YARS: A Physical 3D Simulator for Evolving Controllers for Real Robots

Keyan Zahedi[1], Arndt von Twickel[2], and Frank Pasemann[2]

[1] MPI for Mathematics in the Sciences, Inselstrasse 22, 04103 Leipzig, Germany
zahedi@mis.mpg.de
[2] University of Osnabrück, Institute of Cognitive Science,
Albrechtstraße 28, 49076 Osnabrück, Germany
{arndt.von.twickel,frank.pasemann}@uni-osnabrueck.de

Abstract. This paper presents YARS (Yet Another Robot Simulator), which was initially developed in the context of evolutionary robotics (ER), yet includes features which are also of benefit to those outside of this field. An experiment in YARS is defined by a single XML file, which includes the simulator configuration, the (randomisable) environment, and any number of (mobile) robots. Robots are either controlled through an automatised communication, or by dynamically loaded C++ programs. Therefore, YARS, although still under active development, is comparable with commercial and open-source robot simulators which include a physics engine such as Webots and Breve but with a much stronger focus on requirements originating from the field of evolutionary robotics.

1 Introduction

The development of robots is time-consuming and, therefore, often very expensive. Especially in research, where budgets are limited, and various novel approaches are tested in hardware and software, simulators can play an important role in reducing development cost and time. Another advantage is that research groups can cooperate and exchange results, even if the physical robot platform is not available to all groups. These advantages only hold if the simulator does not require a high implementation effort for a new experiment and if the results obtained in simulation are portable to the physical platform.

In the context of evolutionary robotics (ER) [1] additional requirements must be fulfilled. A simulator is only advantageous if it is much (in the order of ten times) faster than real-time and if the results do not require additional porting effort. Another important criterion is the automatic set-up of the experiment after each evaluation to ensure compatibility of the fitness values.

There is a large number of robot simulators available, emphasising different aspects of robot simulation. Examples are Khepera 2.0 Simulator, Webots, Darwin2k, Adams, Yobotics, Gazebo, Breve, and USARSim [2,3,4,5,6,7,8,9,10]. So why is there a need for Yet Another Robot Simulator? The simulators mentioned above were reviewed by the authors before work on YARS was initiated, but not

S. Carpin et al. (Eds.): SIMPAR 2008, LNAI 5325, pp. 75–86, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Table 1. Comparison of simulators, evaluated with respect to evolutionary robotics as it is performed within the presented context (see fig. 1). The simulators listed here were chosen during the assessment phase of YARS because of their particular emphasis and as they were the most widely used simulators in the field of robotics. The entries ⊖ and ⊕ refer to a positive or negative evaluation, respectively. Evaluations given in brackets were not tested by the authors of this work, but obtained through available documentation. The evaluation of Webots refers to version 3, and might not be true for the current version 5. As none of the available simulators met all the requirements, YARS was initiated. Footnotes: 1) Available source refers to the possibility to include motor and sensor models, 2) Publication [10] states up to 300 times faster than real-time. This could not be validated with the examples provided in the evaluation version (Ver. 5, Mac OS 10.5.4, 2.5GHz Dual Core, 4GB RAM). Achieved maximum was ca. seven times faster than real-time. 3) Documentation states that Webots can be started as batch-process, 4) No statements made in the documentation, 5) Supervisor-concept available, but in Version 3 not well suited for evolution as performed in this context, i.e. with an external evolution- and evaluation-software, and the requirement to set and re-set the simulation externally.

Simulator	Speed	Optional GUI	Free	Auto. reset	Source avail.[1]
Webots	[⊖][2]	[⊕][3]	⊖	[⊖][5]	⊖
Breve	[⊕][4]	[⊕]	⊕	[⊖][4]	⊕
Adams	⊖	⊖	⊖	[⊖][4]	⊖
Darwin2k	[⊖]	[⊖]	⊕	[⊖][4]	[⊕]
Yobotics	[⊖]	⊖	⊖	[⊖]	⊖
YARS	⊕	⊕	⊕	⊕	⊕

chosen because of either cost, speed, or restricted usability for ER (for a discussion see tab. 1). The latter refers to setting up an experiment, and resetting it automatically after every evaluation of an individual. Additionally, customising and adding new sensors and actuators is either not possible or requires a high implementation effort, which excludes corresponding simulators for experiments such as those presented here. Furthermore, the evolution time can be reduced, if the evaluation of populations is distributed in a cluster. This is only possible if the simulator does not require GUI interactions and a running visualisation, two features which are not widely supported.

