes it difficult to facilitate a GUI-stripping option. A review of web posts on the `-g` option (or `roslaunch gui:=false`) when using ROS, shows that this is far from being a mature option incorporated in the fundamental design. Similarly, www.forum.coppeliarobotics.com (V-REP's forum) has a discussion thread regarding the package's `-h` option for simulations in *headless* mode (without a GUI). There is an admission by the developers that the `-h` option "*is not a true headless version of V-REP*"[3]. The discussion thread documents that the attempts to compile V-REP from source with the `makefile_noGui_noGl` option, are also ineffective, despite several version upgrades. This again illustrates that model/view separation and architectural style were not initially in the core design plans of V-REP.

The documentation of *MORSE* discusses a feature to operate headless. The documentation concurs with our argument that such capability enables the execution of the simulator on powerful servers, and integrating the simulator in a Continuous Integration pipeline, which could run automated testing. However, this option for MORSE has only being tested on Linux and still requires compilation against OpenGL. The headless execution is supported by a trick when displaying *MORSE*'s main window. The GUI is not prevented from appearing, but the 3D application elements are rendered in memory instead of the screen by using `Xvfb`[4]. This trick, once again, illustrates the tight coupling that exists between the view and other components of this simulator.

In his analysis of the preconditions necessary to migrate the evaluation and testing of Cyber-Physical Systems to cloud environments, Berger [21] makes the point that "the term 'simulation' is often used interchangeably with 'visualisation' while having a 3D environment in mind". He emphasises that headless simulation environments must be able to execute in an unattached manner and that simulation time be decoupled from real-time.

Continuous integration and testing for the delivery of reliable software that incorporates simulations has been around for quite some time [22]; however, in this paper we emphasise that the capability of simulators to toggle between GUI

[2] playerstage.sourceforge.net
[3] Posting by `coppella`, site administrator, on 25th Match 2017.
[4] www.openrobots.org/morse/doc/latest/headless.html.

and headless mode (because of a strict separation) truly potentiates the software development environment. Therefore, we have developed a simulator using `Swift` and `GTK` that is multi-platform (runs on macOS and Linux). From its conception and design, we incorporate the MVC architectural paradigm. Therefore, the capability to execute simulations stripped from a GUI, what the community has recently named headless simulations, are genuinely possible.

Headless simulations are particularly relevant for continuous integration; however, to the best of our knowledge only one instance [23] has achieved the type of software quality management process proposed here. This earlier deployment infrastructure [23] uses a headless simulator sporadically as part of some unit test. The focus has been the launching of complete compilation in isolated `Docker` images and sharing build load. Robotics Unit Testing [2] has mainly used [2, 5, 6] `Jenkins` for CI although other alternatives (such as `GitLab CI`, `Buildbot`, `Drone`, and `Concourse`) are now available in the market. For our implementation, we have chosen `Jenkins` because several reviews consider it one of the best tools. It is regarded a both powerful and flexible. The usual criticism is that `Jenkins` has a steeper learning curve, but feature extensions through plugins are believed to compensate for this.

4 Testing Behaviours

Logic-labelled finite state machines (LLFSMs) enable compositions beyond reactive behaviours. They can naturally incorporate deliberative components, because transitions labelled by Boolean expressions can represent a query to a task planning component [24], or a reasoning agent [25]. Recall that the origins of behaviour-based control derive from the composition of timed finite-state machines under the subsumption architecture in the development of the seminal Toto [26]. The continuous integration and testing proposed here evaluate behaviours as modules for control in accordance with Behaviour-based control [27]: behaviours are more complex than actions. Therefore, we consider *behaviours* those modules that constitute time-extended processes achieving or maintaining a particular goal. We progress further from the skill (or tasks) testing proposed earlier in Robotic Unit Testing [2]. Our approach here covers both: what has been described as Component Integration Testing and System-level Testing [6].

