A Unified Simulator for Self-Reconfigurable Robots

David Christensen, David Brandt, Kasper Stoy, and Ulrik Pagh Schultz

Abstract—Generic simulation platforms such as Player/Stage are an essential tool in mobile robotics, but until now no similar platforms have been available for the field of self-reconfigurable robots. We here present a generic simulation platform for modular, self-reconfigurable robots: the Unified Simulator for Self-Reconfigurable Robots (USSR). USSR is based on a physics engine, allowing simulation of both self-reconfiguration and dynamic interaction with the environment. The simulator is implemented as a framework that provides numerous components that can be combined to form new or existing modular robots, allowing easy experimentation: USSR currently includes support for the ATRON, Odin, and M-TRAN modular robots.

I. INTRODUCTION

Simulators are an essential tool in robotics, allowing initial experiments with hardware design and controller programming to be performed at a relatively low cost. Nevertheless, developing a simulator from scratch is a significant development task, for which reason generic simulation environments such as Player/Stage and Gazebo have gained wide popularity in the mobile robotics community [7], [8], [13]. There are however no generic simulation environments available within the field of self-reconfigurable, modular robotics; dedicated simulators are typically developed along with each robot. This lack of tool support puts a massive burden on the developer, making the task of experimenting with new designs for self-reconfigurable robots even harder.

As part of an overall effort to make self-reconfigurable robots more accessible, we have developed a simulator for self-reconfigurable robots. This simulator, named the “Unified Simulator for Self-Reconfigurable Robots” (USSR), is designed to support a wide variety of self-reconfigurable robots. Moreover, an important design goal is to make it easy to experiment (in simulation) with new designs for self-reconfigurable robots. In essence, USSR is a framework that provides a number of primitive building blocks that can be combined to form different self-reconfigurable robots. Our vision is that USSR can serve as an open platform for creating simulators for modular and self-reconfigurable robots.

The main contribution of this work is the design and implementation of a simulator for modular robots that supports different types of robots and moreover makes it easy to add new types of robots with a minimal implementation effort. Our simulator currently supports the ATRON, Odin, and M-TRAN self-reconfigurable robots [12], [6], [15]. The simulation is based on a physics engine and hence allows simulation of dynamic interaction with the environment, such as friction and object manipulation, but is also precise enough to simulate self-reconfiguration. This feature enables experiments where robots interact with a simulated environment and use self-reconfiguration to better adapt to the environment. Support for new robots can easily be added to the simulator by combining physical shapes with sensors, actuators, communication devices, and connectors; special attention has been paid to providing a flexible model for connectors, since this is a critical feature of self-reconfigurable modular robots. The simulator is implemented in Java but provides a lightweight interface for controllers written in C, which for example allows existing controllers for the real ATRON robot to run in USSR.

Self-reconfigurable robots: A self-reconfigurable robot is a robot that can change its own shape. Self-reconfigurable robots are built from multiple identical modules that can manipulate each other to change the shape of the robot [2], [12], [15], [18], [20], [24]. The robot can also perform tasks such as locomotion without changing shape. Changing the physical shape of a robot allows it to adapt to its environment, for example by changing from a car configuration (best suited for flat terrain) to a snake configuration suitable for other kinds of terrain.

Example: As a concrete example of a simulation implemented in USSR, consider the image sequence shown in Figure 1: (1) an ATRON robot in an initial “8-shape” is blocked by a wall, (2) the robot self-reconfigures to a snake, (3) traverses the wall, (4) reverts to the 8-shape, (5) self-reconfigures to a car, (6) reaches the final car configuration in a tipped-over state, (7) raises the car to stand on its wheels, and (8) drives away. This simulation is only possible in an environment that not only simulates physical effects such as friction and gravity but also allows self-reconfiguration to be simulated in the same environment. Since the simulator provides a uniform physics-based environment for all the robots it supports, the same simulation could easily have been carried out with several different modular robots.