An additional feature which supports the distribution of the evaluation is the possibility to fully configure YARS either via command-line parameters, a configuration file or through network communication.

These requirements are a few of the features of YARS presented in this paper, which is organised as follows: the following section covers the approach of YARS and explains how it is well-suited for both, ER and mobile robot simulation in general. The third section explains the concept of YARS and describes its most prominent features. The fourth section introduces RoSiML, the XML description language of YARS. The fifth section gives an outlook of the future of YARS and the final section closes with a discussion.

2 Approach

In the context of ER, a simulator is used for four main reasons: 1. During evolution, hardware-damaging behaviours are likely to occur. 2. A simulator can run faster than real-time. 3. The state of the simulator can be precisely set by the experimenter, increasing the comparability of the fitness values of the individuals in a population. 4. For the analysis of the behaviour-relevant dynamics, it is essential to control all the parameters.

Yet, these reasons only hold if the simulator-reality gap does not lead to significant behavioural differences. Closing the gap is related to the precision of the simulator, which stands in contrast to the simulation speed, i.e. there is a trade-off. The central issue here is how precise must the simulator be to ensure the portability of the results and still remain fast enough to fulfil the requirements of ER.

In the approach followed here, it is not important if the characteristic curve of each motor is identical in simulation and reality, as a robust controller should compensate for these differences. Hence, the simulator is sufficiently precise if the observed behaviours in simulation and reality are qualitatively equivalent.

This has implications on the physics engine which is required in a number of experiments, e.g. walking [11,12,13] and gravity driven [14,15] machines. ODE [16] was chosen as the physics engine for YARS, because it is faster than real-time (depending on the complexity of the simulation, see next section for an example), numerically stable and well-documented. Numerically stable, in this case, means that the simulation will not crash, if the internal physics runs into computational singularities. For evolution, this type of simulator behaviour is very important. First, the singularities indicate hardware-damaging behaviour, which can be punished by the fitness-function. Second, for the next individual, the running simulator is simply reset and does not have to be restarted otherwise.

This advantage comes with a trade-off. ODE uses a first-order Euler integrator for the physics, a linear force model for the actuators, and only a Coulomb friction model, which together, result in a fast, numerically stable but not very precise simulation. In the approach followed here, this is not a drawback, as robust controllers are generated by including noise, exploiting the sensori-motor loop, and are evolved on an abstraction of the hardware. A sufficient abstraction is determined by comparing intermediate results on the simulated and real robot on the behaviour level. This approach leads to portable controllers, and hence, validates YARS for ER and robot development. The latter is briefly discussed in the next section, but the procedure is equivalent, except that the evolution is exchanged with other controller-generating or learning methods. An example is the use of YARS to simulate the RunBot [17] (see fig. 3(d)).

YARS has been used for over five years of research in numerous experiments. A small overview is presented in figure 3 (a more comprehensive list is given in [18]). This is only possible because it was designed to be general, while not requiring any programming knowledge. The last two statements are discussed in the following sections.

3 YARS

YARS was initially designed to connect to ISEE [18,19,20], an ER environment. Therefore, the early application was to connect to an external control program. For each robot, a UDP socket communication port is opened automatically and the morphological configuration of the robot is communicated through a handshake mechanism. Each description of a sensor and an actuator includes the mapping of the values. This can be used to adapt the pre- and post-processing of the controller automatically. Java and C++ classes are provided to connect other software by the same mechanism.