Models of behaviour consisting of the composition of some form of state-charts are commonly used in the software engineering community as the best representation from which to automatically generate test-cases. Such representation is combined with graph-coverage criteria. The most popular of such criteria is the Edge Coverage criteria [28]. We found only one earlier proposal for the automatic generation of tests in robotic behaviour [6]. Such approach is based on Communicating Extended Finite-State Machines and thus on the graphical representation of behaviours and graph-coverage criteria [6]. The Edge Coverage Criteria generates a test suite so that each transition of each state-charts is executed in a test. Testing is never a proof of correctness and even the Edge Coverage Criteria is insufficient to comprehensively test behaviour. Our approach is to extend such coverage criteria to testing of the goal that each behaviour is to

achieve. The only approach we have found in this direction is the reported evaluation of the `cob_navigation` package [5]. We also focus on coverage of all the combinations of parameter conditions that define valid inputs for a behaviour (and even some invalid inputs) implementing a *parameter provider* [20].

We now use an example of a complex behaviour that is composed of lower-level, internal behaviours. We chose an example from the RoboCup Standard Platform League (spl.robocup.org). Our entire soccer player behaviour follows a top-down design. The behaviour must, at the top level, maintain a few states, named Initial, Set, Ready, Play, and Penalised. Suffice it to say that the top behaviour; therefore, implements these states as sub-LLFSMs with corresponding transitions reacting to the stimuli (e.g., UDP messages, a whistle, or even buttons on the robot being pushed). We focus on the state of Ready, where a robot must reach a legal position (usually its own half of the field) before game resumes. We present the validation of this complex behaviour (in our simulator with the option that displays the GUI with a video: youtu.be/6bzyf5fhTAQ). Thus, the state of Ready is again broken down into sub-behaviours, namely, to find a landmark (a goal), and identifying whether that landmark is in the opponent's or the player's half. Finding a goal (if not visible) corresponds to scanning using the head, and if that is not enough, to spin the whole robot around a bit (on the spot). However, if the goal is visible, we need two sub-behaviours, one to track the visible object with the head, and one to align the body to the object.

The behaviours themselves are sophisticated LLFSMs, capable of handling parameters and recursion [29]. Thus, the behaviour that tracks a landmark, such as a goal-post, is capable of also tracking and following the ball (option triggered by a parameter). Other parameters regulate which post to follow. Typically, the landmark to follow is set according to the player number, so the behaviour brings a player back to the correct position within the team of robots.

We structure continuous integration tests by functional decomposition [30]. The elementary behaviours (LLFSMs) can be validated on their own: our elementary `SMStopWalking` behaviour stops the walk of the legged robot (including disengaging from the DCM cycle) and sets it in a posture from which it can kick or perform some other recorded motion. All test of behaviours under continuous integration requires a script with a corresponding setup section and a tear-down section. Behaviour and mission tests are not validation tests of the simulator itself. We emphasise here the distinction with the test-driven development (TDD) approach we have taken to the development of our simulator. Figure 1 shows an `XCTest` for unit testing under `Xcode` of the simulator. This test checks that a spin command (of 10 degrees per second) for the robot does cause the robot to change its orientation π radians after 18s of simulation time. This example illustrates that *actions* can be tested on the simulator and that these are properties of correct behaviour. That is, actions can be validated with simpler unit testing frameworks. But the testing of the LLFSM `SMStopWalking`, is conceptually and practically more sophisticated. All tests ob behaviours also require the infrastructure of continuous integration (`Jenkins`) and version control (`git`).

Bottom-up Testing of Behaviours The function composition of a behaviour can be the result of a bottom-up approach, composing basic behaviours into more elaborate ones. But it can also be the result of a top-down construction,

```
func testSpinWalkCommand (){
    let wb = Whiteboard() // post to the whiteboard a spin command
    var naoWalkCommand = wb_nao_walk_command()
    naoWalkCommand.turn = 10  // 10 degrees per second
    naoWalkCommand.walkEngineOn = true
    naoWalkCommand.left = 0; naoWalkCommand.forward = 0
    wb.post(naoWalkCommand,kNaoWalkCommand_v)
    // test parameters
    let seconds = 18
    let test_rate = 30
    // get a controller
    let theController = Controller(the_rate: test_rate, init_robot_positon: false)
    // place the robot in the middle of the field
    let px = theController.viewModel.model.soccerArea.width/2
    let py = theController.viewModel.model.soccerArea.height/2
    let areaPosition = Coordinate2D(x: px, y: py)
    theController.viewModel.model.robot.position = areaPosition
    theController.viewModel.model.robot.orientation = 0
    let ticks = seconds * test_rate //run as many simulator ticks
    for _ in 1...ticks { _ = theController.updateWhiteboard()}
    XCTAssertEqual(theController.viewModel.model.robot.orientation, π, accuracy: 0.0000001)
}
```

Fig. 1: Unit testing under `Xcode` with `XCTest` for TDD of the simulator.

where an elaborate behaviour is structured in a divide-and-conquer approach. For the testing of behaviours, we prioritise them in bottom-up order; that is we test sub-behaviours because if any of them are faulty, the integrating behaviour is probably also faulty.

