CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework

Eric Rohmer, Surya P. N. Singh and Marc Freese[1]

***Abstract*— From exploring planets to cleaning homes, the reach and versatility of robotics is vast. The integration of actuation, sensing and control makes robotics systems powerful, but complicates their simulation. This paper introduces a versatile, scalable, yet powerful general-purpose robot simulation framework called V-REP.**

The paper discusses the utility of a portable and flexible simulation framework that allows for direct incorporation of various control techniques. This renders simulations and simulation models more accessible to a general-public, by reducing the simulation model deployment complexity. It also increases productivity by offering built-in and ready-to-use functionalities, as well as a multitude of programming approaches.

This allows for a multitude of applications including rapid algorithm development, system verification, rapid prototyping, and deployment for cases such as safety/remote monitoring, training and education, hardware control, and factory automation simulation.

I. Introduction

The exponential increase in processing power of computers (not to mention 3D graphics hardware) along with the plethora of open software and hardware standards has drastically changed the landscape in the field of (3D) robotics simulation. Not only has this enabled more complexity on the desktop, but conversely it has provided the ability to run simulations (in real-time) with hardware in-the-loop, or to have mobile/embedded systems controlled from a simulation framework.

While it is possible to assemble a simulator from the various kinematics, physics and graphics libraries, the architecture and control methodology are crucial to determining how these elements interact and thus the overall performance and accuracy of the system. A robust systems approach advocates for a versatile, scalable and fine-grained simulation strategy.

Practically, a general-purpose robot simulator has to provide multitude tools and functionalities simultaneously, while abstracting the underlying robotic systems and their complexity since system specificities cannot be foreseen. Additionally, one wants a flexible controller approach that can be portable and easily coded (and maintained), generalizable to various models, and scalable (i.e. simulation entities should handle multiple models, controllers or any other functionality).

[1] Corresponding author: marc@coppeliarobotics.com
Eric Rohmer is a lecturer at the State University of Campinas, Brasil
Surya Singh is a lecturer at the University of Queensland, Australia
Marc Freese is CEO of Coppelia Robotics, Switzerland.

There are currently several robot simulation platforms available, for instance Open HRP [1], Gazebo [2] or Webots [3]. While some offer competing functionality, many fail in offering a large and complementary palette of programming techniques, and their simulation models and controllers are only partially portable: models, controllers and other functionality are clearly distinct, and thus need separate handling. For example, controller recompilation on a different hardware or platform is often necessary, or the simulation model and controller need to be carefully matched since they represent at least two distinct files, and when scaling is supported, it is done via relatively obscure hard-wired mechanisms.

The Virtual Robot Experimentation Platform [4] (main user interface shown in Fig. 1) – or simply V-REP simulator – is the result of an effort trying to council all requirements into a versatile and scalable simulation framework. Next to offering the traditional approaches also found in other simulators, V-REP adds several additional approaches. Section II of this paper describes V-REP's control architecture, in which the various possible controller types are explained, in particular embedded scripts. That is, they can be an integral part of a simulation model, thus extremely portable and scalable. Section III discusses the overall offered simulation functionality, and its integration into simulation models, also for the sake of portability. Finally, section IV examines three practical V-REP simulation models and their implementation, as an illustration of this paper's content.

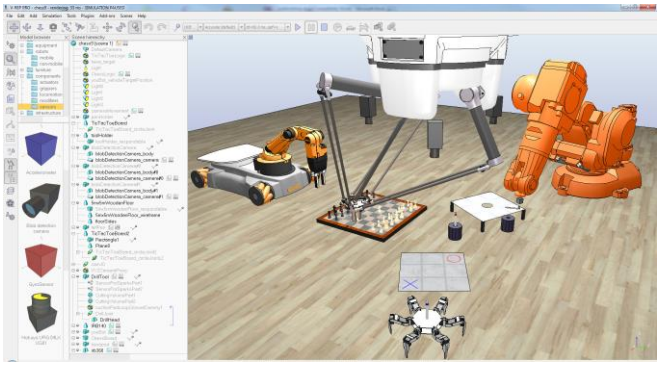

Figure 1. An example V-REP simulation scene showing the diversity of robot types that may be simulated simultaneously

II. Simulation Controllers

If one wants to build complex simulation scenarios, then there is almost no escape from a distributed control framework. It simplifies the task by partitioning control entities, it speeds-up simulation by distributing the CPU load over several cores or several machines, and it allows a

simulation model be controlled by native code execution. There are however simulation requirements that should not be forgotten in pursuit of that goal.