A reload mechanism in YARS supports on-line modification of the XML file. This enables the easy modification of the experiment's parameters and the observation of their influence without the need to halt the controller or to restart the simulation. This is an important feature in closing the simulator-reality gap. There is also the possibility to send the XML file through a socket communication port to YARS, which enables the co-evolution of environment, morphology and controller (see fig. 1). The same mechanism can be used to externally generate complex environments.

The properties discussed above, automatic communication and external configuration and control of YARS, enable YARS to connect to existing software

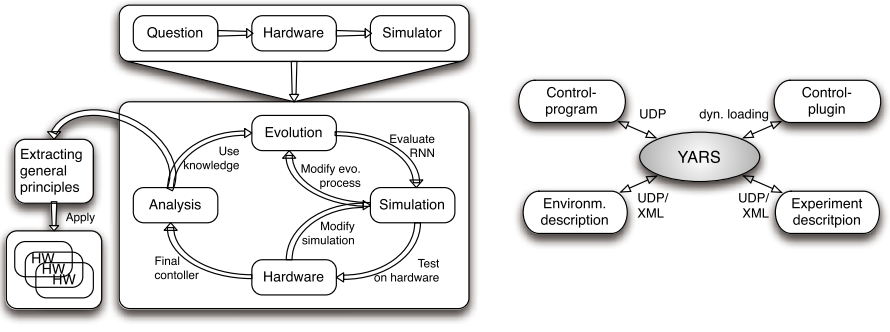


Fig. 1. Interactive evolutionary robotics. Left: The experiments begins with the definition of a question, e.g. insect-locomotion (see Octavio in text below and figure 3(c)). From this question, a well-suited hardware platform is defined, built and a simulation capturing the main physical properties is written in YARS. Recurrent neural networks are evaluated in simulation and the observations are used to modify the evolution parameters. Intermediate results are tested on hardware, and a comparison of the behaviour of the simulated and physical robot yields to modifications of the simulation parameters. Final results are extracted, generalised and may also be used as initial populations in other experiments. Right: YARS offers different possibilities for its controlled and configured. A control program can connect via a UDP connection to exchange sensor and motor commands, but can also be loaded dynamically during runtime. An experiment description is given as a command line parameter, or may be passed through a UDP port to YARS. The latter can be used to generate complex environment descriptions by an external tool.

with little effort, and are features from ER that make YARS attractive for robot simulation in general.

In ER, a large number of different controllers must be evaluated, until a good solution is found, i.e. the simulation speed is crucial. Currently, non-trivial simulations (e.g. Octavio, see fig. 3(c)) run between 10-70 times faster than real-time on a Pentium M 1.7 GHz. The high values result from the possibility to start YARS without visualisation or by reducing the refresh-rate of the rendering, while the lower bound is a consequence of the OpenGL rendering.

In closed-source simulators, actuators and sensors are problematic, as they can either not be extended at all, or require a large amount of implementation effort. YARS includes the most common sensors and actuators, which can be fully configured. Adding a new sensor or actuator is possible, as YARS is open-source (for details see sec. 5, Sensors)

Essential for the analysis of a controller is the ability to log data from the simulator. Sensor and motor values are available through the communication interface, pose of the objects can be written to a text file, and data can be displayed on-line, currently through a gnuplot interface.

RoSiML. Setting-up an experiment can be a very time-consuming process, and often requires programming knowledge or knowledge of 3D modelling languages such as VRML. We chose a different approach and designed our own description language: RoSiML (Robot Simulation Markup Language) [21,22]. This was done for one main reason. If the keywords of the description language are chosen with care, it is human-readable and does not require any programming knowledge.

