Simbeeotic: A Simulator and Testbed for Micro-Aerial Vehicle Swarm Experiments

Bryan Kate
Harvard University
Cambridge, MA, USA
bkate@eecs.harvard.edu

Jason Waterman
Harvard University
Cambridge, MA, USA
waterman@eecs.harvard.edu

Karthik Dantu
Harvard University
Cambridge, MA, USA
kar@eecs.harvard.edu

Matt Welsh
Google, Inc.
Seattle, WA, USA
mdw@mdw.la

ABSTRACT

Micro-aerial vehicle (MAV) swarms are an emerging class of mobile sensing systems. Simulation and staged deployment to prototype testbeds are useful in the early stages of large-scale system design, when hardware is unavailable or deployment at scale is impractical. To faithfully represent the problem domain, a MAV swarm simulator must be able to model the key aspects of the system: actuation, sensing, and communication. We present Simbeeotic, a simulation framework geared toward modeling swarms of MAVs. Simbeeotic enables algorithm development and rapid MAV prototyping through pure simulation and hardware-in-the-loop experimentation. We demonstrate that Simbeeotic provides the appropriate level of fidelity to evaluate prototype systems while maintaining the ability to test at scale.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications; I.2.9 [**Artificial Intelligence**]: Robotics—*Autonomous Vehicles*

General Terms

Design, Experimentation, Measurement

Keywords

Swarm, Micro-Aerial Vehicle, Simulation, Testbed

1. INTRODUCTION

Simulation is often used in systems research for rapid prototyping, emulation of future architectures, and testing at scale. In this paper we present a simulator and hardware testbed that facilitate the development of micro-aerial vehicle (MAV) swarms.

MAV swarms are an emerging class of mobile sensing systems. As opposed to a single, more capable robot, MAV swarms employ a group of autonomous micro-robots to accomplish a common goal. Research platforms include quadrotors [15], fixed-wing aircraft [9], small "flying motes" [20], and insect-scale ornithopters [25]. Like sensor networks, MAV swarms rely on spatial diversity and collective sensing to explore a target area. However, MAVs must also be concerned with classic robotics challenges such as obstacle avoidance, navigation, planning, and environmental manipulation.

Our research is focused on MAV swarms comprised of thousands of smaller, less capable vehicles [21]. In this subset of the MAV swarm space the challenges faced by individual MAVs are similar to those of static sensor network nodes; computation is limited, sensing is minimal, and energy is scarce. However, there are differences between the two domains that invalidate some of the assumptions made in static sensor network deployments. For example, the radio is no longer the primary energy sink – it is dwarfed by the energy needed for actuation. Additionally, duty cycling the hardware to save energy is not an option when the vehicle is in flight. We contend that conducting research in this domain necessitates treating autonomous mobility as a first class concern in simulation tools and testbeds.

The main contribution of this work is a new simulation environment and MAV testbed. The core requirements for building a holistic MAV swarm simulator in the vein of other simulators [13] are defined as follows:

- **Scalability** The simulator must be able to simulate thousands of MAVs in a single scenario. Scale of deployment is an important aspect of swarm research. Without the ability to study algorithms at true swarm scale, some of the hard problems will be missed.

- **Completeness** Simulations should model as much of the problem domain as possible. Though research may be conducted on a subset of swarm design (e.g. flight control or networking), it is advantageous to construct a holistic view of the problem in which complex interactions are revealed. For MAV swarms, this means modeling *actuation*, *sensing*, and *communication* for each application.

- **Variable Fidelity** The desire to improve the accuracy of models is often at odds with simulation performance (scalability in this case). Users should be free to construct models with the appropriate level of fidelity to capture the subtleties of their problem. For example, researchers working on emergent algorithms may not require realistic flight control loops,

http://doi.acm.org/10.1145/nnnnnn.nnnnnn

whereas those working on controls will require accurate sensor and flight dynamics models but may not be concerned with network protocols. Using the same simulator, these researchers can work to improve the modeling of their domain while retaining the ability to combine their efforts and simulate the system as a whole.

- **Staged Deployment** No matter how detailed, simulation cannot completely capture every situation that will be encountered in the real world. While the ultimate goal is to deploy a swarm of MAVs, building hardware can be expensive and time consuming. The simulator can facilitate the development of control software and inform the hardware design process by providing a staged deployment feature, allowing prototype hardware to respond to both real and simulated inputs.

We present Simbeeotic, a simulation framework constructed from the above requirements. Simbeeotic supports both pure simulation and hardware-in-the-loop (HWIL) experimentation with a radio controlled (RC) helicopter testbed. The simulator relies on modular software design principles and a commitment to deployment-time configuration to provide modeling flexibility and ease of use. It is highly extensible and is designed for repeated experimentation. With Simbeeotic we demonstrate that whole-system modeling is feasible for the MAV swarm domain. The primary contribution of Simbeeotic is the tool itself, which is available to the community at `http://robobees.seas.harvard.edu`.

2. RELATED WORK

The MAV swarm domain intersects with other research areas, including biologically-inspired algorithms, robotics, and sensor networks. There are high fidelity simulators that exist in each of these communities. Prior to implementing Simbeeotic, we investigated the possibility of using these tools. In general, we were unable to find a simulator that satisfied our completeness requirement. We considered combining multiple simulators to satisfy this goal, but determined that performance would suffer due to the high fidelity of some of the tools. Each simulator uses considerable machine resources to model its own domain for thousands of agents, making our scalability goal untenable with this approach. Finally, we considered the engineering cost of repurposing multiple simulators to be too high, given that these tools are written in a number of languages and are not uniformly maintained.

Implementing a new simulator has several advantages. We can ensure that our requirements are satisfied and make design decisions that suit our needs. Our approach also allows us to evolve the fidelity of each subdomain (e.g. actuation, sensing, communication) as more accuracy is needed. However, we do not want to reinvent what is considered state-of-the-art in each domain. Whenever possible, we leverage open source tools and learn from existing models to avoid duplication of effort. In the rest of this section we discuss the relevant simulation tools from the swarm intelligence, robotics, and sensor networking communities.

The first set of tools considered come from the multi-agent systems and swarm intelligence communities. These simulators are appealing because they can generally model thousands of agents at once. Swarm [10], MASON [14] are two such tools. The main drawback of these simulators is that the they do not faithfully model the environment and actuation, opting for cell-based or 2D continuous worlds. In Swarm and MASON, a significant amount of effort would be put into modeling a three dimensional, physics-based world that is accurate enough to support the staged deployment requirement. MASON provides a builtin 3D space (known as a field in MASON-speak), but leaves manipulation of objects in the field (e.g. kinematics, collision detection) to the modeler. Breve [12] is very similar to Simbeeotic in that it is a discrete event simulator with an embedded physics engine. Unfortunately, models are written in a domain specific language called Steve (there is limited support for Python), which hinders adoption and limits the number of existing math and science packages available to modelers.

The robotics community has long used simulators as design tools since building hardware is often expensive and time consuming. In many cases, the hardware and software are co-designed, driving the need for accurate modeling of the physical environment. Thus, the strength of robot simulators is generally in modeling the interaction of the robot with the environment (e.g. actuation and sensing). Two commonly used tools are Webots [17] and Player-Stage [5] [23]. Webots models the environment as a three dimensional continuous space and has physics-based sensor models. It is an excellent teaching tool with support for many commercial robot platforms, but fails to meet our scalability requirements. In addition, its commercial nature does not allow for arbitrary modification, as would likely be the case for modeling communication networks and bridging with our testbed. Player-Stage consists of a robot driver interface, Player, and a simulated environment, Stage. Player is used in a client/server fashion to control robot and sensor hardware. Stage is used to simulate robots in a virtual environment, but exports a Player interface so that code can be migrated to a hardware platform. Stage is a 2.5D simulator that scales to handle hundreds of robots in real time for realistic workloads and thousands of robots for simple workloads. Its key limitation as mentioned by the authors [23] is that it is a first-order geometric simulator that does not model acceleration or momentum. Our approach to MAV swarm simulation requires a more comprehensive treatment of vehicle dynamics.