One of the most important and often neglected aspect is the flexibility, portability and scalability of the simulation model: how easy is it to adjust its control code(s)? How many files have to be distributed in order to run the same simulation model on another machine? Will it require recompilation on other platforms? How many versions of the same controller are in circulation? Can various versions operate side-by-side? Can a simulation model easily be instantiated several times, without losing functionality?

Other simulation control requirements are linked to the simulation loop. Some elements, especially the low-level controls such as real-time motion level controllers, require synchronization with the simulation loop. (i.e. executed at the same moment at each simulation pass).

The importance of providing synchronous/asynchronous, external/embedded, native/non-native distributed control techniques in robotics simulations is discussed hereafter.

A. Overview of Common Techniques

The execution of the control code of a simulation or a simulation model is handled using the following three techniques:

- **The control code is executed on another machine**. This could represent a distinct machine or a robot, connected to the simulator machine via a specific network (e.g. socket, serial port, etc.). The main advantage of this approach is the originality of the controller (the control code can be native and running on the original hardware). Another advantage is the reduced computing load on the simulation machine. On the other hand, this approach imposes serious limitations in regards to synchronization with the simulation loop, and the communication delay/lag dictated by the network.

- **The control code is executed on the same machine, but in another process (or another thread) than the simulation loop**. Here also, we can benefit from a reduced, or rather balanced load on the CPU cores, but this comes accompanied with a lack of synchronization with the simulation loop. And most of the time, it comes in pair with a communication lag or thread switching delay (many resources require locking before access, or some algorithms aren't reentrant). This control technique is often implemented via external executables or plug-ins loaded by the simulator.

- **The control code is executed on the same machine and in the same thread as the simulation loop**. The main advantage of this approach is the inherent synchronization with the simulation loop, and the absence of any execution, communication or thread switching lag or delay. This however is only made possible with an increased load on the simulation loop CPU core. This control technique is often implemented via plug-ins loaded by the simulator.

The most common implementations of the above techniques (i.e., using external executables or plug-ins) have as a direct consequence poor portability and poor scaling of simulation models: indeed, since the control code is not attached to its respective simulation model, it will have to be distributed/compiled/installed separately. This increases compatibility problems across platforms, as well as conflict/dependency issues with other libraries. Flexibility is also reduced, since one would have to recompile and reload an executable/plug-in for each small code modification. Model duplication, as in a multi-robot simulation scenario, will have to be supported via hard-wired mechanisms that launch new control instances for each simulation model instance.

B. V-REP Implementation

V-REP allows the user to choose among various programming techniques simultaneously (Table 1) and even symbiotically (Fig. 2):

- **Embedded scripts**. This represents the most powerful and distinctive feature of V-REP. The main simulation loop is a simple Lua [5] script (called "*main script*"), part of a given simulation scene, that handles general functionality (e.g. it will call distinct functions to handle kinematics or dynamics, for instance). The *main script* is also in charge of calling *child scripts* in a cascaded way (with respect to the scene hierarchy). A *child script*, unlike the *main script*, is attached to a specific object in the simulation scene, and handles a particular part of the simulation. It is an integral part of its scene object, and will be duplicated and serialized, together with it. As such, it represents a perfectly portable and scalable control element: there is one single file containing the model definition together with its control or functionality, there is no compatibility issue across platforms, no need for explicit compilation, no conflict among several versions of the same model, model instantiation is implicit, etc. *Child scripts* can be executed in a threaded and non-threaded fashion. The threaded version of *child scripts* still keeps the advantages of the technique described in point 3 of section II, A: indeed, V-REP's thread scheduler handles threads in a way that makes them behave and appear as coroutines, which allow to precisely control the time at which the thread execution is switched back and forth, effectively allowing for an excellent synchronization with the *main script* or other *child scripts*. Additionally, each thread can programmatically request being set into a free-running mode (i.e. allowing them to temporarily behave as real threads). Embedded scripts can also be seen as a "glue component", that binds the various supported programming techniques around V-REP: *child scripts* can register ROS publishers/subscribers, they can open and handle communication lines (e.g. socket or serial port), launch executables, load/unload plug-ins, or start *remote API* server services (see point 4 hereafter). Embedded scripts include also callback scripts, used as low-level