Standardised 3D description languages, e.g. VRML and X3D, were not chosen, because they are too extensive in their possibilities, and require advanced programming knowledge. Their focus lies on scene descriptions and extending them to robotics requires heavy modifications, eliminating the advantage of available graphical development tools. XSLT [23] offers the possibility to convert

```
<RoSiML>
  <simulator>
    <!-- simulator configuration -->
    <!-- cam. specification -->
    <!-- keyboard commands -->
  </simulator>
  <environment>
    <box> ... </box>
    <box>
      <!-- box definition -->
      <coordinate x="7.5" y="0" z="0.15">
        <randomiseInterval method="global">
          <interval>
            <min x="6.5" y="-1" z="0.15"/>
            <max x="8.5" y="1" z="0.15"/>
          </interval>
          <probability x="0.5" y="0.5" z="0"/>
        </randomiseInterval>
      </coordinate>
    </box>
  </environment>
```

```
<movable>
    <body>
      <compound name="robot">
        <cylinder name="main body">
          <!--- cylinder definition  -->

          <!-- twice, with different pose -->
          <infraredDistanceSensor>
            <!-- sensor definition -->
          </infraredDistanceSensor>

          <!-- twice, with different pose -->
          <ldrSensor>
            <!-- sensor definition -->
          </ldrSensor>
        </cylinder>
        <!-- twice, for each side -->
        <sphere name="wheel right/left">
          <!-- sphere definition -->
        </sphere>
```

```
        <!-- twice, for each wheel -->
        <objectConnector>
          <hinge type="velocity">
            <!-- hinge definition -->
          </hinge>
          <source      name="wheel left"/>
          <destination name="main body"/>
        </objectConnector>
      </compound>
    </body>
  </movable>
</RoSiML>
```

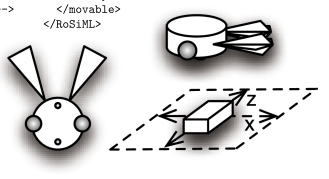


Fig. 2. YARS example: This figure shows snippets of RoSiML code and sketches of the robot. The snippets were taken from the SRN experiment discussed in the text below and shown in figure 3(b). An XML file is given to YARS either through command line, or through a UDP communication protocol (see fig. 1). A proximity sensor is simulated by five rays.

description languages, e.g. to convert VRML to RoSiML, but still requires additional manual modifications.

The first version of RoSiML was used in the German Research Foundation Priority Program 1125 as a general simulator description language to make descriptions exchangeable among the program members (e.g. [21,22]), independent of the simulation system.

The description of an experiment in RoSiML is divided into three main sections, the simulator, the environment and the movables (see fig. 2). The simulator section relates to the general configuration of YARS, e.g. update frequency of the physics and controller, keyboard commands, window size, camera position, etc. The environment section describes all static objects. Their position can be randomised at every reset, but remain fixed in their pose throughout the simulation run. The objects are basic geometric primitives (box, sphere, etc.) that are also used to define a movable. The movable section includes any number of movables, which are either controlled (see below) or passive. An example is a RoboCup [24] scenario in which groups of controlled robots act in a static environment interacting with a movable but otherwise passive ball. The concept of a moveable is detailed below.

Movables. A moveable is a generalised concept of a (mobile) robot. There are four possible types, *active*, *passive*, *controlled*, and *moving*, which are distinguished by their form of control and whether or not communication is established.

For each *active* movable, UDP socket communication is automatically established. Exceptions are *passive* movables, which do not require any form of control or communication. Both types are elaborated next.

In ER it is desirable to have a dynamic environment, i.e. other robots that interact with the robot and controller of interest. An example is an obstacle-avoidance controller that should not only avoid static but also moving obstacles. In this case, only the obstacle-avoider should be *active*, i.e. open to evolution/analysis/development, whereas the behaviour of the other robots remain unchanged. This case is covered by the *controlled* movable. YARS provides the possibility to dynamically load C++ classes. A string identifier in the XML file relates to the name of the C++ class, which contains the implementation of the controller. The *moving* movable is very similar to the *controlled* movable. The difference is that the outputs of the C++ program are forces which are applied directly to the body. The next paragraphs cover the concept of the morphology, sensors, and actuators of a movable.