Therefore, we organise our tests by developing the `Jenkins` scripts that validate behaviours that do not depend on other behaviours. Figure 2 shows the functional decomposition of the behaviours that constitute the soccer player (which sits among several applications installed in our Nao robot). Thus, the automatic generation of behaviours is guided by the hierarchical structure of the functional decomposition. The leaves of this hierarchy are tested independently of other behaviours, but may require additional modules (object recognition in images, Kalman filtering). We already illustrated one such leaf, the `StopWalking` behaviour. But the testing also integrates all behaviours in each subtree. Figure 2 shows a frame when testing the behaviour that focuses on a designated goal (and can also, determined by a parameter, walk to it until a certain distance). So this behaviour requires four test types: (1) finding the opponent's goal by spinning alone, (2) finding the opponent's goal and approaching it, (3) finding our goal by spinning alone, (4) finding our goal and approaching it. Naturally, these tests should be repeated from different initial conditions defined by the posture (position and orientation) of the robot performing the behaviour. Therefore, we define the coverage of the test as all possible combinations of the parameters of the behaviour. In the example above, type of goal and type of focus results in 4 possible types of inputs to the behaviour. As mentioned earlier, we automate the generation of parameters in discrete ranges (and testing all combinations of "blocks" has been noted to generate many test-cases [6]). We also aim to characterise the types of environment settings. But this is significantly more difficult to

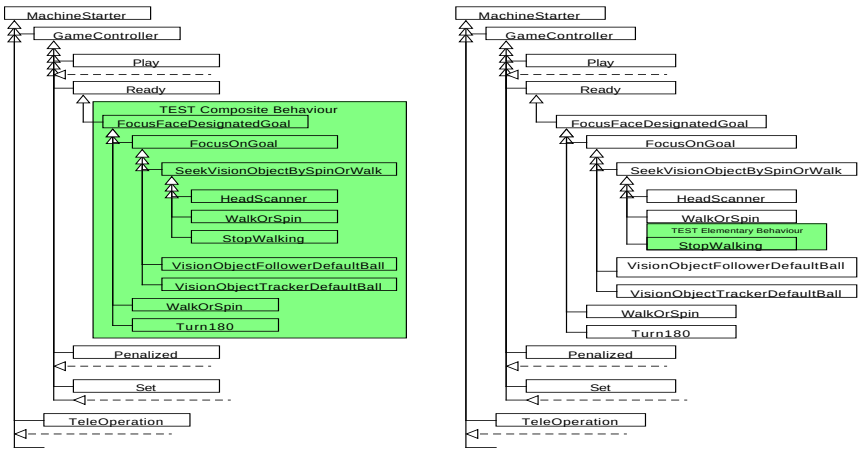

Fig. 2: The hierarchy by which a behaviour is composed of sub-behaviours.

characterise and automate without providing the detailed semantics of the behaviour's mission. Thus, at the moment, we configure the different environments is a manual process (similarly as the manual coding reported earlier [6]).

Lessons learnt The implementation of our proposal revealed that significant gains are obtained with complex behaviours; although initially, the process of developing tests seems laborious for just validating a simple behaviour. E.g., the scripts that perform the test can be subject to significant factorisation. The setup and tear-down phases of each test have many elements in common across tests enabling a generic setup script and a generic tear down script.

Another aspect related to this development is that the test scripts (normally developed in a UNIX-`shell`) are subject to testing and validation. Therefore, we recommend that such scripts also offer the option to switch between headless or with GUI for the simulator. The option to execute the script with GUI enables visual validation of the test script before submitting it and setting it up in the Continuous Integration engine. The same simulation should produce the same result although it will typically execute differently in headless mode than with a GUI. The operating system hosting the simulator (in our case MacOS) is multitasking the simulator with many other processes. For testing and analysing component resource utilisation (such as CPU usage under varying load), "a dedicated host free of other tasks to avoid resource sharing issues" is recommended [20] to ensure fidelity of the measurements. We argued in the introduction that headless mode accelerates the testing overall, and therefore, the difference in execution between headless mode and GUI-mode is to be expected. This difference also puts the behaviours to the test. The behaviours are running in a different process (with `clfsm` [31] this is typically a single thread), and modules such as camera image processing, Kalman filtering or localisation particle filters are their own processes. Therefore, the Continuous Integration ex-

ecutions test variable concurrency conditions that result from the multi-tasking environment that executes all processes (on the robot or on the CI server).