customized joint controllers for instance. The functionality of embedded scripts can be extended by the user via two mechanisms: with Lua extension libraries, or with custom Lua functions registered through a plug-in.

- **Add-ons**. In a similar way as embedded scripts, add-on are supported in V-REP via Lua scripts. They can be used as stand-alone functions (convenient for writing importers/exporters), or as regularly executed code (convenient as a lightweight simulator customization method).

- **Plug-ins**. Plug-ins are used in V-REP as a convenient simulator customization tool. They can register custom Lua commands, allowing the execution of fast callback functions from within an embedded script. They can also extend the functionality of a particular simulation model or object. Often they also implement specific importers/exporters, or offer an interface to a specific hardware. The *remote API* interface as well as the ROS interface (see next items) are implemented via plug-ins.

TABLE I. COMPARISON OF THE FIVE PROGRAMMING TECHNIQUES SUPPORTED IN V-REP

	Embedded script	Add-on	Plug-in	Remote API client	ROS node
API mechanism	Regular API	Regular API	Regular API	Remote API	ROS
Supported programming language	Lua	Lua	C/C++	C/C++, Python, Java, Matlab, Urbi	Depends on ROS support
Available API functions	>280 functions	>270 functions	>400 functions	>100 functions	>150 services, publisher and subscriber types
API is user-extendable	Yes, with custom Lua functions	Yes, with custom Lua functions	Yes, V-REP is open source	Yes, remote API is open source	Yes, ROS interface is open source
Control entity is external and can be native	No	No	No	Yes	Yes
Simulation models are fully portable and scalable	Yes	No	No	No	No
Communication lag	None	None	None	Yes	Yes
Synchronous operation is supported*	Yes, inherent. No delays	Yes, inherent. No delays	Yes, inherent. No delays	Yes. Slow due to comm. lag	Yes. Slow due to comm. lag
Asynchronous operation is supported*	Yes	No	Yes	Yes	Yes
Can start, stop, pause and step a simulation	Stop, pause	Start, stop, pause	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step

* *Synchronous* in the sense that each simulation pass runs synchronously with the control entity

- **Remote API clients**. The *remote API* interface in V-REP allows interacting with V-REP or a simulation, from an external entity via socket communication. It is composed by *remote API* server services and *remote API* clients. The client side can be embedded as a small footprint code (C/C++, Python, Java, Matlab & Urbi) in virtually any hardware including real robots, and allows remote function calling, as well as fast data streaming back and forth. On the client side, functions are called almost as regular functions, with two exceptions however: remote API functions accept an additional argument which is the operation mode, and return a same error code. The operation mode allows calling functions as blocking (will wait until the server replies), or non-blocking (will read streamed commands from a buffer, or start/stop a streaming service on the server side). The ease of use of the *remote API*, its availability on all platforms, and its small footprint, makes it an interesting alternative to the ROS interface (see next item).

- **ROS [6] nodes**. V-REP implements a ROS node with a plug-in which allows ROS to call V-REP commands via ROS services, or stream data via ROS publishers/subscribers. Publishers/subscribers can be enabled with a service call, and also directly enabled from within V-REP, via an embedded script command.