Morphology. The morphology description of a movable is organised in a four-level tree (see fig. 2). The first-level node is named *body*, and it includes *compounds* and *connectors*. A compound is a group of connected rigid bodies or composites of rigid bodies, called objects for short. Connectors are active or passive joints between two objects. Inter-compound connectors are defined below the *body* node, intra-compound connectors within the compound. For each object, the physical parameters, e.g. weight and friction coefficients, must be specified.

Composites [16] allow the definition of complex rigid bodies. Trimesh objects will be included in a future release.

Sensors. Currently, different generic and specific sensors are implemented, both exteroceptive and proprioceptive. Exteroceptive sensors are attached and positioned relative to an object. Proprioceptive sensors are included in the actuator definition. The list of exteroceptive sensors includes: generic rotation sensor (3D compass), generic proximity sensor, two specific Sharp infra-red proximity sensors (DM2Y3A003K0F, GP2D12-37), generic light-dependent resistor sensor, and a generic directed camera sensor. The generic sensors are fully configurable, including noise. Available proprioceptive sensors are: joint (angular) position, joint (angular) velocity, joint force/torque, and an energy sensor. A special group are global sensors, which are not usually available in physical robots and which are used for the evaluation. Currently included are a global coordinate sensor and an ambient light sensor. Both were used to calculate the fitness in the examples discussed below.

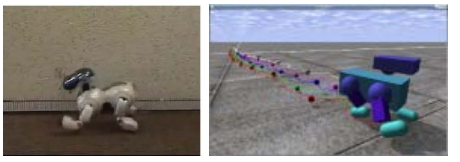
Custom sensors can be added through modification of the source code, or may result from the combination of available sensors, e.g. a laser-scanner can be simulated by an array of proximity sensors.

Actuators. Actuators connect two rigid bodies, and are positioned relative to their source. Possible actuators are hinge, hinge2 (combination of two hinges), slider, ball joint, and a complex hinge. They are configurable in torque/force, max. deflection, damping and spring properties, and noise.

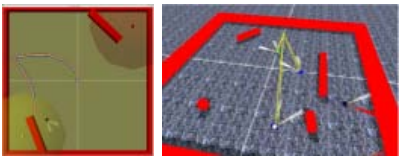
4 Examples

Aibo. The first example is an evolved neuro-controller for a fast quadrupedal walking behaviour [13] (see fig. 3(a)). The experimentation platform is the Sony Aibo robot [25]. A detailed 3D model of the Aibo is available, which enables the extraction of the body's proportions, but there are no specifications available about the motors and the weight distribution. This increases the difficulty of evolving a controller in simulation and porting it to the real hardware. Further challenges were the unknown friction coefficients and the non-trivial shape of the Aibo legs, which could not be simulated in detail. The solution to these problems were manifold. First, a few tests were conducted with the actual robot in order to get rough approximations of the motor torques. The second step was to find a good approximation for the weight distribution and morphology. The third step was to test intermediate evolution results on the real hardware, using the German Team framework [26], until the behaviour was qualitatively equivalent. With these techniques, the final solution only required minimal changes to a few synaptic weights in order to run on the physical hardware.

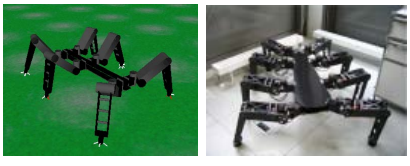
Octavio. Octavio is an example of a complex walking machine where a multitude of nonlinear mechatronic effects have to be taken into account in simulation to



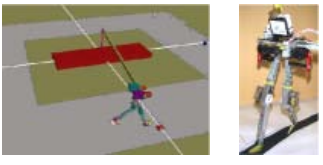
(a) Fast Quadrupedal Walking [13]. Left: The physical robot platform Right: Simulated Aibo.



(b) Left: The randomised environment used in the SRN adaptive light-seeker experiment. Right: A swarm experiment.



(c) Octavio is a modular eight-legged walking machine. First different controllers for single-leg control were evolved [11] and then combined in a walking machine (left). Right: The physical platform.