Table 1: The behaviours coded by executable models (LLFSMs) and incorporated into composed behaviours as per Figure 2.

Behaviour	Description
`MachineStarter`	Select an application using the robot's buttons.
`TeleOperation`	Application for remote control of the robot.
`GameController`	Application for RoboCup: respond to SPL league controller and implement referee button interface.
`Play`	Actively play: chase ball, kick, maintain formation.
`Ready`	Place in formation and legal position to re-start the game.
`Set`	Await re-start the game trough wizzle focusing on ball.
`Penalized`	Await signals to resume game and re-start localization.
`FocusFace DesignatedGoal`	Find a goal, decide if its opponents, turn until find our goal, walk to our field tracking our goal, mindful of your position, once in position, face opponents.
`FocusOnGoal`	Find any goal as a landmark, and line head and body directly at it.
`Turn180`	Spin on the spot using odometry.
`SeekVisionObject bySpinorWalk`	Find a landmark (a goal or the ball as per parameter), and do so by only spinning on the spot or optionally by exploratory walk.
`HeadScanner`	Rotate neck left to right and up and down expanding the field of vision of the camera.
`WalkOrSpin`	Perform a spin on the spot (parameter indicates clockwise vs counter-clockwise) or additionally perform exploratory walk, do not control head.
`StopWalking`	Stop the walk of the legged robot, disengage from the DCM cycle and set it in a posture from which it can kick or perform some other recorded motion.
`VisionObjectTracker DefaultBall`	Apply a *Proportionate* control as a feedback loop on top of a Kalman filter to track an object with the robot's' head.
`VisionObjectFollower DefaultBall`	Apply a feedback loop control to walk the robot and minimize the distance to the object as well as align the body to the object.

The setup phase involves the following, generic steps, even if the platform used is ROS and a simulator like `Gazebo`. However, as mentioned earlier, we found this platform extremely fragile in that the launch infrastructure repeatedly fails.
1.- Verify all environment variables are set.
2.- Confirm all behaviours are compiled and reside in corresponding launching directory.
3.- Terminate all other executing instances of modules that are reachable via the middleware and could interfere (image-processing, network messages, etc)
4.- Re-start the middleware
5.- Re-start modules required for the test.
6.- Use an interface to the headless simulator to set up the scene for the test.
We test each behaviour covering all combinations of types of the parameter to the behaviour (for numerical values we discretise [5] on ranges. We ensure that a positive value, a negative value and zero are validated even if a negative value is an incorrect parameter for some cases; such as a distance, but it is usually fine for an orientation or bearing).

The issue of setting up the scene demands an interface to a headless simulator. A very important aspect of a headless simulator is that it must offer a mechanism

[5] The only other work on test-generation [6] refers to this as "blocking" or "dividing into blocks" a range of values (for instance, for a robot's position, each of its x, y coordinates is "*blocked*" into intervals of 0.5m).

to operate in headless mode the model and the widgets that its GUI would offer. Simulators provide interactive options to place the robot or other objects in particular postures and operate on sensors (for example, push a button on the robot) or enable/disable other modules (change resolution on camera, provide image-processing filters). Such headless interface access to the simulator's model is the source of significant complexity if unavailable [2] and even if available, middleware timing issues can complicate unit testing [20]. Therefore, in our design for its headless operation, we provide two alternatives. We offer command-line options that can set the scene and the characteristics of the robot(s) and its environment. But we also provide access through our efficient shared-memory middleware communication framework [32]. For those familiar with ROS and `Gazebo`, our facilities are similar to the capacity to use `rostopics` from setup or tear-down scripts to configure a scene for `Gazebo`.

These facilities are also essential to validate the behaviour. The middleware channels enable to retrieve from the headless simulator the final posture of a robot or objects, elapsed mission time, and other criteria to evaluate the success of the behaviour.

