Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

INTRODUCING THE SIMULATION PLUGIN INTERFACE AND THE EAS FRAMEWORK WITH COMPARISON TO TWO STATE-OF-THE-ART AGENT SIMULATION FRAMEWORKS

Lukas König
Daniel Pathmaperuma
Felix Vogel
Hartmut Schmeck

Karlsruhe Institute of Technology (KIT)
Institute for Applied Informatics and Formal Description Methods (AIFB)
D-76133 Karlsruhe, GERMANY

ABSTRACT

This paper proposes a novel architectural concept for developing agent-based simulations called *Simulation Plugin Interface (SPI)*; furthermore, a simulation framework called *Easy Agent Simulation (EAS)* based on the proposed architecture is presented. The SPI introduces an intermediate layer between the simulation engine and the simulation model. It contains all types of functionality which are required for a simulation but are logically separable from the simulation model. This includes visualization, probes, statistics calculations, logging, scheduling, API to other programming languages, etc. The architecture is particularly suitable to guide student programmers with low experience to well-structured and reusable simulation components. The SPI architecture is not bound to the EAS Framework, but can be implemented as an extension to most state-of-the-art simulation frameworks. In a comparative study, the EAS framework is compared to the agent simulation frameworks NetLogo and MASON, using the well-known "Stupid Model" as a test scenario.

1 INTRODUCTION

The growth of the field of *agent-based simulation (ABS)* since the early 1990s opened a new chapter to academic research in many domains. It has been established in computer science, natural science, social science, education, networking, artificial intelligence, urban simulation, and many more. (Siebers and Aickelin 2008) In the last two decades various ABS platforms have been developed, more or less loosely connected to the domains they have been developed for; this includes (but is not limited to) MAML for social sciences, NetLogo, StarLogo and recently ReLogo for education, Jade, NeSSi and OMNET++ for networking, Breve and Repast for artificial intelligence, MatLab, and OpenStarLogo for natural science, and Aimsun and OBEUS for urban / traffic simulation. There also exist many general-purpose ABS frameworks of different powerfulness and scope, such as Anylogic, Ascape, DeX, EcoLab, Madkit, Magsy, Mason, NetLogo, ReLogo, Repast, Swarm, SOAR, etc. These lists are far from being complete; a more comprehensive list is given by (Nikolai and Madey 2009).

One issue particularly important for rather big research projects is the development of structured and reusable code. Structured code can be the foundation of an error-free implementation, especially if unskilled programmers are involved. This is, for example, frequently the case in academic research where students are involved within the scope of a thesis or a laboratory course, but also in other projects including people without a software engineering background. While there exist popular easy-to-learn ABS toolkits like MASON or NetLogo, their architectures do not guide unskilled users actively to a well-structured code, but leave it to their own competency. This can lead to code that is hard to reuse, and understand, and, in turn, to potentially erroneous implementations.

In this paper we present an architectural concept for developing ABS which is called *Simulation Plugin Interface (SPI)*. It introduces a layer between the simulation core and the simulation model which is used to hold all the implementations necessary for a simulation run, but is not directly connected to it. This includes visualization, probes, scheduling, and many more. The SPI allows for modularized programming, modules being flexibly attachable to and detachable from the simulation. Furthermore, the modules can be attached to different simulation scenarios making code reusable. The SPI is inspired by the *Model / View / Controller (MVC)* architectural pattern (Burbeck 1992) and can be seen as a generalization of that pattern.

Furthermore, an ABS framework called *Easy Agent Simulation (EAS)* implementing the SPI is presented. EAS is programmed in Java and can be used as a general-purpose simulation framework. It is a rather new simulation platform being developed for about 2 years. It was started out of the need for an ABS framework that could be used in projects where academic research is combined with student work.

To compare EAS and the SPI to state-of-the-art simulation platforms in terms of implementation time and code structure, we perform a comparative study where students, unexperienced with ABS, implemented Railsback's "Stupid Model" (Railsback, Lytinen, and Grimm 2005; Railsback, Lytinen, and Jackson 2006) in EAS, MASON (Luke, Cioffi-Revilla, Panait, Sullivan, and Balan 2005) and NetLogo (Sklar 2007), respectively. Our results indicate that the code implemented in EAS is better structured than that in MASON and NetLogo. Furthermore, the SPI has been specifically mentioned by the students as beneficial in comparison to the other platforms.