(d) RunBot is the fastest two-legged walking machine [17] (robot image was taken from the BCCN Göttingen site)

Fig. 3. YARS application examples

enable an efficient transfer of neuro-controllers to the real hardware. Octavio is a modular four-, six-, or eight-legged machine with autonomous legs with regard to control and energy supply (see fig. 3(c)). Each leg has three active and one passive joint of which each active one is equipped with a DC-motor-gear combination, a spring coupling, a pre-stressed spring, an angle, and a current sensor. Instead of using the motors as servo motors with the desired positions as input, controllers may take full advantage of the four states that the motor offers: forward torque, backward torque, relaxed and brake. Activation amplitude is determined by pulse-width modulation. On the one hand, this gives more power to the neuro-controller to e.g. save energy by making use of the relaxed mode; on the other hand it imposes a higher demand on the simulator in terms of transferability of controllers to hardware because effects like backlash, friction and inertia have a much more direct impact on performance. This is because they are not hidden from the neuro-controller by means of a black-box servo control. For successful transfers of neuro-controllers to hardware the usual strategy of reproducing weight distributions, including sensor- and motor noise etc. (see e.g. the Aibo example above) was not sufficient and the simulator therefore, had to be extended in several ways, of which a few examples are given here: simple models including static and dynamic joint friction which were derived from experiments, rotor inertia is taken into account as an energy storage that greatly influences the passive dynamics, pulse-width to maximum no-load velocity and

maximum stall torque mappings were determined experimentally and backlash effects were quantified and included in the simulator. Going beyond the transferability of controllers from simulation to Octavio a comparability of artificial neuro-controllers with biological controllers in e.g. the stick insects is desired. To this end, simple muscle models based on biological data are implemented to take into account the neuro-muscular transform.

A detailed elaboration on the implementation of the neuromuscular transform will be subject of future publications.

Adaptive Light-Seeker. The adaptive light-seeker with the Self-Regulating Neuron (SRN) model [18,27] demonstrates the randomisation possibilities of an environment in YARS (see fig. 3(b)). The SRN model is an extension of the standard additive neuron model, which is motived by Ashby's Homeostat [28]. Coupled in an embodied and situated recurrent neural network, it enables adaptivity within such structures. To demonstrate this, an environment was chosen in which a light source has to be found under varying light conditions. The robot cannot distinguish between a light source and the ambient light in the raw sensor data. YARS enables the randomisation of the pose of any object in the environment and the value of the ambient light. The former feature was used to first evolve a light-seeker without ambient light. The obstacles were randomised such that a static, non-explorative behaviour, e.g. cyclic movements with increasing radius, would not lead to a good fitness, as the environment changes from generation to generation. In the next step, the ambient light was randomised. The result is a pure feed-forward SRN network that is able to find a light source under varying ambient light conditions, as a result of the homeostatic properties of the SRN and the interaction with the environment [18].

Another example, RunBot [17], not in the context of evolutionary robotics, is shown in figure 3(d).

5 Outlook

The current state of YARS is well suited for experimentation in- and outside the field of ER (see examples given in figure 3). With XML as the description language, researchers who may not be familiar with programming are able to create their experiments within YARS. The communication is established automatically, and sources in Java and C++ are available to connect YARS to other programs. Controllers can also be written directly in C++ and loaded dynamically during the start-up of YARS. Hence, YARS can also be used without any additional software, such as ISEE. Recompiling YARS to test new controllers/morphologies/environments is not necessary. Nevertheless, there are still considerable improvements currently under development or in planing phase.

Modularisation/Plug-in Concept. The entire source of YARS is built into one monolithic executable, with the exception of the C++ controllers which are loaded dynamically during runtime. The next step of the YARS development

will split functional subgroups of YARS into shared libraries, which can then be easily exchanged without the need to recompile the entire system. Such functional groups are the physics engine, the visualisation, sensors, actuators, and logging. Each of them is discussed in the following paragraphs.