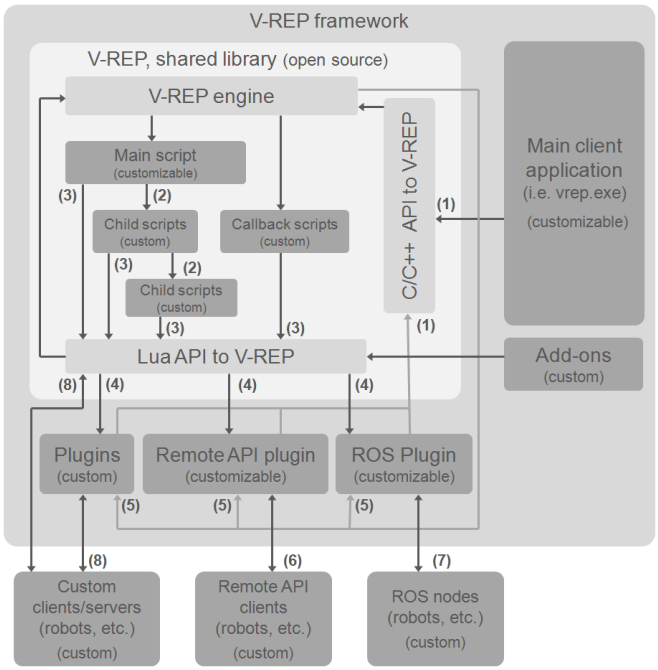

Figure 2. V-REP control architecture. Greyed items are control entities. (1) C/C++ API calls, (2) cascaded *child script* execution, (3) Lua API calls, (4) custom Lua API callbacks, (5) V-REP event callbacks, (6) remote API function calls, (7) ROS transit, (8) custom communication (socket, serial, pipes, etc.)

III. SIMULATION FUNCTIONALITY

V-REP is designed around a versatile architecture. There is no *main* or *central* functionality in V-REP. Rather, V-REP possesses various relatively independent functionalities, that can be enabled or disabled as required, also on a model-base.

Imagine a simulation scenario where an industrial robot has to pick-up boxes and move them to another location; V-REP computes the dynamics for grasping and holding the boxes and performs a kinematic simulation for the other parts of the cycle when dynamic effects are negligible. This approach makes it possible to calculate the industrial robot's

movement quickly and precisely, which would not be the case had it been simulated entirely using complex dynamics libraries. This type of hybrid simulation is justified in this situation, if the robot is stiff and fixed and not otherwise influenced by its environment.

In addition to adaptively enabling various of its functionalities in a selective manner, V-REP can also use them in a symbiotic manner, having one cooperate with another. In the case of a humanoid robot, for example, V-REP can handle leg movements by (a) first calculating inverse kinematics for each leg (i.e., from a desired foot position and orientation, all leg joint positions are calculated); and then (b) assigning the calculated joint positions to be used as target joint positions by the dynamics module. This allows specifying the humanoid motion in a very versatile way, since each foot would simply have to be assigned to follow a 6-dimensional path: the rest of calculations are automatically taken care of.

Functionality is related to specific *scene objects*, or to specific *calculation modules*, both of them are described hereafter.

A. Scene Objects

A V-REP simulation scene, or simulation model contains several *scene objects* or elemental objects that are assembled in a tree-like hierarchy. The following *scene objects* are supported in V-REP:

- **Joints:** *joints* are elements that link two or more *scene objects* together with one to three degrees of freedom (prismatic, revolute, screw-like, or spherical). They can operate in various modes (e.g. force/torque mode, inverse kinematics mode, etc.)
- **Shapes:** *shapes* are triangular meshes, used for rigid body simulation and visualization. They can be optimized for fast dynamic collision response calculation, as a grouping of primitive or convex shapes. Other *scene objects* or *calculation modules* heavily rely on *shapes* for their calculations (*proximity sensors*, the dynamics module, or the mesh-mesh distance calculation module for example).
- **Proximity sensors (Fig. 3):** they perform an exact minimum distance calculation to the part of a *shape* contained in a configurable detection volume [7], as opposed to simply performing detection based on rays. This results in a more continuous operation and thus allows for more realistic simulation.
- **Vision sensors:** *vision sensors* allow to extract complex image information from a simulation scene (colors, object sizes, depth maps, etc.). A built-in filtering and image processing function enables the composition of blocks of filter elements. *Vision sensors* make use of hardware acceleration for the raw image acquisition (OpenGL).
- **Force sensors:** they represent rigid links between *shapes*, that can record applied forces and torques, and that can conditionally break apart when a given threshold is overshot.
- **Graphs:** *graphs* can record a large variety of predefined or custom data streams. Data streams can then be displayed directly (time graph of a given data type), or combined with each other to display X/Y graphs, or 3D curves.
- **Cameras:** they allow scene visualization when associated with a viewport.
- **Lights:** *lights* illuminate a scene or individual *scene objects*, and directly influence *cameras* or *vision sensors*.
- **Paths:** they allow complex movement definitions in space (succession of freely combinable translations, rotations and/or pauses), and are used for guiding a welding robot's torch along a predefined trajectory, or for allowing conveyor belt movements for example.
- **Dummies:** a *dummy* is a reference frame, that can be used for various tasks, and is mainly used in conjunction with other *scene objects*, and as such, can be seen as a "helper."
- **Mills:** they represent customizable convex volumes that can be used to simulate surface cutting operations on *shapes* (e.g., milling, laser cutting, etc.).

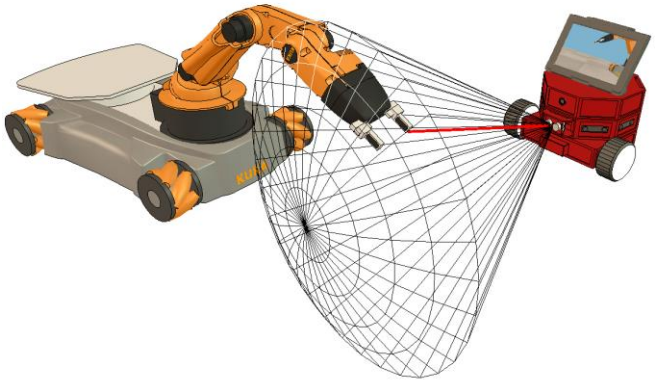

Figure 3. Proximity sensor in V-REP

A. Calculation Modules

Scene objects are rarely used on their own, they rather operate on (or in conjunction with) other *scene objects* (e.g. a *proximity sensor* will detect *shapes*). In addition, V-REP offers several *calculation modules* that can directly operate on one or several *scene objects*. Following are the main *calculation modules*:

- **Kinematics module:** allows kinematics calculations (forward/inverse) for any type of mechanism (branched, closed, redundant, containing nested loops, etc.). The module is based on calculation of the damped least squares pseudoinverse [8]. It supports conditional, damped/undamped, and weighted resolution.
- **Dynamics module:** allows handling rigid body dynamics calculation and interaction (collision

response, grasping, etc.) via the Bullet Physics Library [9] and the Open Dynamics Engine [10]. Dynamics-based simulations still being in its infant shoes and often based on approximations, it is important to not only rely on one single physics engine, in order to validate results. At the time of writing, a third, high fidelity physics support via Vortex Dynamics [11] is in preparation.

- **Collision detection module:** allows fast interference checking between any *shape* or collection of *shapes*. This module is fully independent from the collision response calculation algorithm of the dynamics module. It uses data structures based on a binary tree of oriented bounding boxes [12] for accelerations. Additional optimization is achieved with a temporal coherency caching technique.
- **Mesh-mesh distance calculation module:** allows fast minimum distance calculations between any *shape* (convex, concave, open, closed, etc.) or collection of *shapes*. The module uses the same data structures as the collision detection module. Additional optimization is also achieved with a temporal coherency caching technique.
- **Path/motion planning module:** handles holonomic path planning tasks and non-holonomic path planning tasks (for car-like vehicles) via an approach derived from the Rapidly-exploring Random Tree (RRT) algorithm [13]. Path planning tasks of kinematic chains are also supported.