The Robot Operating System (ROS) is a collection of hardware drivers, algorithms, and tools for building robotic applications [22]. ROS users compose agent behaviors from a large set of open source packages that provide functionality for data acquisition and processing, planning, and locomotion. For the most part, ROS is a complementary technology to Simbeeotic. It is primarily used to construct a fully functioning software stack that can be deployed on one or more robots. There are packages that integrate ROS with simulators (including Player-Stage) to execute a robot in a virtual world, but these packages are insufficient for our needs due to shortcomings mentioned above. However, it should be possible to integrate ROS with our simulator in a similar way - a topic that is discussed further in Section 6.

The construction of the GRASP Micro UAV Testbed [16] is similar to Simbeeotic in that an offboard computer remotely controls the vehicles, relying on accurate position and orientation information from a motion capture system. One difference between the two testbeds is fidelity. The researchers using the GRASP testbed are interested in vehicle control, so the simulation includes a dynamics model and accounts for aerodynamic effects. Though we have performed a system identification on our helicopters and constructed a dynamics model, our efforts in simulation have focused on modeling larger swarms with lower fidelity vehicle movements. If researchers are interested in the aggregate behavior of a large swarm, foregoing the simulation of control loops can significantly improve simulation scalability.

The wireless networking and sensor network communities have invested heavily in simulation tools. GloMoSim [26] and ns-3 [19] are widely adopted simulators that model the OSI seven layer architecture. While they do an excellent job of implementing RF propagation, radio models, and network protocols, these tools are singu-

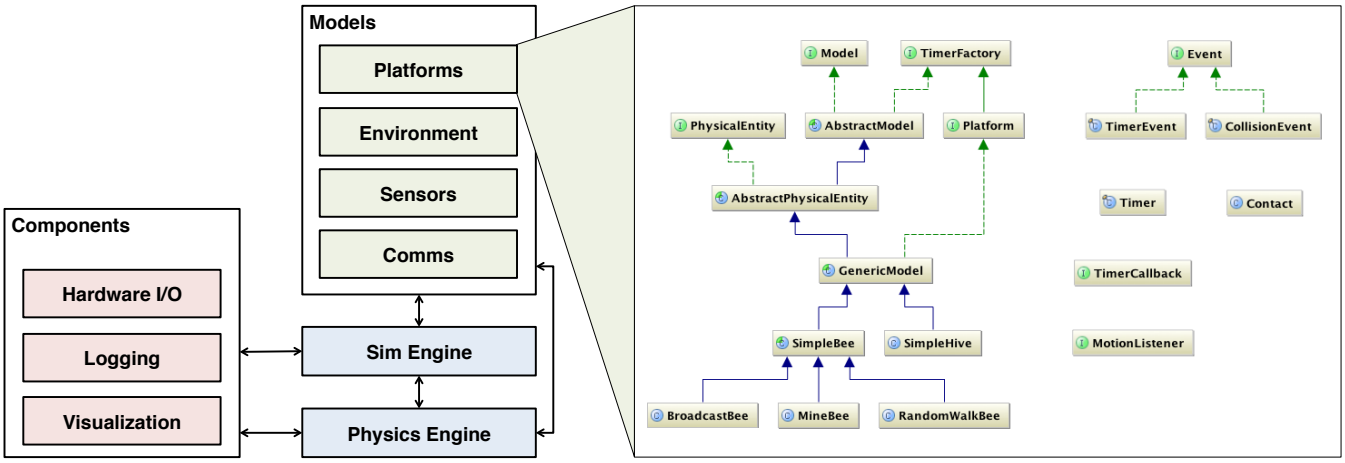


Figure 1: The Simbeeotic architecture. Domain models are plugged into a discrete event simulation engine. The kinematic state of models with physical presence is managed by an integrated physics engine. Several levels of abstraction in the model layer provide flexibility and convenience to modelers. The simulation architecture can be augmented by user-supplied plugin components.

larly focused on networking. A significant effort would be needed to model actuation and sensing to meet our completeness requirement. Rather, our approach is to start with a physical simulation and add networking fidelity as needed. This strategy allows us to selectively integrate the parts of these tools that are useful in our domain.

TOSSIM [13] and EmStar [6] are two popular wireless sensor network simulators. TOSSIM takes the completeness and bridging requirements to an extreme by providing a virtual environment in which the embedded mote software (running TinyOS) is executed. Though the whole-system approach is appealing, TOSSIM restricts users to writing applications in TinyOS. We borrow the idea of staged deployment from EmStar, which allows virtual models (e.g. radios) to be replaced by physical hardware in a testbed. Our staged deployment goal is derived from a desire to iterate on software and hardware designs using virtualized representations prior to building a deployable system. We do not consider EmStar a viable starting point for a MAV swarm simulator because the software is no longer maintained.

3. SIMULATOR DESIGN

At its core, Simbeeotic is a general purpose discrete event simulator. A simulation execution consists of one or more *models* that schedule *events* to occur at a future point in time. The virtual time of the simulation is moved forward by an executive that retrieves the next event from a queue of causally ordered pending events and passes it to the intended recipient for processing. In effect, time passes in between events – the events themselves represent discrete points in time. Since we are interested in modeling the MAV swarm domain, Simbeeotic builds upon the basic discrete event mechanism to provide convenient abstractions for building MAV swarm simulations, such as a virtual environment, robotic platforms, sensors, and radios.

Simbeeotic is written in the Java programming language. Java was chosen for a number of reasons. First, it is widely understood amongst our team and easily learned by neophytes. Second, it is for the most part a cross-platform language. We have confidence that Simbeeotic can be compiled on or distributed in binary form with little effort to the most popular (and some esoteric) operating systems. Third, there exists a large repository of high quality, open source libraries that can be leveraged by our modelers. At present, Simbeeotic consists of 13,387 lines of Java code in 148 classes and 506 lines of XML schema. Of this code base, 48% makes up the core (including the simulation executor, modeling interfaces, base classes, and common model implementations), 26% is for testbed integration, 13% is example code, 6% defines tools that generate random enclosed environments (such as mine shafts and office buildings), 6% is for visualization components, and 1% is for the main entry point. This codebase builds atop a collection of open source libraries that provide support for physics simulation, linear algebra, statistics, 3D visualization, plotting, message serialization, code generation, and logging.

3.1 Architecture

We have constructed Simbeeotic to fulfill the requirements established in Section 1 (scalability, completeness, variable fidelity, and staged deployment). In addition, we are careful to provide repeatability and promote ease of use and extensibility throughout our design. Scenario repeatability is of utmost importance; experiments must be reproducible given identical inputs. As such, our framework provides seeded random number streams to models and causally orders scheduled events using a set of deterministically generated tiebreak fields. Ease of use improvements include the elimination of boilerplate code through convenience mechanisms and a simple configuration system.