Availability: The USSR simulator is freely available as open source software under the BSD licence and includes numerous simulation examples and documentation both for the novice programmer, who simply wants to experiment in simulation with existing designs for self-reconfigurable robots, as well as for the more experienced programmer, who can use USSR as a framework for rapidly prototyping simulation environments for new self-reconfigurable robots. The simulator runs on Windows, Linux, and Mac OS X.
II. RELATED WORK

The Player, Stage, and Gazebo projects together provide a versatile simulation environment that is widely used both in academia and industry [7], [8], [13]. Player provides a generic, robot-independent platform for writing controllers, allowing a controller running on a workstation to control both simulated and real hardware through the same interface. Stage and Gazebo are both compatible with the Player interface; Stage provides a simulated 2D environment for large populations of mobile robots, whereas Gazebo provides a physics-based 3D environment for a smaller number of mobile robots.

USSR does not have a Player interface: most modules for self-reconfigurable robots are low-end embedded systems without a network connection, obviating one of the primary advantages of using Player. Gazebo could have been used as a platform for implementing the simulation, but since modular robots can be built from thousands of modules, it is critical that the simulator provides different degrees of precision to enable scalability to large populations of robots. Moreover, one of the primary challenges in implementing a simulator for self-reconfigurable robots is implementing the module connector mechanism, so we expect that the existing Gazebo components only would provide a limited advantage. Recent alternatives to Player and Gazebo include Webots [4] and Microsoft Robotics Studio [10], but for similar reasons we do not believe that these platforms provide any additional advantages for implementing modular robots. In general, these platforms are designed for simulating one or more physically independent robots, whereas the strength of USSR is in simulating physically connected modular robots, in particular self-reconfiguring robots where the connections between modules change throughout the simulation. The Webots platform has been used to simulate the modular, self-reconfigurable robot YaMoR [14], and from this point of view could have made an interesting starting point for USSR; an important goal of USSR is however to be an open-source platform generally usable by the community, which precludes using a commercial platform such as Webots.

Many projects on self-reconfigurable robots also include the development of a simulator [12], [15], [24], [9]. The development of a simulator is typically a means to an end, namely making real self-reconfigurable robots. As an example, consider the ATRON robot: a transition-based simulator has been used for numerous simulated experiments. This simulator scales to thousands of modules, but has severe limitations with regards to the actual physical behavior of the modules, making it impossible to simulate dynamic interaction with the environment such as friction, stability, and motor torque. Nevertheless, in the future we hope to be able to integrate the transition-based simulation engine with USSR, enabling the developer to switch between different modes of simulation when working with ATRON modules.

Scalability is a major issue in any simulator for modular robots. This issue is particularly critical in the Claytronics project where large numbers of microscopic modules are being simulated using DPRSim [22], [17], [5], [9]. DPRSim is capable of simulating hundreds of thousands of modules, in part due to the simple geometry of the Claytronics modules, in part due to the availability of different underlying engines providing different degrees of realism. Although USSR can easily be made to simulate hundreds of modules (as demonstrated by our usage examples, see Section IV), we would probably need to implement similar module-specific optimizations to scale to larger numbers of modules.
III. A SIMULATOR FOR MODULAR ROBOTS

A. Overview

The Unified Simulator for Self-reconfigurable Robots (USSR) provides an extensible physics-based simulation and visualization environment for modular robots. USSR is implemented using a layered architecture providing several layers of abstraction, enabling e.g. replacing the physics engine with a transition-based engine. Concretely, USSR is an object-oriented framework for building modular robot simulators, implemented in Java. The framework provides numerous building blocks that can be combined or extended to implement specific modular robots. Concretely, at the time of writing we have implemented three modular robots in USSR: ATRON, Odin, and M-TRAN [7], [8], [13].

B. Architecture

USSR is implemented using a layered architecture: the robot layer, the model layer, the physics and visualization layer, and the implementation layer. The architecture is described in detail in Section V, here we provide an overview. Specific simulations are implemented in terms of a robot layer, which defines the controller API of a specific robot (ATRON, Odin, . . . ). The controller runs in a separate thread and can be implemented in Java or C or controlled remotely over a socket. The model layer provides a robot-independent API or writing controllers and moreover implements the larger part of the communication model (only the computation of whether a signal reaches a specific recipient is defined at lower layers). The physics and visualization layer provides a simulated world and a set of components from which modular robots can be constructed. This layer is implemented in terms of the implementation layer which basically consists of the open-source JME framework; JME is currently based on ODE and OpenGL with a PhysX implementation underway [16], [11], [23].

C. USSR framework

USSR is essentially an object-oriented framework for constructing simulators for modular robots. Most modular robots can be simulated in a standard physics environment with gravity, ground, and obstacles, so emphasis has been put on providing components that can be combined to form different self-reconfigurable modular robots.