For versatility the above modules are implemented in a general way, without making any assumptions on the underlying simulation scenes or models. The purpose of having them integrated in V-REP, instead of relying on external libraries is somewhat similar to the purpose of having embedded scripts, as described in section II, B: a vast majority of simulations or simulation models do not require any specific or high-end tool. They instead require a good set of basic tools. If those are integrated to the simulator, and their task definitions directly attached to simulation models, then models become extremely portable: distribution of a simulation model to a different machine or platform is done via a single model file; there is no need to distribute, recompile, install or reload a plug-in. In a similar way, this makes models very scalable too: duplicated models are automatically functional, without the need to modify any source code. The duplication process can even happen during simulation.

The traditional approach of extending functionality via a plug-in, in order to support a specific simulation model is of course also supported in V-REP.

IV. A Case Study

Sometimes there is no escape from using a controller that is separate from its simulation entity, typically when dealing with a robot's main controller, that can take very complex proportions. Or when the controller needs to run natively. But other times, is it really necessary to implement a plug-in for each small sensor, new feature or small function? Following three examples illustrate nicely the versatility and portability of simulation models offered in V-REP.

A. Simulation Model of a Laser Scanner

Fig. 4 shows a laser scanner simulation model in V-REP. The model is composed by a body or casing, a revolute joint, and a ray-type *proximity sensor* mounted on the *joint*. A non-threaded *child script* is attached to the sensor casing, and is in charge to move the *joint* by a given angle, read the *proximity sensor*, generate a line primitive in the scene (and an auxiliary point primitive where a detection occurred), then move to the next angular position. Since the *child script* runs non-threaded, it will have to process as many joint angle positions as the *joint* would have moved within one simulation step.

The model can be dragged and dropped into a scene, and will be immediately operational during simulation. The whole model fits into a single file directly usable on other platforms too, and compatible with current as well as future V-REP versions. The model can be duplicated as often as required, and its control code modified at will.

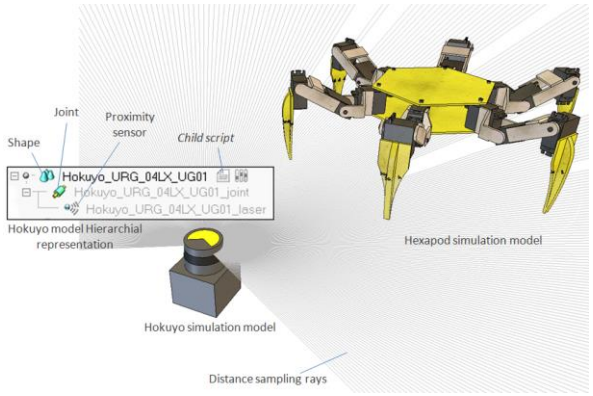

Figure 4. Laser scanner and hexapod model in V-REP

In a similar way, other such models can easily be created, be it a drawing pen, a paint nozzle, a gripper, a blob detection camera, or a whole robot.

B. Simulation Model of a Parallel Manipulator

Fig. 5 shows a parallel manipulator model controlled in forward kinematics from an external application that connects via the *remote API* to it. In order to correctly handle all the loop closure constraints, the model is handled via V-REP's kinematics module. Since all related kinematics task definitions are attached to the model, this model is self-contained too, and immediately duplicable and operational on other platforms too. Even physical scaling of the model, which is another feature that V-REP supports, will automatically adjust all kinematic tasks (among others), and keep kinematic resolution consistent - without the need to adjust any code.

C. Simulation Model of a Smart Human

Fig. 6 illustrates a simulation model of a human, performing path planning tasks between its current position and a desired target position. While the path planning task in

itself is computed by V-REP's path planning module, a *child script* attached to the model will trigger path planning calculations, actuate legs and arms, and correctly move the model along the calculated path. Here also, the model is fully self-contained and fully portable.