Figure 1 shows an overview of the simulation architecture (left-hand side) and a partial class diagram of the modeling abstractions (righthand side). The heart of the simulation is the simulation engine, which manages the discrete event queue and dispatches events to models, pushing virtual time forward. Prior to the start of the scenario, the simulation engine populates the virtual world from a supplied configuration and initializes all of the models by calling a model-specific initialization routine. The simulation engine is also responsible for answering queries about the model population. It provides an API for locating models based on type or ID.

The model layer sits on top of the simulation engine. The majority of user-supplied code will use model layer interfaces to implement features of the target domain. Simbeeotic employs a layered strategy to provide extension points within the model space. The layered approach API is one way that Simbeeotic fulfills the vari-

```
compass = getSensor("compass", Compass.class);

// a timer that takes a compass reading periodically
Timer compTimer = createTimer(new TimerCallback() {

    public void fire(SimTime time) {

        float h = compass.getHeading();
        ...
    }
}, 0, TimeUnit.SECONDS,
   sensorTimeout, TimeUnit.MILLISECONDS);
```

Figure 2: A code snippet from a model initialization routine demonstrating how to query for attached equipment and schedule a periodic timer (starting immediately and firing every `sensorTimeout` ms).

able fidelity design goal outlined in Section 1. Modelers introduce new functionality by building on layer with the interface that most closely matches the desired level of fidelity of the new model.

At the very bottom are the `Model` and `Event` interfaces. All models implement the `Model` interface, but few do so directly. The `AbstractModel` base class provides a default implementation that introduces other useful mechanisms, such as a seeded random number generator and a timer abstraction. We have committed to a continuous, three dimensional representation of space in Simbeeotic. The `PhysicalEntity` interface is defined to standardize the representation of a physical object (its size, shape, and mass), the information that can be queried about its kinematic state, and how its state can be manipulated (by applying forces, torques, and impulses). While it is possible for users to directly implement the `PhysicalEntity` interface, there exists a base class, `AbstractPhyscicalEntity`, that implements the interface by delegating to a rigid body physics engine (described below).

The next level of abstraction, the `GenericModel` class, treats the established physical body as a robotic platform, allowing equipment (e.g. sensors and radios) to be associated with the platform. The attached equipment models do not implement the `PhysicalEntity` interface. Rather, they are granted access to the host platform's physical presence and are attached using a body-relative position and orientation (e.g. antenna position and pointing direction). It is possible for a modeler to develop a new robotic platform by extending `GenericModel`, attaching sensors and radios, and defining custom agent logic using the timer mechanism. We introduce a final abstraction layer with the `SimpleBee` base class. This class provides a simple actuation API (e.g. `turn`, `setLinearVelocity`, `setHovering`) that makes the simulation more accessible to modelers who do not require high fidelity actuation modeling. The `SimpleBee` carries out the actuation commands with an internal kinematic update loop, translating the desired motion into the appropriate forces and torques and applying them to the body.

Modelers do not generally use the event scheduling mechanism directly. Rather, they implement agent logic using the `Timer` mechanism introduced by the `AbstractModel` class. Timers are a familiar abstraction that most modelers are comfortable using. Timers can be scheduled periodically or for single use. A custom callback is provided by the modeler, to be fired when the timer expires (Figure 2). Timers are implemented with a self-scheduled `TimerEvent` under the covers.

We also discourage the use of events for inter-model communication. We feel that in-domain communication mechanisms (e.g. the radios) should be used for the sake of realism and consistency. These mechanisms expose a familiar API to the modeler and are implemented internally with events.

In addition to building models in the target domain, users can extend the functionality of the simulator by providing *components*. Component implementations can interact with the simulation engine and physics engine directly, or with models by scheduling events. Two components that have received heavy use in our research are the 3D visualization component and a component used to communicate with our MAV testbed (discussed in Section 4). Component instances are created prior to model initialization and can operate in a separate thread of execution. This way it is possible to provide asynchronous I/O components, such as buffered loggers.

The final piece of the Simbeeotic architecture is the physics engine. As described above, the physics engine is used as the backing implementation for the `PhysicalEntity` interface, which is implemented by all models with a physical presence in the simulation. The physics engine we use is JBullet [11], a six degrees of freedom (6DoF) rigid body physics engine written in pure Java. JBullet provides a number of features that are useful in modeling MAV swarms at high fidelity:

- **Rigid Bodies** The MAV platforms and the virtual environment are composed from simple shapes (e.g. box, sphere, cone) and complex geometries (e.g. convex hull, triangular mesh).
- **Dynamics Modeling** The kinematic state of every object is maintained by integrating the forces and torques (e.g. rotor thrust, gravity, wind) applied to physical entities over time.
- **3D Continuous Collision Detection** Physical interactions between objects, such as environmental manipulation by a robot or bump sensors, are easily modeled.
- **Ray Tracing** Used primarily to implement sensors, such as range finders and optical flow.

When a descendant of `AbstractPhysicalEntity` is initialized, a representative rigid body is registered with the physics engine. The information associated with the body include its size, shape, mass, inertial properties, initial position, and orientation. As the rigid body is manipulated over time, its kinematic state is updated. During the course of an event, a model can query the kinematic state of an `AbstractPhysicalEntity`, which delegates the request to the rigid body. The simulation engine invokes JBullet in between events to push the dynamics simulation forward to the time of the next discrete event. We modified the JBullet library to break out of the dynamics simulation if a collision is detected during an update. In this case, the simulation engine checks a registry of interested collision listeners (registered by the models). If found, an event is generated to inform the listener (e.g. a bump sensor) of the collision. If no listener is interested in the collision, the dynamics simulation is resumed.

JBullet integration enables high fidelity actuation and sensor models, but this fidelity comes at a cost. Most of the routines in JBullet execute sequentially, therefore the performance of the simulator is explicitly coupled with the size of the swarm and complexity of the environment (i.e. the number of states that must be integrated and bodies checked for collisions). Section 5.1 evaluates the effect of environmental complexity and swarm size on simulation performance. Our conclusion is that the performance tradeoff is acceptable given the corresponding increase in fidelity.

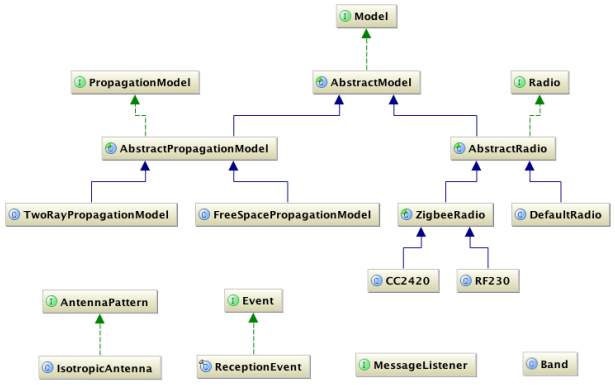


Figure 3: A class diagram for the RF communications package in Simbeeotic. The abstraction defines a physical layer packet-driven radio.

3.2 MAV Domain Models

Modelers contribute new functionality to the community codebase using the extension points described above. Simbeeotic constructs the virtual world from the rigid bodies defined by the physical entities and object definitions supplied in a world configuration file. The configuration file contains definitions of obstacles, structures, and environmental features to be inserted into the environment. Weather is modeled in the simulation by an abstract model (one without physical presence) that can be queried for the current weather state with respect to location. High fidelity models can simulate the effects of weather on themselves (e.g. by applying a wind force to a physical entity) or other models using the information provided (e.g. wind speed and direction).