Our study is performed along with the study by Railsback (2006), although omitting a runtime investigation which is not within the scope of this study. Additionally, we perform a code analysis similar to the study performed with EcoLab (Standish 2006). However, the results are not directly comparable, as a different code counting method has been used. Standish used the Linux command `wc` which counts any lines in the files. We found it more accurate to use the program `cloc` (http://cloc.sourceforge.net/) which counts lines of real code only, omitting comments and blank lines. For example, Standish reports 816 lines of code for his implementations in MASON while our students needed on average 631.8.

The remainder of this paper is organized as follows: in the next section, the SPI is presented from an architectural point of view; Sec. 3 describes the implementation of the SPI in the EAS framework; in Sec. 4, the comparative study is described and the results are discussed; a conclusion and an outlook to future research is given in Sec. 5.

2 ARCHITECTURAL CONCEPT

The SPI architecture and its implementation in the EAS framework arose from a background of academic evolutionary swarm robotics research, cf. König, Mostaghim, and Schmeck (2009), and the need for a simulation program which simplifies the combining of simulations originating from research studies with students' results from lab courses or theses studies. The EAS framework was created in Java as a general-purpose simulation program that is not limited to ABS. However, as the name indicates, a special emphasis of EAS is ABS (discrete stepwise or discrete event based), and practically all implementations so far are agent-based. Therefore, the architecture and the implementation in this and the following section are described at a universal level, but agent-based scenarios are chosen as examples and for the evaluation.

2.1 Preliminaries

To provide clearness about basic terms used throughout the paper, some definitions are given in this section. Some of the terms are adjusted to the proposed architecture and may be defined differently than in the literature. For example, a distinction is made between a time instant, a tick and an event as these notions are explicitly distinguished within the EAS framework.

Notification: An object that is passed from one part of a program to another providing information about state changes. Notifications are used to propagate the progression of time and the occurrence of events.

Time instant: A point in continuous time space $\mathbb{R}$ represented by a positive double value.

Tick: A discrete time instant in $\mathbb{N}_0 \subseteq \mathbb{R}$ represented by a positive integer value (subset of time instants).

Event: An object of a specific type (namely `EASEvent` in EAS) that can be broadcast at any time during simulation runtime. Events may be time-less or bound to a time instant; in that, they can be seen as a superset of time instants.

Simulation core: The basic skeleton of the simulation program, minimally required to run a simulation (may contain several classes / packages).

Simulation engine: Part of the simulation core which provides a notion of time and creates notifications of events and the progression of time (class `SimulationTime` in EAS).

Simulation model: Implementation of the scenario that is simulated including rules of state changes due to progression in time / events. A simulation model typically consists of an environment and several agents acting in it. A simulation model may contain several classes / packages. In EAS, there exists (at least) one distinguished entrance class called *main runnable* (of type `EASRunnable`) that receives notifications and passes them on to other classes of the simulation model, if required.

Scheduling: Generally the (potentially manifold) assignment of objects from some arbitrary finite set $S = \{O_1, \ldots, O_n\}$ to points in time, i. e., time instants. Can be expressed as a mapping $A : S \to \mathbb{R}$. In our context, S is the set of invokable methods within the simulation model. A *scheduler* is a program (part) that chooses a mapping from A and performs the according method invocations at the appropriate time instants based on notifications from the simulation engine. A simulation program is called *completely schedulable* if it allows for every mapping $a \in A$ the implementation of a scheduler that chooses the mapping a.

2.2 Classification of the Architecture

The concept of the SPI introduces an intermediate layer called *Scheduler / Plugin Layer (SPL)* between the simulation engine and the simulation model. The SPL can be used to place all types of functionality inside that is logically separable from the simulation model, but still influences or observes the simulation at runtime. The architectural choice of introducing an SPL originates from the following observations made for a variety of ABS frameworks:

1. There are several common types of simulation-related functionality that influence the simulation at runtime, but should be separated from the simulation model:
   (a) Scientific simulations usually use a concept of "probes" (different types of views on the simulation data) and "controllers" (user interaction tools, sometimes called probes, too) which are separated from the simulation model and observe and influence the simulation at runtime (Railsback, Lytinen, and Jackson 2006).
   (b) In addition to probes, sometimes more sophisticated functionality is logically separable from the simulation model; for example, in a scientific experiment a scenario generator object might be useful that restarts a setting several times with different preconditions.
   (c) In some situations it can even be desired to separate functionality from the simulation model which logically belongs to it; for example, in an agent learning scenario one might want to implement the learning functionality as a module which can be activated or unplugged to run the simulation in a non-learning mode.
2. Scheduling is a functionality that influences the simulation at runtime and is usually separated from the simulation model. In that sense scheduling can be seen as a special case of the functionality listed in (1).
3. An API for runtime control of simulations from "outside" can also be seen as such functionality. For example, the microscopic traffic simulation AIMSUN provides an API by informing the outside program of every step the simulation performs and providing methods for the observation and manipulation of the simulation state (Barcelo, Codina, Casas, Ferrer, and Garcia 2005; Casas, Ferrer, Garcia, Perarnau, and Torday 2010).

4. Many of the above-mentioned functionality might be useful in different scenarios, for example, a learning module could be used in several types of environments containing different types of agents.

Typically, the above listed types of functionality are treated separately from each other. As an example, probes are usually implemented as a special concept limited to observation and user interaction. For example, in MASON probes (called "inspectors" there) are GUI panels which allow the user to inspect or modify object parameters using reflections and Java Beans (Luke, Cioffi-Revilla, Panait, Sullivan, and Balan 2005). Swarm provides a more powerful probe concept using the reflection capabilities of Objective-C, to allow any object's state to be read or set and any method to be called in a generic fashion (Minar 1996; Daniels 2000). NetLogo provides a rather static probe capability by a set of GUI items that can observe and manipulate variables of predefined types (Sklar 2007). However, in all cases probes are solely meant to cover a subset of the functionality in (1) and none of that in (2) and (3). The latter are treated in individual ways by each simulation.

In EAS all functionality from (1), (2) and (3) is treated in the same way by using the SPL. There, the concept used in AIMSUN is generalized to provide an interface for a large set of functionalities. The basic idea is to let the simulation engine pass notifications not directly to the simulation model, but only to the SPL. In the SPL, a list of "plugins" or "schedulers" (the terms are actually equivalent from the architectural point of view, and only "plugin" will be used in the following) decide what to do when they receive a notification. They can pass it on to the simulation model (scheduling) or perform other actions like visualization of the environment or statistical data (probes), managing user interactions (controllers), logging, API, etc. At least one special plugin called *master scheduler* has to be defined for every run to ensure a basic scheduling. Plugins can be attached or detached to any simulation run providing for a clean way of including and excluding functionality that is optional to a simulation run.

Another point of view on the SPI is the connection to the MVC architectural pattern. MVC, first described by Trygve Reenskaug in 1979 (Reenskaug 1979b; Reenskaug 1979a) and implemented by Jim Althoff for Smalltalk-80, separates user interaction from data processing, letting both be altered independently (Krasner and Pope 1988; Burbeck 1992). The SPI can be seen as a generalization of the MVC pattern as MVC can be implemented in the plugin structure. In fact the visualization and user interaction parts of the EAS Framework presented below are implemented according to the MVC scheme.

Within the MVC paradigm techniques have been developed to handle, for instance, multiple views of an object, a single view that captures aspects from different objects, various controller types working simultaneously, or models physically separated from views and controllers in web applications. These techniques can also be applied to the EAS framework making it possible to flexibly attach and detach views and controllers at runtime and make them completely independent of the (simulation) model.

2.3 The SPI Architecture

The presented architecture is in many respects similar to architectures used in state-of-the-art ABS frameworks like MASON or Swarm. A general claim at design time was to keep the simulation core as slim as possible to provide a hopefully error-free and computationally efficient simulation core. Therefore, inspired by the MASON implementation, the EAS simulation engine consists basically of a loop which repeatedly produces notification messages based on a list of scheduled notifications (at specific time instances or asynchronously occurring events; cf. Sec. 3, class `SimulationTime`). The objects that build the entrance point to the simulation model are called *EAS-Runnables* (ERs). They have a similar function as the "Steppables" in MASON, i. e., progression in time according to notifications.