The simulator itself as well as other processes that execute concurrently have options to be launched with different levels of verbosity that can be logged. In our setting, we have configured our execution of an LLFSM arrangement with `clfsm` [31] to record the trace of execution. We also can log the operator of our Kalman filter as well as the traffic on our middleware infrastructure (`gusimplewhiteboard` [32]). This fulfils three crucial requirements. First, it is possible after a test-failure to play out, trace and analyse the behaviour and the environment evaluation to understand what lead to the failure. Second, it is possible to carry out "*models at run time*" in the sense expressed by Barbier [33]. Since LLFSMs are executable models of behaviour, we can visualise, and follow close their execution even if the simulator is executing headless. Third, the logs can be inspected to proactively be used for an analysis that determines the failure or success of the mission (from whether the robot is stuck, or whether there are no more actions because the mission is completed).

We also consider important to mention that we can simultaneously launch monitoring behaviours [34] and use behaviours that validate other behaviours [35].

5 Conclusions

Here we have demonstrated that a strict separation between the model and the view (following the MVC paradigm) can be extended towards an abstract view that allows a simulation to run with and without a graphical user interface. Importantly, we enable the evaluation of complex behaviour faster than real-time, enabling us to scale up the testing of sophisticated robotic teams and systems. Using our case study of a team of robotic soccer players, we have demonstrated that this can be automated using a continuous integration infrastructure. Thus, validation of the behaviour of autonomous robots and other, complex distributed real-time systems become part of a systematic, integrated software development process, greatly enhancing system quality and productivity.

References

1. P. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley 2007.
2. A. Bihlmaier and H. Wörn, "Robot unit testing," *Simulation, Modeling, and Programming for Autonomous Robots*, Springer 2014, 255–266.
3. N. Hempe, R. Waspe, and J. Rossmann, "Combining complex simulations with realistic virtual testing environments – the eRobotics-approach for semantics-based multi-domain VR simulation systems," *Simulation, Modeling, and Programming for Autonomous Robots*, Springer 2014, 110–121.
4. M. R. Zofka, F. Kuhnt, R. Kohlhaas, and J. M. Zöllner, "Simulation framework for the development of autonomous small scale vehicles," *IEEE Int. Conf. Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2016, 318–324.
5. F. Weisshardt, J. Kett, d. Freitas Oliveira T. A., A. Bubeck, and A. Verl, "Enhancing software portability with a testing and evaluation platform," in *ISR/Robotik; 41st Int. Symp. on Robotics*, 2014, 1–6.
6. M. Abdelgawad, S. McLeod, A. Andrews, and J. Xiao, "Model-based testing of real-time adaptive motion planning (RAMP)," *5th IEEE Int. Conf. on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR*. 2016, 162–169.
7. S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Catalin-Balan, "MASON: A multiagent simulation environment," *Simulation*, v. 81, n. 7, 517–527, 2005.
8. T. Mikkonen, R. Pitkänen, and M. Pussinen, "On the role of architectural style in model driven development," *Software Architecture*, Berlin: Springer 2004, 74–87.
9. H. Prajapati and V. Dabhi, "High quality web-application development on java EE platform." IEEE Comp. Soc., 04 2009, 1664 – 1669.
10. P. L. Thung, C. J. Ng, S. J. Thung, and S. Sulaiman, "Improving a web application using design patterns: A case study," *Int. Symp. IT*, v. 1, 2010, 1–6.
11. P. Worrall and T. Chaussalet, "Development of a web-based system using the model view controller paradigm to facilitate regional long-term care planning," *2011 24th Int. Symp. on Computer-Based Medical Systems (CBMS)*, 2011, 1–7.
12. S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "USARSim: a robot simulator for research and education," *IEEE Int. Conf. on Robotics and Automation*, 2007, 1400–1405.
13. N. P. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator." *IROS Int. Conf. Intelligent Robots and Systems*, Sendai, 2004, 2149–2154.
14. A. Jain, J. Guineau, C. Lim, W. Lincoln, M. Pomerantz, G. Sohl, and R. Steele, "ROAMS: Planetary surface rover simulation environment," *Int. Symp. Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, 2003 1923.
15. O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, v. 1, n. 1, 39–42, 2004.
16. E. Rohmer, el al. "V-REP: a versatile and scalable robot simulation framework," *IROS Int. Conf. on Intelligent Robots and Systems*, 2013, 1321–1326.
17. R. F. Tobler, "Separating semantics from rendering: a scene graph based architecture for graphics applications," *The Visual Computer*, v. 27, n. 6, 687–695, 2011.
18. P. Jayasekara, G. Ishigami, and T. Kubota, "Testing and validation of autonomous navigation for a planetary exploration rover using open source simulation tools," *11th Intl. Symp. on Artificial Intelligence, Robotics and Automation in Space*, 2012.
19. K. Takaya, T. Asai, V. Kroumov, and F. Smarandache, "Simulation environment for mobile robots testing using ROS and Gazebo," *20th Int. Conf. on System Theory, Control and Computing (ICSTCC)*. IEEE, 2016, 96–101.