Most of the builtin sensors provided by Simbeeotic are based on information provided by the physics engine. At present, interfaces and default implementations exist for inertial (accelerometer, gyroscope, optical flow), navigational (position, compass), and environmental (camera, range, bump) sensors. The inertial and navigational sensors use the kinematic state of the host platform, whereas the environmental sensors (and the optical flow sensor) use advanced features of the physics engine, such as ray tracing and collision detection. All of the default sensor models can be configured to produce inaccurate readings from truth state using a Gaussian noise model. Modelers can introduce new implementations of sensors that closely reflect the accuracy, precision, and error profile of real hardware.

Modeling RF communication is something that is done well by community standard simulators [19]. As such, the philosophy for RF in Simbeeotic has been to implement the smallest portion of the OSI seven layer architecture as possible and evolve the fidelity of the models (or integrate another simulator) when the need arises. Figure 3 shows a class diagram for the communications package in Simbeeotic. We implement a simple physical layer abstraction that includes the radio, antenna, and path loss model interfaces. Modelers are free to implement layers on top of the packet-driven radio abstraction.

3.3 Software Engineering Tricks

Simbeeotic relies on two features of the Java programming language, reflection and runtime annotation processing, to provide convenient interfaces to the end user. Though not necessary to achieve our original design goals, these features provide usability improvements over alternative implementations.

```
@Inject
private double maxVel;
private boolean useRadio = false;

@Inject(optional=true)
public void setUseRadio(@Named("use-radio")
                        boolean use) {
    this.useRadio = use;
}
```

Figure 4: A code snippet demonstrating the usage of the `@Inject` annotation for model parameterization.

Both reflection and runtime annotation processing are used to provide a flexible configuration system in Simbeeotic. Our design treats models and components as plugins to the simulator and configures them through dependency injection. Specifically, we use Java reflection to construct scenarios from an arbitrary number and type of models. We define an XML schema for our scenario configuration file that allows users to specify the fully qualified name of Java classes they wish to load and execute. When the scenario file is parsed, the user supplied type is checked for compliance (that it implements the required interfaces) and the specified number of instances are instantiated, registered with the simulation engine, and initialized. Other simulation frameworks, such as Player-Stage, allow for an arbitrary number of user defined scenarios to be loaded based on a configuration file. However, users are restricted to a pre-existing set of known model types. By using reflection, any class or component on the Java classpath is eligible for inclusion in the simulation.

The second part of configuration is parameterization. As a convenience to the user, we allow for a set of key-value pairs to be associated with each model or component definition in the scenario file. We use an open source dependency injection library, Google Guice [7] to configure the newly instantiated objects using the supplied parameters. After an object is instantiated, Guice inspects the instantiated class for injection sites (annotated fields or setters). To identify parameters for a model or component, users simply annotate their classes with the `@Inject` annotation, which can be attached to fields and methods. Guice uses the type of the field or method argument to match the injection site with a supplied configuration parameter. An additional `@Named` annotation that is used to disambiguate between parameters of the same type. Figure 4 depicts the usage of these annotations on fields and methods to prepare a model for parameter injection. With the ability to load arbitrary model and component implementations and inject parameters, many decisions regarding scenario construction can be pushed to deployment time.

Figures 11 and 12 of Appendix A list example Java model code and scenario configuration XML that leverage many of the features mentioned above. More comprehensive examples are available in the Simbeeotic source code.

4. HELICOPTER TESTBED

In addition to Simbeeotic, we maintain an indoor MAV testbed for conducting small-scale experiments. The testbed is primarily used to test algorithms in a more realistic environment. Despite our best effort, the simulator cannot form a complete representation of the real world. Our approach is to develop new systems and algorithms at scale in simulation and experiment with smaller deployments in the testbed.

We chose the E-flite Blade mCX2 [4] RC helicopter as the aerial platform for the testbed. The mCX2 is a low cost ($100), off-the-shelf vehicle. The mCX2 is quite limited in its capabilities; it has a payload of up to 5 grams and a flight time on the order of 5-10 minutes. It carries a proprietary control board that processes RC commands and stabilizes flight with an embedded gyroscope (yaw axis only). As a stock system it has no other processors, sensors, or radios. There are several advantages to using this platform. First, building a swarm from these helicopters is not prohibitively expensive. Second, the small size (20cm in length) allows multiple helicopters to be flown in our 7m x 6m laboratory space. The helicopter serves as a convincing prototype for the intended target of our research, insect-scale MAVs, in terms of flight time and capability. One disadvantage of the mCX2 is that it is a toy, not a research robot. Processing, sensing, and communication hardware must be added to make the vehicle into an autonomous swarm agent.

4.1 Remote Control

The helicopter testbed is instrumented with a Vicon [24] motion capture system. The Vicon sensors are capable of capturing the position and orientation of an object (adorned with reflective markers) in our testbed with sub-millimeter accuracy at 100Hz. This information is made available to programs that remotely control the helicopters. We achieve computer control by disassembling the supplied joystick and removing the radio transmitter daughterboard. Though the wireless protocol between the transmitter and helicopter is proprietary, the transmitter board is driven by a serial interface. The input signal to the transmitter is composed of four RC command values; yaw, pitch, roll, and throttle. We connect the transmitter to a PC with a USB-serial cable and allow the RC commands to be generated programmatically.

A testbed gateway machine mediates access to the observed helicopter state and RC transmitters. For helicopter state (measured by Vicon) the gateway provides a publish-subscribe mechanism for pushing updates to interested clients. Clients receive updates via messages that are serialized using Google Protocol Buffers [8]. The information in each update includes the Vicon frame number (essentially a timestamp) along with the object's identifier, position, orientation, and an occlusion flag (indicating that Vicon has lost track of the object in this frame). The gateway also provides a server for controlling each helicopter, which accepts `<yaw, pitch, roll, throttle>` command tuples. The server ensures that at most one client is connected to each helicopter and sends the latest RC commands to the transmitter at the required 50Hz. Clients communicate with the testbed gateway machine over a Gigabit Ethernet LAN.

4.2 Simbeeotic Integration

It is possible to write a standalone program that communicates with the testbed gateway to control the helicopters in the testbed. However, we realize that writing such programs would result in significant overlap with Simbeeotic, given that virtual sensor outputs would need to be constructed from the absolute position and orientation information provided by Vicon. We chose instead to integrate the helicopter testbed with Simbeeotic, allowing the modeler to leverage the virtual sensor implementations that already exist and conduct hybrid experiments with simulated and real MAVs. We refer to this operating mode as hardware-in-the-loop (HWIL) simulation. This technique is similar to the staged deployment mechanism in EmStar [6], which allows a simulated network to be transparently backed by real hardware.

We accomplish the testbed integration, depicted in Figure 5, by introducing *ghost models* in the simulator for physical objects that

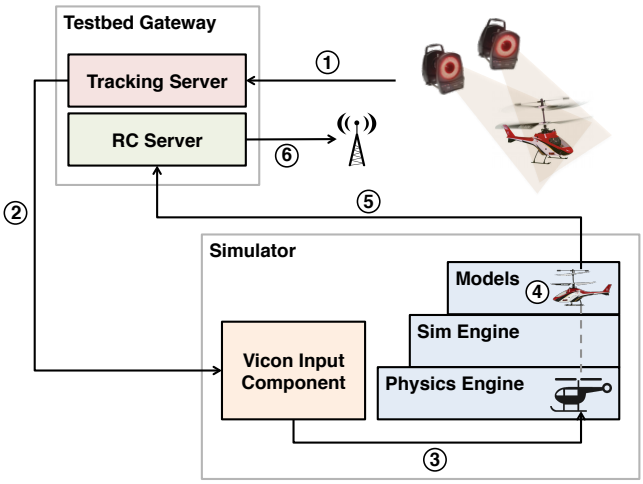


Figure 5: The HWIL cycle in Simbeeotic. Vicon cameras track the position and orientation of a helicopter and push frames to a tracking server (1), which pushes updates (2) to registered clients. A Vicon input component in Simbeeotic receives the update and overrides the kinematic state (3) of the corresponding object in the physics engine. When the ghost model executes an event (4) it has the most recent state of the helicopter. If a command is issued, it is sent to the RC command server (5) where it is dispatched by the RF transmitter (6) to the helicotper.