The novel idea about the SPI architecture is to build an intermediate layer, the SPL, between the simulation engine and the ERs, separating the simulation model from the simulation engine, cf. Fig. 1. One of the SPL's functions is to handle the propagation of notifications from the engine to the simulation model. The simulation engine does not reference the simulation model directly, but sends notifications to the SPL which passes them on to the simulation model. In this way the SPL provides scheduling for

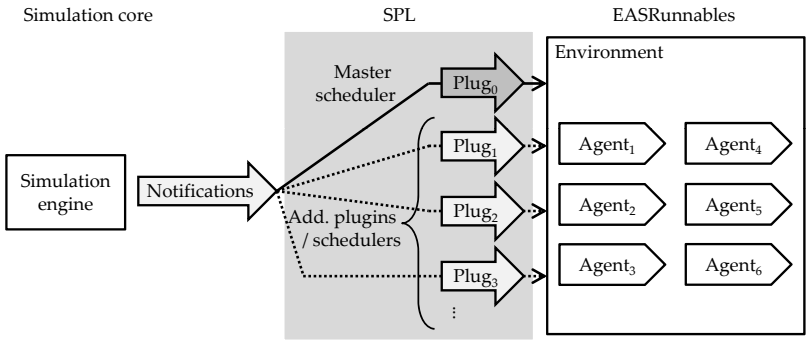


Figure 1: Schematic view on EAS architecture. Notifications are sent from simulation engine to the SPL. The plugins can pass notifications on to the simulation model. One distinguished plugin called *master scheduler* provides for the basic scheduling of the model.

the simulation model by deciding which messages are passed on in which order to the simulation model. This mechanism makes the concept of plugins sufficiently powerful to make simulations using the full SPI completely schedulable. Beyond that all types of functionality can be located within the SPL which are logically neither part of the simulation model nor the simulation core.

The SPL holds objects of the type `Plugin` which is a rather simple Java interface described in detail in Sec. 3. The SPL may contain several plugins at once, however, one of them has to be a distinguished master scheduler which provides for the basic scheduling of the model, cf. Fig. 1. At least the master scheduler has to be present in every run, as otherwise the simulation model would not receive any notifications. The master scheduler also selects the ERs and generates the setting before a run; therefore, by selecting a master scheduler, the scenario to simulate is determined. The plugins' most important method is `runDuringSimulation` that is invoked by the simulation engine at any occurring notification the plugin is registered for. By default, plugins are registered for ticks, but they can cancel them or request further notifications. At any notification, the simulation engine passes the main runnable for the current run to the plugin. The plugin, in turn, can notify the main runnable or subordinate ERs, if desired. Otherwise, the plugin can simply observe the simulation model or intervene in the run when being invoked. In this way, plugins can perform all types of manipulations, observations, etc. required for functionality which should be outsourced from the simulation model.

Additionally, the SPI constitutes a mechanism to build reusable modules as plugins are not bound to a specific type of simulation model. They can be attached to different simulation models by simply combining them with different master schedulers. Therefore, for example, the same learning plugin can be used in a 2D grid environment, a 3D continuous environment or even a more abstract multi-objective optimization scenario without any notion of space.

3 IMPLEMENTATION OF THE EAS FRAMEWORK

This section describes how the basic parts of the EAS framework (i. e., the simulation core and a little additional functionality) are implemented; more generally, it is supposed to show how the SPI can be implemented in an ABS framework using Java. The description covers the core classes of EAS on a verbal and rather informal pseudo-code level, omitting implementation details where not required for understanding the basic concept. The current version (0.12) of EAS can be downloaded including all sources and documentation from sourceforge (http://sourceforge.net/projects/eas-framework).

EAS contains overall 169 packages with 899 Java classes and 92,000 *lines of code (LOC)*. However, the biggest part of this code belongs to implementations of concrete environments and scenarios. The fairly slim simulation core has about 450 LOC and is contained in the packages (1) `eas.simulation` and (2) `eas.plugins`. The simulation core consists of the following classes: `SimulationTime`,