V. Conclusion

V-REP is introduced as a versatile and scalable simulation framework. By offering a multitude of different programming techniques for its controllers, and by allowing to embed controllers and functionalities in simulation models, it eases the programmers task and reduces the deployment complexity for the users.

Figure 5. Delta Arm manipulator model in V-REP

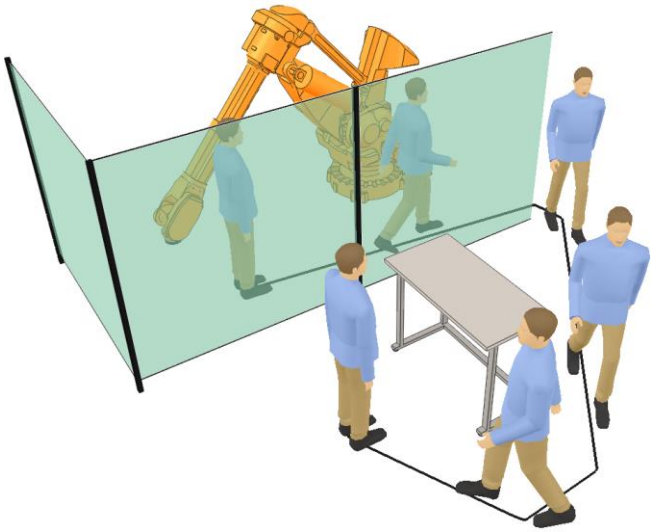

Figure 6. Path planning human model in V-REP

Currently V-REP has grown to a robust and widely used robot simulator and controller, present in the academic as well as industrial field. It performs tasks ranging from system verification, algorithm optimization, simulation of complex assembly chains in factory automation applications, to robot task planner and controller.

Acknowledgment

Eric Rohmer thanks the Sao Paulo Research Foundation FAPESP for its financial support.

References

[1] F. Kanehiro, H. Hirukawa, and S. Kajita, "Open HRP: Open Architecture Humanoid Robotics Platform," *Int. J. of Robotics Research*, vol 23, pp. 155-165, 2004

[2] N. Koenig, and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in Proc of *Int. Conf. on Intelligent Robots and Systems*, pp. 2149-2154, Sendai, Japan, Sept.-Oct. 2004

[3] O. Michel, "Webots: professional mobile robot simulation," *Int. J. Adv. Robot. Syst.*, vol. 1, pp. 39-42, 2004

[4] V-REP simulator : http://www.coppeliarobotics.com

[5] Lua: http://www.lua.org

[6] M. Quigley, B. Gerkeyy, K. Conleyy, J. Fausty, T. Footey, J. Leibsz, E. Bergery, R. Wheelery, and A. Ng, "ROS: an open-source Robot Operating System," *in Proc of IEEE Int. Conf. of Robotics and Automation*, Kobe, Japan, May 2009

[7] M. Freese, F. Ozaki, and N. Matsuhira, "Collision Detection, Distance Calculation and Proximity Sensor Simulation using Oriented Bounding Box Trees," *4th International Conference on Advanced Mechatronics*, pp. 13-18, Asahikawa, Japan, Oct. 2004

[8] C. W. Wampler, "Manipulator Inverse Kinematic solutions based on Vector Formulations and Damped Least Squares Methods," *in IEEE Trans. Syst., Man, Cybern.*, vol. 16, pp. 93-101, 1986

[9] Bullet physics library : http://www.bulletphysics.org

[10] Open dynamics engine: http://www.ode.org

[11] Vortex Dynamics: http://www.vxsim.com

[12] S. Gottschalk, M. C. Lin, and D. Manocha, "OBB-tree : a hierarchical structure for rapid interference detection," *ACM SIGGRAPH*, pp. 171-180, New Orleans, USA, Oct. 1996

[13] J. J. Kuffner Jr., "RRT-Connect: an Efficient Approach to Single-Query Path Planning," *in Proc of IEEE Int. Conf. of Robotics and Automation*, San Fransisco, USA, Apr. 2000