are tracked by Vicon. The ghost models implement the same `PhysicalEntity` interface as the simulated models, so interaction between the two is unchanged. The difference is that the ghost model's kinematic state is derived from the Vicon input, not the physics engine. However, the virtual sensors and other models that interrogate the virtual environment rely on the presence of an object in the physics engine for every physical entity. To fulfill this requirement, we simply create an object with the correct size, shape, and mass in the physics engine and periodically override its kinematic state with the information from Vicon. We introduce a new component that is responsible for connecting to the testbed gateway and receiving state updates. The simulation allows for the internal state of tracked objects in the physics engine to be updated prior to executing each event. Thus, whenever an event is executed, the state of all physical entities in the simulation is correctly represented by the physics engine. Some minor modifications to JBullet were required to allow the state to be set and to integrate the new state forward correctly in between Vicon updates.

Sending RC commands is similar. Upon initialization, each ghost MAV model opens a socket that connects to the testbed gateway. The RC commands are fed over the wire to the transmitter, which controls the helicopter in turn. The effects of the commands are witnessed by the Vicon, and the loop is closed.

Simbeeotic processes events as fast as possible when executing a pure simulation. However, the simulator must make an effort to run in realtime when hardware is attached. We make the assumption that the wallclock time necessary to execute an event is less than the virtual time between the current event and its immediate successor. If this assumption holds then it is trivial to maintain a soft realtime schedule by delaying the processing of an event until a corresponding system time has passed. When event processing violates this assumption, events are processed as fast as possible to catch up. This approach works in practice, though it compels modelers to keep events simple (arguably a good thing) and avoid scheduling simultaneous events.

4.3 HWIL Discussion

The testbed integration allows us to fly real vehicles using virtual sensors in a simulated environment. This arrangement allows us to transform our laboratory space into an arbitrarily complex proving ground, with virtual obstacles and features. One advantage of this setup is the ability to test the limits of proposed hardware and software payloads. The physics engine cannot always capture subtleties like aerodynamic ground effects and servo actuation error. The HWIL tests can aid in the iterative design process by observing these phenomena early on.

Our HWIL arrangement has some disadvantages as well. First and foremost, we are completely coupled to Vicon. Without a very accurate measurement of position and orientation, we would not be able to write sensors that convey the truth about the physical object. Second, we cannot fly outdoors. This is not a severe limitation at the moment, but our laboratory can only accommodate a handful of physical helicopters. Third, the control software for the helicopter is running in the simulator on a PC-class system. We run the risk of developing software that uses far too many resources for the eventual platform to handle. A TOSSIM-like approach to whole-system simulation may be needed to keep the modelers honest. Finally, our current setup does not allow for any processing or sensing to occur on the physical helicopter. This is why we refer to the remote-control HWIL solution as staged deployment. It is merely a stepping stone to truly autonomous MAVs. Section 6 discusses the possibility of extending the HWIL approach to communication hardware and how Simbeeotic can facilitate a move toward autonomous MAVs.

There are multiple sources of latency in the HWIL loop described above, including capture and processing time for Vicon frames, the transmission of MAV tracks to Simbeeotic over the LAN, processing events in Simbeeotic, sending RC commands to the testbed gateway over the LAN, and broadcasting the RC commands via the wireless link. If needed, the tracking server, RC server, and simulation could be co-located, eliminating the LAN. However, our experiments have shown that the round-trip loop latency in the testbed does not cause control instability or a substantive delay in MAV reaction time. We speculate that the latency introduced by the processing loop is absorbed by the relatively slow update rate of the RC helicopters (50Hz). In addition, if a command is delayed there is not a noticeable impact on the position and orientation of the MAV. Unlike the GRASP testbed, which focuses on fast, complicated maneuvers, our MAVs typically move at a rate of 0.5–1.0 $\frac{m}{s}$. At this velocity a 20ms latency might result in a positional drift of a few centimeters. Since the HWIL loop latency is not an observable hindrance to our experiments, little effort has been put into characterizing and minimizing the delay in our testbed.

When we first integrated the testbed MAVs into Simbeeotic we defined a common helicopter interface with the intention of creating HWIL-agnostic control behaviors. We defined two implementations of the interface, one that is purely simulated and one that interacts with the testbed hardware. The simulated helicopter implements control commands by applying a force (derived from a dynamics model) on its body in the physics engine. The HWIL helicopter implementation forwards the command to the testbed gateway as described above. However, our users never embraced the simulated helicopter model, preferring to try new behaviors at scale atop models based on `SimpleBee` and produce a separate behavior that interacts with the helicopter interface (for HWIL experiments) if needed. We conclude that the fidelity afforded by the simulated helicopter implementation is not required for the swarm experiments conducted by most members of our group.

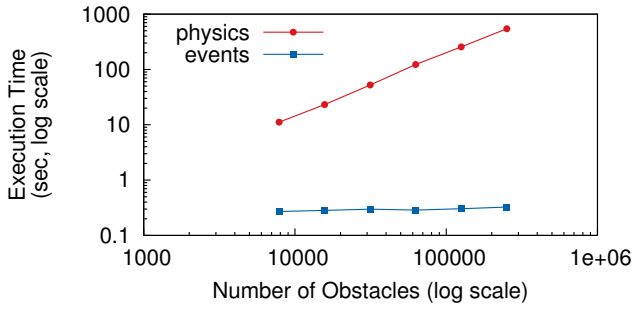


Figure 6: The overhead of collision detection in Simbeeotic. A fixed number of MAVs are simulated with a varying number of static obstacles. The amount of time to execute the event logic is constant. The number of required collision checks between MAVs and obstacles (and the time spent in the physics engine) grows linearly as obstacles are introduced.

5. EVALUATION

We have used Simbeeotic for over two years (one year with HWIL) to conduct research on MAV swarms. In this section we evaluate the performance of the simulator and present two applications that use Simbeeotic to explore the MAV swarm domain.

5.1 Simulation Performance

Since Simbeeotic is used in daily experimentation, we can state from experience that the tool meets our needs. However, it is beneficial to know the limits of the simulator, how modeler and user decisions can impact performance, and how the tool might be improved with future work. We evaluate the performance of the simulator and our ability to meet our scalability objectives based on three challenges:

- **Environment Complexity** The number of objects defined in the environment (e.g. obstacles, structures) determines how much collision checking is necessary during each physics update. Complicated scenarios can slow down the simulator.
- **Swarm Size** As more MAVs are introduced there is more work to be done by the physics engine to maintain the kinematic states of the moving objects. In addition, each new MAV represents an additional workload (events to process) to execute the agent's logic.
- **Model Complexity** Higher fidelity agent logic is likely to impact performance since complex events take longer to simulate.

Defining a single performance goal for the simulator is difficult given that modelers can construct scenarios that contain models of varying fidelity and execute in arbitrarily complex environments. Our motivation for constructing the simulator was to study large swarms of less capable MAVs. Thus, we focus on a performance goal of simulating one thousand MAVs executing a typical workload in soft realtime or better[1]. Our experiments show that Simbeeotic is capable of simulating thousands of MAVs executing a typical workload and hundreds of MAVs executing a complex workload in

[1]The scalability goal is lowered for HWIL scenarios to ensure that RC commands are issued as close as possible to a realtime schedule (i.e. the helicopters do not crash).