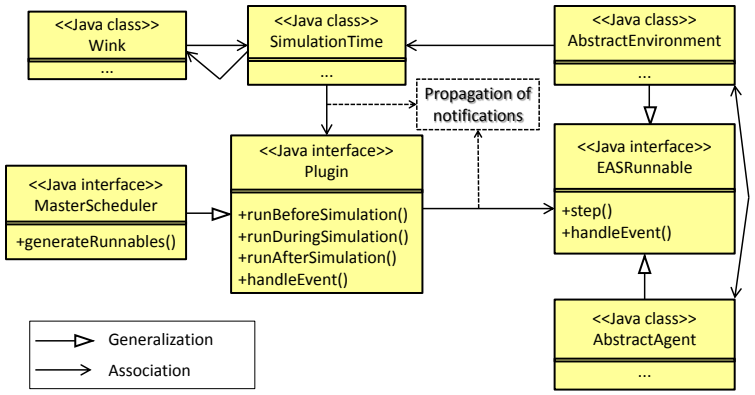


Figure 2: Schematic view on the major classes of EAS including central methods. Method signatures are omitted. Notifications about events and the proceeding of time are fired by class `SimulationTime` and reach the ERs by making a detour over a plugin (usually a master scheduler).

`EASRunnable` (interface) and `Wink` in (1), and `Plugin` (interface) and `MasterScheduler` (interface) in (2). This core implementation is described in the following.

Overview. Fig. 2 shows the interconnection of the basic classes of EAS. The main interface for simulation objects is `EASRunnable` implementing the concept of ERs. Derived from that, the classes `AbstractAgent` and `AbstractEnvironment` provide customized functionality for ABS. The class `Wink` is bound to notifications holding information about the current point in simulation time. The central methods in `EASRunnable` are `step(Wink)` (which receives a `Wink` object) and `handleEvent(EASEvent, Wink)` (which receives a description of an asynchronous event occurred and a `Wink` object denoting the last time instance right before the event occurred). A time or event notification is fired by the class `SimulationTime` and first propagated to all registered plugins (implementing interface `Plugin`). Every plugin, in turn, can pass the notification on to the ERs by invoking their methods `step` or `handleEvent`, respectively. The master scheduler implements the interface `MasterScheduler` which is derived from `Plugin` and offers the additional method `generateRunnables`. The purpose of this method is to generate a list of ERs at the start of the simulation. These ERs are the main runnables which are directly connected to the plugins. They are simulated independently, meaning that each has its own `SimulationTime` object, and in parallel using different threads. Each of these main runnables may contain further ERs similar to the SWARM concept (Daniels 2000) that have no direct connection to the plugins. In the agent-based case, the main runnable is an environment derived from `AbstractEnvironment` that contains arbitrarily many agents derived from `AbstractAgent` which, themselves, are ERs.

Plugins. The interface `Plugin` defines the main functionality of all plugins in EAS. The most important methods of the interface are

(1) `void runBeforeSimulation(EASRunnable, ...)`,
(2) `void runDuringSimulation(EASRunnable, Wink, ...)` and
(3) `void runAfterSimulation(EASRunnable, ...)`, as well as
(4) `void handleEvent(EASRunnable, Event, Wink, ...)`.

As their names suggest, the first three are called by the `SimulationTime` (1) before the beginning, (2) at runtime, and (3) after the last time step of the simulation, respectively. All of them receive an ER object which is a main runnable as defined by the master scheduler. Method (2) is called at each notification the plugin is registered for and additionally receives a `Wink` object denoting the current time instance. From this method the `step` method of the received ER can be called to pass on the notification. A plugin can register for and unsubscribe from notifications to determine what notifications it wants to receive by using

Table 1: Methods for customizing plugin notifications in class `SimulationTime`. The table shows the effects on the notifications of a Plugin P (which can be a master scheduler) after invocation of the methods denoted in the first column. There, $t \in \mathbb{R}$ is an arbitrary point in time, and F is an event filter.

Invocation of method	Time step t	Tick ($\forall t \in \mathbb{N}$)	Event matching F
[standard, no invocation]	no	yes	no
`requestTicks(P)`	[no change]	yes	[no change]
`neglectTicks(P)`	[no change]	no	[no change]
`requestNotification(P, t)`	yes	[no change]	[no change]
`neglectNotification(P, t)`	no	[no change]	[no change]
`requestAllNotifications(P)`	[if t req. by any plugin]	yes	yes
`neglectAllNotifications(P)`	[if req. before]	[if req. before]	[if req. before]
`requestEvents(P, F)`	[no change]	[no change]	yes
`requestAllEvents(P)`	[no change]	[no change]	yes

the `request...` and `neglect...` methods of the `SimulationTime` class (see below). By default, the plugins are notified at all ticks.

The method `handleEvent` (4) is invoked any time an event occurs that matches the event filter defined by the plugin. By default, a plugin is not registered for any events.