20. J. Wienke and S. Wrede, "Continuous regression testing for component resource utilization," *5. IEEE Int. Conf. Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR*, 2016, 273–280.
21. C. Berger, "Cloud-based testing for context-aware cyber-physical systems," in *Software Testing in the Cloud: Perspectives on an Emerging Discipline*, IGI Global, 2012, 68–95.
22. M. A. Salichs, E. A. Puente, L. Moreno, and J. R. Pimentel, "A software development environment for autonomous mobile robots," *Recent Trends in Mobile Robots*. World Scientific 1994, 211–253.
23. C. Berger, "An open continuous deployment infrastructure for a self-driving vehicle ecosystem," *Open Source Systems: Integrating Communities*, Springer 2016, 177–183.
24. V. Estivill-Castro and J. Ferrer-Mesters, "Path-finding in dynamic environments with PDDL-planners," *16th Int. Conf. Advanced Robotics (ICAR)*, Montevideo, 2013, 1–7.
25. V. Estivill-Castro, R. Hexel, and A. Ramírez Regalado, "Architecture for logic programing with arrangements of finite-state machines," *First Workshop on Declarative Cyber-Physical Systems (DCPS) at Cyber-Physical Systems* IEEE, 2016, 1–8.
26. M. Mataric, "Integration of representation into goal-driven behavior-based robots," *Robotics and Automation, IEEE Transactions on*, v. 8, no. 3, 304 –312, jun 1992.
27. R. C. Arkin, *Behavior-Based Robotics*. Cambridge: MIT Press, 1998.
28. A. C. R. da Silva, A. R. ana Paiva and da Silva V. E. R., "Towards a test specification language for information systems: Focus on data entity and state machine tests," *6th Int. Conf. Model-Driven Engineering and Software Development*, Portugal: SCITEPRESS 2018, 213–224.
29. V. Estivill-Castro and R. Hexel, "Verifiable parameterised behaviour models for robotic and embedded systems," *6th Int. Conf. on Model-Driven Engineering and Software Development*, Portugal: SCITEPRESS 2018, 364–371.
30. K. K. Aggarwal and Y. Simgh, *Software Engineering* New Age, 2008.
31. V. Estivill-Castro and R. Hexel, "Arrangements of finite-state machines semantics, simulation, and model checking," *Int. Conf. on Model-Driven Engineering and Software Development MODELSWARD*, Barcelona: SCITEPRESS 2013, 182–189.
32. V. Estivill-Castro, R. Hexel, and C. Lusty, "High performance relaying of C++11 objects across processes and logic-labeled finite-state machines," *4th Simulation, Modeling, and Programming for Autonomous Robots Int. Conf., SIMPAR* , LNCS 8810. Bergamo: Springer, 2014, 182–194.
33. F. Barbier, "Supporting the UML state machine diagrams at runtime," in *Model Driven Architecture – Foundations and Applications*, LNCS 5095. Berlin: Springer 2008, 338–348.
34. V. Estivill-Castro and R. Hexel, "Run-time verification of regularly expressed behavioral properties in robotic systems with logic-labeled finite state machines," *5th IEEE Int. Conf. Simulation, Modeling, and Programming for Autonomous Robots SIMPAR*, 281-288 2016.
35. V. Estivill-Castro, R. Hexel, and J. Stover, "Modeling, validation, and continuous integration of software behaviours for embedded systems," in *9th IEEE European Modelling Symp.*, Madrid, 2015, 89–95.