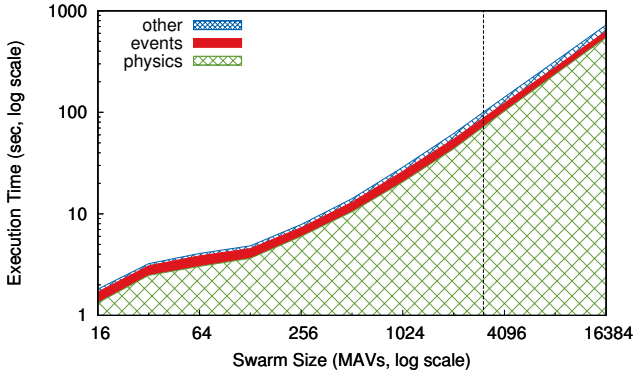


Figure 7: The number of events to process and kinematic states to integrate increases linearly with swarm size. The corresponding event and physics execution times reflect this increase. The dashed vertical line indicates the point above which soft realtime cannot be achieved with this workload (3,074 MAVs).

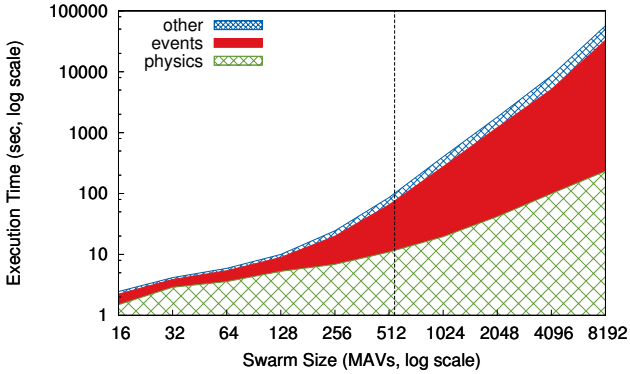


Figure 8: The simulation runtime does not increase linearly for the broadcast scenario. A nontrivial amount of work is undertaken for each radio transmission event, which may also generate reception events on all other MAVs. The event execution time dominates this scenario as the swarm scales. The dashed vertical line indicates the point above which soft realtime cannot be achieved with this workload (550 MAVs).

soft realtime. As with other discrete event simulators, Simbeeotic is capable of simulating faster than realtime when there is no testbed hardware in the loop.

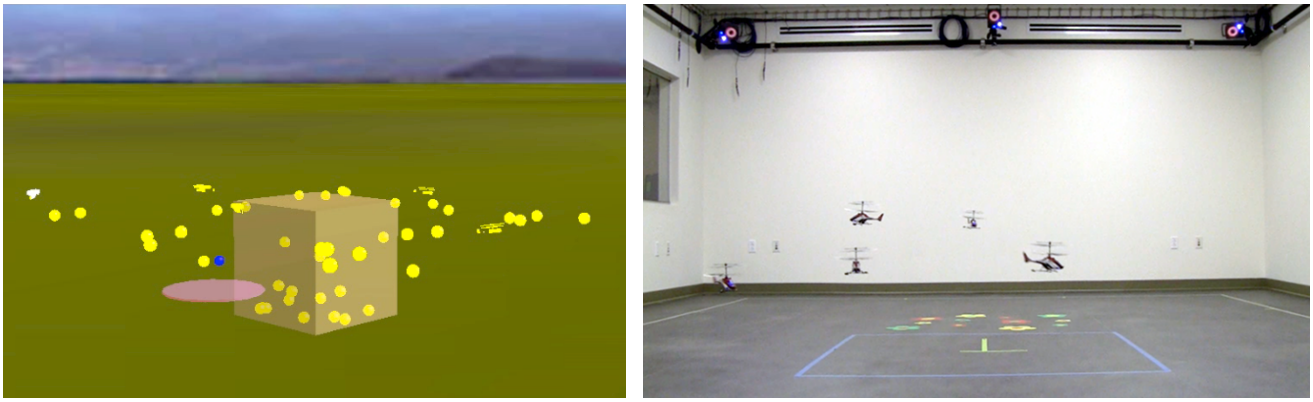
We define a typical MAV workload to consist of a random walk (10Hz kinematic update rate) and a periodic sensor reading (1Hz compass). In all of the following experiments the MAVs operate for 100 virtual seconds and start from random locations within 20m of the origin. We instrument Simbeeotic to record the amount of wall-clock time necessary to simulate the physics (in between events) and run the agent logic (the events themselves). All measurements are taken on a 2.2GHz quad-core laptop with 8GB of RAM using the HotSpot JVM version 1.6.0_26.

We begin with an experiment that addresses the environmental complexity challenge. We measure the overhead of collision detection by simulating a small swarm (32 MAVs) executing the workload defined above. A variable number of static obstacles are introduced into the environment at each iteration of the experiment. As the number of obstacles grows, we expect the collision detection routines to take more time. Performing naive collision detection is $O(n^2)$ in time. Fortunately, JBullet employs more sophisticated collision detection routines that reduce the number of compared objects. Since the kinematic state of a static object in JBullet is not integrated forward at each time step, we can attribute any increase in the physics simulation time to increased collision checks (and likely some added overhead). Further, we expect this increase to be linear with respect to the number of obstacles (as opposed to quadratic) because two statically placed objects are not checked for collisions. Thus, the only collisions being checked are between the MAVs and the obstacles. The results in Figure 6 show that the amount of time to execute the events (agent logic) is constant through the course of the experiment (the swarm size does not change). However, the overall time spent in the physics simulator increases linearly with the number of objects introduced. MAV swarm modelers must be informed that environmental complexity, not just swarm size, can have a significant impact on the performance of the simulation.

The next experiment aims to characterize the scalability of the simulator with respect to swarm size. With each iteration of this experiment we vary the number of MAVs deployed into a constant environment (no obstacles). The MAVs execute the typical workload defined above. We expect increased collision checks (between MAVs) and a linear increase in the time needed to update the kinematic states of the MAVs. Figure 7 shows the results of this experiment. The simulation scales roughly linearly as the swarm size is increased. The number of events (and the corresponding event execution time) scales linearly as well. Using this workload, it is possible to simulate 3,074 MAVs in soft realtime. These scalability results are comparable to the performance of Player-Stage using a similarly defined "simple" workload [23].

We address the final performance challenge, model complexity, by introducing an additional element to the workload – each MAV broadcasts a radio message at 1Hz. The result of this addition is a significant increase in the event execution time. The increase in event time has two main causes, event complexity and message explosion. The former refers the nontrivial amount of work that must be done to send each packet. The propagation model considers every other radio-equipped model as a potential recipient and performs path loss calculations between the two radios. This includes determining the antenna positions and orientations, extracting the gains from the antenna patterns, and computing the signal strength at the recipient. Though there is a cutoff distance in the path loss model, this optimization is not useful in the scenario under test because the MAVs are closely spaced. Message explosion refers to the number of receive events that will be generated as a result of each packet transmitted. It is possible that n^2 events are generated each second in the simulation. In this case, some events are not generated due to low signal strength at the recipient. Despite the relative simplicity of the receive event processing, the sheer number that need to be processed can add significant overhead. The results of this experiment are shown in figure 8. The overhead of creating and enqueuing these events is likely the source of the increase in the 'other' category. With this workload, we can simulate 550 MAVs in soft realtime.