Master schedulers. Derived from `Plugin`, the interface `MasterScheduler` offers one additional method:

(5) `EASRunnable[] generateRunnables(...).`

This method is used to generate the main runnables before the beginning of a run. For every ER in the list, a separate `SimulationTime` object is created as well as a (copy of) every plugin object including the master scheduler. At simulation start all the `SimulationTime` objects are triggered, each running in parallel in a different thread and notifying its own list of plugins.

The classes `SimulationTime` and `Wink`. The class `SimulationTime` contains the main simulation loop and is responsible for sending notifications to plugins. Every plugin is notified at least twice: once before the start of the simulation by calling its `runBeforeSimulation` method and once afterwards by calling its `runAfterSimulation` method. Additionally, the plugin can be notified arbitrarily many times in between using the `runDuringSimulation` method and the `handleEvent` method. Both, the `runDuringSimulation` method and the `handleEvent` method receive a `Wink` object which is generated by the `SimulationTime` to contain the current point in `SimulationTime`.

To customize the notification policy for individual plugins, the class `SimulationTime` provides the methods shown in Tab. 1. Using these methods, every plugin can choose which notifications to receive in future. Note, that the method `requestAllNotifications` informs a plugin about all time steps that have been requested by any other plugin as well as all ticks and all events. The complementary method `neglectAllNotifications` does not lead to that plugin never being notified again (this behavior is given by a method called `neverNotifyAgain`); rather, it recreates the state before calling the method `requestAllNotifications` or has no effect if the latter has not been called before.

The interface `EASRunnable`. This interface has to be implemented by every object that is to be clocked by the simulation engine (e. g., environments and agents). The method `step` defines how an object progresses in the course of time. It should perform a state change of the simulated object depending on its current state and the current simulation time. The method `handleEvent` defines how the ER reacts on events.

The ERs' `step` and `handleEvent` methods can be invoked by the master scheduler of a run, but also by other plugins or ERs. For example, an environment usually contains a list of agents as subsequent ERs whose `step` and `handleEvent` methods can be invoked by the environment itself.

Running a simulation. The class `SimulationStarter` is used to start a simulation run. This process is performed in five steps. (1) The master scheduler is selected and constructed. (2) The main runnables $R_1, \ldots R_n$ ($n \in \mathbb{N}$) are generated by using the master scheduler's `generateRunnables` method. (3) A

set of `SimulationTime` objects $T_1,\ldots,T_n$ is created, and every T_i is connected to R_i ($i \in \{1,\ldots,n\}$). (4) For the selected plugins $P_0,P_1,\ldots,P_m$ ($m \in \mathbb{N}_0$; P_0 being the master scheduler), a list of copies $P_{1_i},\ldots,P_{m_i}$ ($i \in \{1,\ldots,n\}$) is created, and the list $(P_{1_i},\ldots,P_{m_i})$ is connected to R_i. Therefore, every main runnable is connected to its own copies of the `SimulationTime` and `Plugin` objects. (5) Each of the `SimulationTime` objects $T_1,\ldots,T_n$ is started in a separate thread.

To start a simulation, the `main` method of the class `SimulationStarter` has to be started with program parameters specifying the master scheduler and other plugins. Beyond that, EAS provides a mechanism to create new parameters flexibly corresponding to each plugin, to make the plugin behavior adjustable at program start. To simplify this process, there is a graphical user interface (GUI) provided by the class `Starter` which guides the selection of plugins and other program parameters and the start of a simulation. Due to space limitations, these and other additional functionalities (for example, the sensor / actuator concept and the implementation of agent brains) are not described here.

Getting started. To start developing an own agent simulation in EAS, typically three steps have to be done. (1) Create an environment class derived from `AbstractEnvironment`; (2) create one or more agent class(es) derived from `AbstractAgent`; (3) create a master scheduler derived from `AbstractDefaultMaster` (a helper class with typically needed scheduling behavior) that generates a setting with the new environment and agents. The abbreviation "EAS" can also be used as a mnemonic aid to remember the recommended implementation sequence "Environment → Agent → Scheduler". For a first start, the only real implementations (besides empty method bodies) have to be performed in the methods `generateRunnables` and `id` of the master scheduler. The latter simply has to return an identification string used to select the scheduler. EAS notices the new master scheduler class automatically using reflections and makes its id appear in the list of master schedulers in the `Starter` GUI. Selecting it and the "videoplugin" from the list of plugins, the new scenario can be started and visualized.