The physical appearance of a simulated module is constructed from a number of geometric shapes, either standard 3D shapes such as spheres or boxes, or (when appropriate) CAD models. The geometric shapes are combined into module parts that are connected using actuators; USSR currently provides linear and rotational actuators. Each module part is equipped with sensors, communication elements, and connectors. The set of available sensors is currently fairly limited, as is typical for the kinds of sensors available on real self-reconfigurable robots. Communication can either be directional infrared, wired (through a connector), or omnidirectional radio; directional transmitters are typically mounted on the connector, but this need not be the case.

D. Extending USSR

Extending USSR to support new modular robots can be done by reusing existing components, in which case implementing a new robot basically amounts to implementing a module constructor using USSR and JME primitives. For example, the ATRON robot is defined in roughly 250 lines of code whereas the six module types currently supported for Odin are defined in roughly 300 lines of code. Figure 3 shows parts of the ATRON implementation: the physics-engine independent description of geometry and properties such as connector type and connection distance, as well as the physics-engine specific description of the main actuator. Note that the implementation of the main actuator has been carefully tuned to match the physical behavior of a real ATRON module; this work would have to be done again if we were to use a different physics and visualization layer with physics support, but would not be needed for e.g. a transition-based physics and visualization layer. When adding a new robot to the simulator, the set of available components may however be insufficient, in which case new components must be implemented using JME primitives. In this case the task is open-ended and requires knowledge of the physics engine, but is nonetheless limited to those components not already found in the USSR framework.

IV. EXAMPLES OF USE

In addition to the snake-car reconfiguration and locomotion example described at the beginning of this paper, we have performed a number of simulated experiments with the
• Fragment of the physics-engine independent description of ATRON, in the robot layer:

```java
RobotDescription description = new RobotDescription("ATRON");
// Define two hemispheres
MeshShape hemi1, hemi2;
hemi1 = new MeshShape("ATRON", 0.935f, new Vector(0,0,0),new Rotation(0,pi,pi/4));
hemi1.setProperty("north", true);
hemi2 = new MeshShape("ATRON", 0.935f, new Vector(0,0,0),new Rotation(0,0,pi/4));
hemi2.setProperty("north", false);
hemi1 setColor(Color.BLUE); hemi2 setColor(Color.RED);
ModuleComponentDescription hemi1desc = new ModuleComponentDescription(hemi1);
ModuleComponentDescription hemi2desc = new ModuleComponentDescription(hemi2);
description.addModuleComponent(hemi1desc);
description.addModuleComponent(hemi2desc);
// Connectors
ConnectorDescription.Properties properties = new ConnectorDescription.Properties();
properties.setGeometry(new Geometry[] {new ConeShape(2.5f*0.005f,0.05f)});
properties.setType(ConnectorDescription.Type.MECHANICAL_CONNECTOR_RIGID);
properties.setMaxConnectionDistance(0.03f);
Color[] colors = new Color[] {Color.BLACK, Color.WHITE, Color.BLACK, Color.WHITE};
Vector[] northPos = ...;
Quaternion[] northRotQ = ...;
ConnectorDescription[] northConnectors = new ConnectorDescription[4];
for(int i=0; i<4; i++)
northConnectors[i] = new ConnectorDescription(properties, northPos[i],
new Rotation(northRotQ[i]), colors[i]);
hemi1desc.setConnectors(northConnectors);
// ... similarly for southern hemisphere
```

• Method from the JME-specific description of ATRON properties, in the physics and visualization layer:

```java
private void addCenterActuator(Module module) {
  JMERotationalActuator centerActuator = new JMERotationalActuator(simulation,"center");
  module.addActuator(new Actuator(centerActuator));
  DynamicPhysicsNode north = ((JMEModuleComponent) module.getComponent(0)).getModuleNode();
  DynamicPhysicsNode south = ((JMEModuleComponent) module.getComponent(1)).getModuleNode();
  centerActuator.attach(south,north);
  float velocity = 6.28f/6f;
  centerActuator.setControlParameters(3f, velocity, 0, 0);
  centerActuator.setMaxStopAcceleration(15f);
  centerActuator.setErrorThreshold(0.001f);
  centerActuator.setDirection(0, 0, 1);
  centerActuator.setPIDParameters(50, 100, 1);
}
```

Fig. 3. Parts of the ATRON implementation in the robot and physics and visualization layers.