We set out to create a complete simulator for the MAV swarm domain. These experiments demonstrate that Simbeeotic meets our scalability goals for typical workloads (thousands of MAVs in soft realtime). They also reaffirm the premise that environment complexity and model fidelity can significantly impact performance. Section 6 discusses potential modifications to improve the scalability of the simulator.



(a) Virtual World (b) Helicopter Testbed

Figure 9: A HWIL deployment of a MAV swarm. Five testbed MAVs are deployed alongside 45 simulated MAVs to search a space for flowers. The circle in the virtual world represents a flower patch (also visible in the testbed floor), and the box at the center denotes the MAV hive.

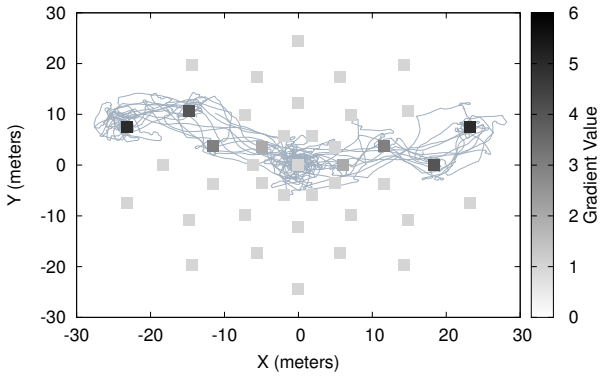


Figure 10: An overhead trace of five simulated MAVs navigating through the environment with the assistance of a gradient field provided by RF beacons (square dots). The gradient in this case specifies two paths away from the center. The MAVs use the value and the signal strength of beacon packets as input to a biased random walk (chemotaxis) algorithm. The MAVs are successful in traveling between the hive and the edge of the gradient field along the two paths.

5.2 Example Scenarios

We describe two MAV swarm scenarios that we have simulated using Simbeeotic. The main goal of the first scenario is coverage. The MAV swarm is deployed to search a space for features of interest (e.g. flowers) and manipulate the environment where the features are located (e.g. chemical sampling, pollination). There are many possible solutions to the swarm coordination problem, including static task assignment, cooperative planning, and emergent behavior. We employ a system that coordinates the actions of the swarm from a centralized location called the hive [2]. We discretize the world into cells and dispatch MAVs from the hive to perform a specific task until they are low on energy, at which point they return to recharge. A planner at the hive analyzes the results of the trip (the information collected) and determines which cells require more attention. Figure 9 shows a snapshot of our swarm management system executing a search and survey scenario using 45 virtual MAVs and 5 testbed helicopters. The lefthand panel shows a Simbeeotic visualization of the virtual world, while the righthand panel shows the helicopters flying under PC control. This example demonstrates that Simbeeotic has adequate modeling fidelity in actuation and sensing to fly real hardware, and that the staged deployment goals are satisfied.

The second scenario explores the possibility of using RF beacons embedded in the environment as navigational aids for flying MAVs. Figure 10 shows an overhead trace of MAVs using a biased random walk algorithm in a gradient field [3] to navigate along two preferred paths. The MAVs and beacons are equipped with virtual CC2420 radios and isotropic antennas. The two-ray RF propagation model is used to calculate path loss. The MAVs use the value and signal strength of beacon packets to determine the direction of travel in the gradient. This example demonstrates one way that RF communication can be used in a MAV swarm.

6. ONGOING WORK

There are three main directions for future work – scalability, fidelity, and autonomy. From the results in Section 5.1 it is clear that the the physics engine is a bottleneck. We rely heavily on JBullet for modeling actuation (dynamics, collision detection) and sensing (ray tracing). Though it has satisfied our needs thus far, we may consider replacing JBullet with Bullet [1] as we move toward modeling swarms with tens of thousands of MAVs. JBullet is a pure Java port of Bullet, which is written in C++. In addition to being written in a native language, newer versions of Bullet support hardware acceleration on the GPU. The potential performance improvement may be worth the modest engineering effort to create Java wrappers for the subset of the Bullet interfaces used by Simbeeotic.

Though we model the breadth of the MAV swarm domain, the fidelity of the networking models in Simbeeotic could be improved. To date, our work on MAV swarms has not focused on communication. It is likely that the networking interfaces will need to evolve beyond the simple physical layer implementation. We will look to leverage community standard tools and models such as ns-3 as our needs develop. In addition, we may expand our HWIL capabilities to include real radios in a mote testbed, much link in EmStar. On first inspection, it appears that the ghost model approach will work well with a radio interface. Packets sent on a ghost interface would be transmitted on the physical radio in addition to the virtual radio, and packets received on the physical radio would be captured and

injected as a virtual packet reception. Some care must be taken to prevent duplicate transmission and reception events by ghost radio models participating in both domains.

A major limitation of our testbed is the dependence on an accurate motion capture system. We expect to phase out this dependence as our MAV platform evolves. For example, our MAVs may soon be equipped with enough sensors to stabilize themselves and perform local obstacle avoidance but lack the computing power for path planning. In this case a Simbeeotic model may simulate the higher levels of the MAVs software stack (e.g. planning, coordination) while the lower levels (e.g. control, obstacle avoidance) are executed onboard. With such a deployment we can no longer rely on a purely simulated environment. Obstacles must be represented in both worlds, a feat that can be accomplished by adding Vicon markers to physical objects and creating a corresponding ghost representation. Ideally, we would want to feed virtual sensor information wirelessly to the testbed MAVs, allowing them to sense purely virtual objects. However, this might not be feasible considering the bandwidth and latency requirements of the sensors.

As we develop the software stack that will execute on the autonomous MAVs, it may be possible to leverage ROS [22]. This presents an opportunity for Simbeeotic to be used as a virtual input to software that will be embedded on a vehicle. We view this TOSSIM-like approach as another (purely simulated) intermediate step toward MAV autonomy that is orthogonal to HWIL operation.

On our path toward fully autonomous MAVs we may relax the requirement that physical objects and virtual objects are co-visible. Instead, we could construct a virtual world to match the physical world and ignore interactions between MAVs. This would allow us to experiment outside of the testbed and obviate the need for accurate tracking once the MAVs are fully autonomous (other than for ground truth during experimentation). We have considered constructing a less accurate tracking system using Microsoft Kinect [18] sensors that can be deployed outside of our testbed. This system would be used for collecting ground truth positional information of indoor exploration experiments. Simbeeotic would remain as a useful tool, allowing physical MAVs to coordinate with simulated MAVs via HWIL in the communication layer, as described above.

7. CONCLUSIONS

MAV swarms are an emerging class of mobile sensor systems with strong ties to the robotics, sensor networking, and swarm intelligence communities. We present Simbeeotic, a simulation environment and testbed for MAV swarms to support research effort in this area. Simbeeotic is designed to be flexible and easy to use. The domain modeling interfaces are designed to cover a complete view of the application space, including actuation, sensing, and communication. We show that Simbeeotic is capable of simulating MAV swarms at scale, and demonstrate its usefulness in exploring new concepts with real hardware. Simbeeotic is available as open source at `http://robobees.seas.harvard.edu`.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insight and detailed feedback. Special thanks to Eddie Kohler for his encouragement and editorial contributions. This work was partially supported by the National Science Foundation (award number CCF-0926148). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] Bullet Physics Library. `http://bulletphysics.org/wordpress`.

[2] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 121–134, New York, NY, USA, 2011. ACM.

[3] A. Dhariwal, G. Sukhatme, and A. Requicha. Bacterium-inspired robots for environmental monitoring. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA '04)*, volume 2, pages 1436–1443, May 2004.