Visualisation. The ODE visualisation engine, drawstuff, was replaced by a faster OpenGL implementation which also supports multiple cameras. The next step is to make the visualisation optional at compile-time, and to allow the user to choose between different visualisations, i.e. none, minimal, such as OpenGL, or more comfortable such as e.g. wxWidgets. The more comfortable GUI will also allow graphical, interactive manipulation of the scene. As our focus was on exploiting the capabilities of simple sensors in the sensori-motor loop, textures for photo-realistic rendering has, so far, not been included, but will follow with the refactoring of the visualisation.

Physics. Current developments in the field of open-source physics engines tend towards impulse-based physics simulation [29]. Physics engines will be added after evaluation, if they meet the requirements and provide improvements.

Sensors/Actuator. A sensor and an actuator requires almost the same implementation effort in YARS. At this stage, first the XSD grammar has to be changed, followed by the parser, the internal representation, the simulation of the sensor/actuator, and finally the communication. Although well-structured, this is a considerable amount of implementation to add a new sensor/actuator. Under current planing is a plug-in concept to reduce this effort significantly and to support dynamic loading.

Logging/Plotting. The possibility to log and plot simulation variables is essential in order to analyse the quality of a controller or, as in the context in which YARS was developed, to understand the correlation between the neuro-dynamics and behaviour, given the sensori-motor loop. A template concept will support logging of data into any format, such that also exports to e.g. povray [30] will be possible.

Multi-OS. YARS runs on Linux (gcc 4.x), and is currently ported to Mac OS X 10.5 and Win32.

6 Discussion

YARS is a very flexible, highly configurable robot simulator. If physics is required and the on-line visualisation does not need to be highly sophisticated, it is currently, to the best of the authors' knowledge, the fastest available simulator. YARS' main contribution is simulation speed, but keeping the simulator-reality gap in mind, ensuring quick portability of simulation results to the physical platforms. Other contributions of the YARS development are easy integration of new sensors and actuators, and concerning evolutionary robotics; automating of communication, randomisation of the environment, and the possibility to reset

the experiment through a communication channel. The experiment description file may also be passed to YARS through socket communication, which enables co-evolution of the environment, morphology and controller and enables generation of complex environments by external programs. YARS has also proven to be useful in experiments outside the field of ER, as in e.g. RunBots.

Therefore, YARS already has many desired features for research which is currently discussed in the field of ER, but also supports robotics development outside this field.

YARS is open-source and available from sourceforge:
`http://sourceforge.net/projects/yars/`.

Acknowledgements. Steffen Wischmann contributed to the implementation of the early version of the YARS core. The generic communication interface between YARS and Hinton was designed and implemented by Björn Mahn. Verena Thomas implemented the dynamical loading of control programs, the OpenGL visualisation, and the virtual camera sensor.

This work was partly funded by the DFG grants CH 74/9, PA 480/4, PA 480/6.

References

1. Floreano, D., Urzelai, J.: Evolutionary robots with on-line self-organization and behavioral fitness. Neural Netw. 13(4-5), 431–443 (2000)
2. Michel, O.: Khepera simulator 2.0 (last visited: August 2008), `http://diwww.epfl.ch/lami/team/michel/khep-sim/`
3. Michel, O.: Webots (last visited: August 2008), `http://www.cyberbotics.com/products/webots/`
4. Leger, C.: Darwin2k (last visited: August 2008), `http://darwin2k.sourceforge.net/`
5. MSC Software: Adams (last visited: August 2008), `http://www.mscsoftware.com/products/adams.cfm`
6. Yobotics Inc.: Yobotics website (last visited: August 2008), `http://yobotics.com/`
7. Koenig, N., Polo, J.: Gazebo (last visited: August 2008), `http://playerstage.sourceforge.net/index.php?src=gazebo`
8. Klein, J.: breve: a 3d simulation environment for the simulation of decentralized systems and artificial life. In: Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems, Sydney, Australia. MIT Press, Cambridge (2002)
9. Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C.: Usarsim: a robot simulator for research and education. In: Proceedings of the 2007 IEEE Conference on Robotics and Automation (2007)
10. Michel, O.: Webots: Professional mobile robot simulation. Journal of Advanced Robotics Systems 1(1), 39–42 (2004)
11. von Twickel, A., Pasemann, F.: Reflex-oscillations in evolved single leg neurocontrollers for walking machines. Natural Computing 6(3), 311–337 (2007)
12. Manoonpong, P.: Neural Preprocessing and Control of Reactive Walking Machines Towards Versatile Artificial Perception-Action Systems. In: Cognitive Technologies. Springer, Heidelberg (2007)