ATRON, Odin, and M-TRAN implementations in USSR. The ATRON implementation is by far the most mature in terms of fidelity, but the other implementations are complete enough to ensure the generality of the USSR framework: Odin is a heterogeneous robot with a tethered communication system and the physical structure of M-TRAN is significantly different from that of the ATRON.

A. ATRON

The snake-car reconfiguration and locomotion example described at the beginning of this paper was programmed in Java: each controller runs in a separate thread and interacts with the hardware using the same API as is available on the real modules. Indeed, the transformation sequence from 8-shape to car was manually translated from the original controller for the real ATRON hardware [1]. This transformation sequence is shown in Figure 4, although upside-down compared to the simulated self-reconfiguration of Figure 1 pictures 4–6. The original C controller used for this self-reconfiguration sequence however also works with USSR using the JNI-based C interface. The only significant modification to the controller program was the addition of a context parameter to every API call and the use of a macro to access global variables. In effect, the same controller program performing a self-reconfiguration runs both in the simulator and on the real hardware with identical results.

USSR has been used for experiments with on-line learning of locomotion in ATRON robots [3]. Here, the simulator was used to train controllers to perform locomotion of ATRON robots in different configurations, one of which is shown in Figure 5. USSR was validated by successfully transferring gaits from simulation to the physical robot. Furthermore, when performing learning directly on the physical robots,
the learning strategy found gaits equivalent to those found in simulation.

USSR was also used for prototyping a virtual machine for ATRON modules implemented in C and running both in the simulator and on real hardware (in a preliminary version) [19]. The virtual machine provides a role-based middleware for mobile programs, using distributed control diffusion to install behaviors on specific modules depending on what role they are playing in the robot. The primary example, obstacle avoidance independently of the concrete car configuration, is shown in Figure 5. Using the simulator here provided massive advantages in terms of debugging the virtual machine and the distributed communication.

B. Odin

The Odin modular, heterogeneous, reconfigurable robot is currently being developed at the University of Southern Denmark [6], [21]. USSR is being used throughout the development process: experiments with new types of modules are easily performed in USSR by combining various components with the standard Odin connector components, locomotion patterns are investigated using the simulator, and communication strategies are similarly being investigated using the simulator which both provides an immediate graphical feedback as well as detailed logging information.

As an example of simulations of Odin robots, consider the robots shown in Figure 6: different ways of performing locomotion are investigated using a snake, a simple car, and a “self-deformable” sphere. Moreover, a larger experiment

---

**Fig. 4**. Self-reconfiguration on the real ATRON using the original controller subsequently used with USSR [1]. Frames from movie correspond to frames 4–6 in Figure 1, although the robot is upside-down in the simulation since moving as a snake causes the robot to rotate.

**Fig. 5**. ATRON examples: locomotion with on-line learning (left), real robot moving with the gait learned in simulator (center), and obstacle avoidance programmed using roles and mobile code (right).

**Fig. 6**. Simulations of the Odin robot in various configurations: locomotion as snake using rotational actuators, locomotion using a wheel module, self-deforming sphere for locomotion experiments, and a self-deforming cube built from 303 modules.

**Fig. 7**. Simulation of the M-TRAN robot in a snake configuration (left) and simultaneously with ATRON and Odin (right).
with a self-deformable cube structure is also included; the robot is built from 303 Odin modules (ball joints and linear actuators), each with a controller running in a separate thread. Each module communicates with its neighbors to control the actuation, creating pulsing movements throughout the robot. When run on a 2.4GHz dual-core MacBook Pro with 4Gb of RAM, the simulation runs at roughly 1/10 realtime speed. These experiments not only demonstrate that USSR supports the Odin robot, but also that it scales to large numbers of modules even without robot-specific optimizations.

C. M-TRAN

To demonstrate the generality of the USSR framework, we have recently added support for the M-TRAN self-reconfigurable robot [15]. The implementation is preliminary in the sense that the fidelity compared to the real modules has not been verified and moreover there is only the generic model-level API available for programming the robot. Nevertheless, we expect to further extend this simulation to include the complete API of the M-TRAN robot. As a simple example of using USSR to simulate M-TRAN, we have implemented a snake robot that performs locomotion, shown in Figure 7 both alone and simultaneously with the Odin and ATRON robots. The three-robot example serves to demonstrate how USSR enables simulations that study interactions between several different self-reconfigurable robots. This also includes hypothetical experiments where e.g. a special M-TRAN module is equipped with an ATRON connector thus forming a heterogeneous robot.