[4] E-flite Blade MCX2. `http://www.e-fliterc.com`.

[5] B. Gerkey and R. Vaughan. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, 2003.

[6] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, 2004.

[7] Google Guice. `http://code.google.com/p/google-guice`.

[8] Google Protocol Buffers. `http://code.google.com/apis/protocolbuffers`.

[9] S. Hauert, L. Winkler, J. Zufferey, and D. Floreano. Ant-based swarming with positionless micro air vehicles for communication relay. *Swarm Intelligence*, 2(2-4):167–188, 2008.

[10] D. Hiebeler. The Swarm simulation system and individual-based modeling. In *Proceedings of Decision Support 2001: Advanced Technologies for Natural Resource Management*, Toronto, Sept. 1994.

[11] JBullet. `http://jbullet.advel.cz`.

[12] J. Klein. 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*, 2002.

[13] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys '03)*, Nov. 2003.

[14] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. MASON: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, 2004.

[15] N. Michael, J. Fink, and V. Kumar. Cooperative manipulation and transportation with aerial robots. *Autonomous Robots*, 30(1):73–86, Sept. 2010.

[16] N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar. The GRASP Multiple Micro-UAV Testbed. *Robotics & Automation Magazine, IEEE*, 17(3):56–65, 2010.

[17] O. Michel. WebotsTM: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):40–43, 2004.

[18] Microsoft Kinect. `http://www.xbox.com/kinect`.

[19] ns-3. `http://www.nsnam.org`.

[20] A. Purohit, Z. Sun, M. Salas, and P. Zhang. SensorFly: Controlled-mobile sensing platform for indoor emergency response applications. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN '11)*, Apr. 2011.

[21] RoboBees. `http://robobees.seas.harvard.edu`.

[22] Robot Operating System. `http://www.ros.org`.

[23] R. T. Vaughan. Massively multi-robot simulations in stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.

[24] Vicon Motion Capture Sytems. `http://www.vicon.com`.

[25] R. Wood. The first takeoff of a biologically inspired at-scale robotic insect. *IEEE Transactions on Robotics*, 24(2):341–347, 2008.

[26] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the twelfth workshop on Parallel and distributed simulation (PADS '98)*, pages 154–161. ACM, July 1998.

APPENDIX

A. EXAMPLE MODEL AND CONFIG

```
public class InstrumentBee extends SimpleBee
                        implements MessageListener {

private Compass compass;
private Radio radio;

@Inject(optional=true)
@Named("max-vel")
private float maxVelocity = 2.0f;     // m/s
private float velocitySig = 0.2f;     // m/s
private float headingSig = 0.2f;      // rad
private long sensorTimeout = 1000;    // ms
private long radioTimeout = 1000;     // ms

@Override
public void initialize() {

  super.initialize();
  setHovering(true);

  compass = getSensor("compass", Compass.class);
  radio = getRadio();

  createTimer(new TimerCallback() {

    public void fire(SimTime time) {
      compass.getHeading();
    }
  }, 0, TimeUnit.SECONDS,
     sensorTimeout, TimeUnit.MILLISECONDS);

  createTimer(new TimerCallback() {

    public void fire(SimTime time) {
      radio.transmit(new byte[] {1, 2, 3, 4});
    }
  }, 0, TimeUnit.SECONDS,
     radioTimeout, TimeUnit.MILLISECONDS);
}

@Override
public void finish() {
}
```

```
@Override
protected void updateKinematics(SimTime time) {

  // randomly vary the heading (rot. about Z axis)
  turn(getRandom().nextGaussian() * headingSig);

  // randomly vary the velocity in X & Z dirs
  Vector3f newVel = getDesiredLinearVelocity();

  newVel.add(new Vector3f(getRandom().nextGaussian() *
                            velocitySig,
                          0,
                          getRandom().nextGaussian() *
                            velocitySig));

  // cap the velocity
  if (newVel.length() > maxVelocity) {

    newVel.normalize();
    newVel.scale(maxVelocity);
  }

  setDesiredLinearVelocity(newVel);
}

@Override
public void messageReceived(SimTime time, byte[] data,
                            double rxPower) {
  // do nothing
}

@Inject(optional=true)
public final void setVelSig(@Named("vel-sigma")
                                final float sigma) {
  this.velocitySig = sigma;
}

@Inject(optional=true)
public final void setHeadSig(@Named("heading-sigma")
                             final float sigma) {
  this.headingSigma = sigma;
}

@Inject(optional=true)
public final void setSensorTO(@Named("sensor-timeout")
                              final long t) {
  this.sensorTimeout = t;
}

@Inject(optional=true)
public final void setRadioTO(@Named("radio-timeout")
                             final long t) {
  this.radioTimeout = t;
}
}
```

Figure 11: A listing of the Java code for the MAV used in the experiment from Figure 8. The MAV is based on the `SimpleBee`, which provides a simplified locomotion interface (used in `updateKinematics` to implement a random walk). Timers are established at initialization to take a sensor reading and send a message periodically. Both long and short forms of parameter injection (method and field annotation) are demonstrated. This code demonstrates the basic Simbeeotic APIs but serves no useful purpose other than to instrument the simulator in our experiments.

```
<?xml version="1.0"?>

<scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://harvard/robobees/simbeeotic/configuration/scenario">

    <master-seed>
        <constant value="111982"/>
    </master-seed>

    <simulation>
        <end-time>100.0</end-time>
    </simulation>

    <models>
        <model>
            <java-class>harvard.robobees.simbeeotic.model.comms.FreeSpacePropagationModel</java-class>
            <properties>
                <prop name="noise-floor-mean" value="-100"/>
                <prop name="noise-floor-sigma" value="10"/>
                <prop name="range-thresh" value="30"/>
            </properties>
        </model>
        <model>
            <java-class>harvard.robobees.simbeeotic.model.SimpleHive</java-class>
            <start-position x="0" y="0" z="0"/>
        </model>
        <model count="8192">
            <java-class>harvard.robobees.simbeeotic.example.InstrumentBee</java-class>
            <properties>
                <prop name="kinematic-update-rate" value="100"/>
                <prop name="use-random-start" value="true"/>
                <prop name="random-start-bound" value="20"/>
                <prop name="allow-bee-collisions" value="true"/>
                <prop name="radio-timeout" value="1000"/>
                <prop name="sensor-timeout" value="1000"/>
            </properties>
            <sensor name="compass">
                <java-class>harvard.robobees.simbeeotic.model.sensor.DefaultCompass</java-class>
            </sensor>
            <radio>
                <java-class>harvard.robobees.simbeeotic.model.comms.CC2420</java-class>
                <properties>
                    <prop name="tx-power-level" value="31"/>
                </properties>
                <!-- by default an isotropic antenna will be attached -->
            </radio>
            <start-position x="0" y="0" z="0"/>
        </model>
    </models>

    <!-- if you want to see what is happening, uncomment this component
    <components>
        <variation>
         <java-class>harvard.robobees.simbeeotic.component.VisComponent3D</java-class>
         </variation>
    </components>
    -->
</scenario>
```

Figure 12: The scenario configuration file used in the experiment from Figure 8. Users can control the global simulation properties (e.g. master random seed, simulation end time), and add models and components. The structure of a model entry (which must conform to the scenario XML schema) allows users to specify the type of the model to instantiate along with the number of instances. In addition, equipment such as sensors and a radio can be attached. The key-value properties are passed to Guice for injection into user classes that contain `@Inject` annotations. A description of the virtual world in which these models operate is supplied in a separate configuration file.