13. Markelić, I., Zahedi, K.: An evolved neural network for fast quadrupedal locomotion. In: Xie, M., Dubowsky, S. (eds.) Advances in Climbing and Walking Robots, Proceedings of 10th International Conference (CLAWAR 2007), pp. 65–72. World Scientific Publishing Company, Singapore (2007)
14. Popp, J.: sphericalrobots (last visited: August 2008), http://www.sphericalrobots.org
15. Wischmann, S., Hülse, M., Pasemann, F.: (Co)Evolution of (de)centralized neural control for a gravitationally driven machine. Advances in Artificial Life, 179–188 (2005)
16. Smith, R.: ODE (last visited: August 2008), http://www.ode.org
17. Geng, T., Porr, B., Wörgötter, F.: Fast biped walking with a sensor-driven neuronal controller and real-time online learning. The International Journal of Robotics Research 25(3), 243–259 (2006)
18. Ghazi-Zahedi, K.M.: Self-Regulating Neurons. A model for synaptic plasticity in artificial recurrent neural networks. PhD thesis, University of Osnabrück (2008)
19. Zahedi, K., Hülse, M.: ISEE – integrated structure envolution environment (last visited: August 2008), http://sourceforge.net/projects/isee/
20. Hülse, M., Wischmann, S., Pasemann, F.: Structure and function of evolved neurocontrollers for autonomous robots. Connection Science 16(4), 294–296 (2004)
21. Laue, T., Spiess, K., Röfer, T.: Simrobot — a general physical robot simulator and its application in robocup. In: Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y. (eds.) RoboCup 2005. LNCS (LNAI), vol. 4020, pp. 173–183. Springer, Heidelberg (2006)
22. Mayer, N., Boedecker, J., da Silva Guerra, R., Obst, O., Asada, M.: 3d2real: Simulation league finals in real robots. In: RoboCup 2006: Robot Soccer World Cup X, pp. 25–34 (2007)
23. Clark, J., Lipkin, D., Marsh, J., Thompson, H., Walsh, N., Zilles, S.: XSL Transformations (XSLT) Version 1.0. W3C (1999)
24. The RoboCup Federation: Robocup official site (last visited: August 2008), http://www.robocup.org
25. Sony: Sony aibo europe (last visited: August 2008), http://support.sony-europe.com/aibo/
26. Röfer, T., Laue, T., Burkhard, H.D., Hoffmann, J., Jüngel, M., Göhring, D., Düffert, U., Spranger, M., Altmeyer, B., Goetzke, V., von Stryk, O., Brunn, R., Dassler, M., Kunz, M., Risler, M., Stelzer, M., Thomsas, D., Uhrig, S., Schwiegelshohn, U., Dahm, I., Hebbel, M., Nistico, W., Schumann, C., Wachter, M.: Germanteam 2004 (2004) (last visited: August 2008), http://www.germanteam.org/GT2004.pdf
27. Zahedi, K., Pasemann, F.: Adaptive behavior control with self-regulating neurons. In: Lungarella, M., Iida, F., Bongard, J.C., Pfeifer, R. (eds.) 50 Years of Aritficial Intelligence. LNCS (LNAI), vol. 4850, pp. 196–205. Springer, Heidelberg (2007)
28. Ashby, W.R.: Desgin for a brain. Chapman & Hall Ltd., London (1954)
29. Bender, J.: Impulse-based dynamic simulation (last visited: August 2008), http://www.impulse-based.de/
30. Persistence of Vision Raytracer Pty. Ltd.: Povray – the persistence of vision raytracer (last visited: August 2008), http://www.povray.org/