V. ARCHITECTURE

The software architecture of USSR has been designed to provide programmers with a high-level model of the system while at the same time maximizing the flexibility with regards to future extensions. USSR is structured in four layers upon which concrete simulations can be implemented, as illustrated in Figure 8.

The robot layer (ATRON, Odin, ... ) provides an API for setting up and controlling simulations for a specific robot. Each implementation of a robot typically provides a dedicated controller expressed in terms of the more general controller model. The controller runs in a separate thread and can be implemented in Java or C or controlled remotely over a socket. Controllers implemented in C can be made to run in a Java thread inside the simulator by using a JNI-based wrapper; this approach preserves the scalability of the simulator but does require all API calls to include a context parameter that defines the identity of the module. Moreover, access to module state must be done using a macro that allows the simulator framework to provide thread-local state for each module. Alternatively, a platform-neutral interface for writing controllers is provided that allows any controller to connect to the simulator using a TCP socket; for small numbers of modules this approach is however less efficient compared to running all controllers using threads.

The model layer provides a physics- and robot-independent abstraction over the underlying physics and visualization layer. This abstraction not only allows controllers to be defined independently of the underlying layer, but also allows some parts of the simulator, such as the communication model, to be partially implemented in a generic way. Concretely, for the communication model, all aspects of infrared- and radio-based communication can be defined abstractly, except the computation of whether a given transmitted signal reaches the recipient. This way, if the current physics-based simulation layer is replaced by a transition-based layer, the larger part of the communication model can still be used in the implementation.

The physics and visualization layer provides a simulated world including module components, various entities such as the ground and obstacles, and a concretization of the communication model. This layer is currently implemented using JME [11], see below. Module components are described in terms of the objects from which they are built (basic 3D shapes or CAD models), and include both the physical appearance, actuators such as joints and connectors, and sensors such a proximity or accelerometers. The static composition,
dynamic behavior, and visualization is at this level programmed in Java using JME. The concrete robot modules are assembled from individual components by module-specific construction classes, which typically are concerned with the 3D-composition of individual components.

The implementation layer provides a physics engine, visualization, and programming environment. As an implementation layer, we use JME (“JMonkey Engine”), an open-source framework for implementing games in Java [11]. JME is by design physics-engine and graphics-layer independent, but currently only supports ODE and OpenGL [16]. The underlying physics-engine (ODE) provides simulation of collisions and rigid-body dynamics. Support for PhysX, which provides hardware-based acceleration, is however underway and would be directly usable with USSR once it is completed [23]. In addition, since USSR is implemented in Java, programmers have the entire Java API at their disposal, including e.g. the Swing framework for implementing user interaction.

VI. CONCLUSION AND FUTURE WORK

The Unified Simulator for Self-Reconfigurable Robots (USSR) provides a reusable framework for implementing simulations for self-reconfigurable robots. USSR currently supports three different modular robots, and already makes it significantly easier for researchers and students alike to experiment with new uses and designs for modular robots. Nevertheless, now that USSR has been made available as an open-source platform for creating simulators, our goal is to extend USSR to include more existing self-reconfigurable robots, hopefully in collaboration with other researches within the area.

In terms of future work, adding support for more modular robots will provide useful experiences with the design and architecture of USSR, allowing us to further refine the implementation. As mentioned earlier, we are also interested in allowing specific robots to make use of a dedicated simulation engine, for example based on a transition-based system. Moreover, adding support for debugging similarly to the facilities in DPRSim would significantly improve the usability of the simulator [17], [5], and can e.g. for communication be done at the model level and hence independently of both the concrete robot and the specific simulation engine. In general, we expect to be able to provide numerous such robot- and simulation engine-independent facilities.

VII. ACKNOWLEDGEMENTS

This research is funded by the Danish Council for Technology and Production through the project “Morphing Production Lines”. The authors also want to thank Nicolai Dvinge for having contributed useful ideas to the paper.

REFERENCES