4 EVALUATION

In this section, a study is described where EAS is compared to the ABS frameworks MASON and NetLogo. MASON was chosen for its architectural comparability to EAS as both are Java-based ABS frameworks and flexible enough to build various complex scenarios. NetLogo, on the other hand, was chosen as it is often used as an access point for beginners to ABS due to its simplicity. The frameworks are compared to each other with respect to the objective of guiding unskilled users to well-structured code.

4.1 Method

The study took place within an advanced laboratory course with six students as test persons during one semester (18 weeks), starting in October 2011. As a test scenario the well-known Stupid Model (Railsback, Lytinen, and Grimm 2005) was used. Tab. 2 shows a summary of the 16 tasks of the Stupid Model.

Every test person had to individually implement the complete Stupid Model within each of the three frameworks. The test persons were allowed to discuss implementation questions with their instructors, however, to every simulation program a specific instructor has been assigned. To keep the results as unbiased as possible of the instructors' background knowledge of the simulation platforms, the instructors had no former experiences with the simulation they were assigned to, but started working with it all at the same time before the beginning of the semester.

Overall, the test persons were not able to complete 20 of the $16 \cdot 3 \cdot 6 = 288$ tasks (6 in EAS and NetLogo, 8 in MASON). These cases were counted by using the mean value of all the other test persons for the according task and simulation. All test persons had earlier experience with Java, none knew MASON or EAS. While two of them had worked with NetLogo before, the rest had no experience with ABS at all.

After the implementation phase, the students were anonymously asked a series of survey questions. The complete survey including questions and results can be downloaded in SPSS and Excel format (in German) from http://www.aifb.kit.edu/images/0/0e/AllSurveyDataWSC2012.zip.

Table 2: Summary of the 16 tasks of the Stupid Model.

No.	Task
1	100 agents ("bugs") distributed randomly into a 100^2 grid space (movement and visualization).
2	A second bug action: growing by a constant amount.
3	Habitat cells that grow food; bug growth is equal to the food they consume from their cell.
4	"Probes" letting the user see the instance variables of selected cells and bugs.
5	Parameter displays letting the user change the value of key parameters at run time.
6	A histogram of bug sizes.
7	A stopping rule that causes execution to end when any bug reaches a size of 1000.
8	File output of the minimum, mean, and maximum bug sizes each time step.
9	Randomization of the order in which bugs move.
10	Size-ordering of execution order: bugs move in descending size order.
11	Optimal movement: bugs move to the cell within a radius of 4 that provides highest growth.
12	Bugs have a constant mortality probability, reproduce when they reach a size of 10.
13	A graph of the number of bugs.
14	Initial bug sizes drawn from a random normal distribution.
15	Cell food production rates read from an input file; graphical display of cell food availability.
16	A second "species": predator agents that hunt bugs.

Additionally, a structural analysis of the LOC and the number of files (corresponding to classes in the Java case as no inner classes were used) in relation to the tasks of the Stupid Model have been performed. A normal distribution has been assumed to underlie the data. *Standard deviation (SD)* and Student's t-test have been calculated to indicate significance.

The program versions used in the study have been 0.9 for EAS, 15 for MASON and 4.1.3 for NetLogo (note that a later version of EAS was already available, but has not been used as some improvements have been performed after studying the Stupid Model specification, which possibly would have biased the results). Eclipse has been used as an IDE for the Java-based frameworks. The program `cloc` (http://cloc.sourceforge.net/) has been used to count LOC.

4.2 Results and Discussion

Evaluation of surveys. The survey questions can be clustered in four main parts. (1)-(3) were simulation-specific and had to be answered right after completing the respective implementations, (4) consisted of comparative questions, that had to be answered after the completion of the implementation in all three frameworks.

Unsurprisingly, the well established frameworks MASON and NetLogo took the lead in the comparing questions (4), especially concerning available documentation and examples. The installation of the framework and the launch of provided examples were reported to be easiest in NetLogo. Also, certain functionality (mainly charts and parameter changing) that were available in MASON and NetLogo were lacking EAS at the time showing that EAS is not quite as elaborate as the other two, yet.

In one major block of the simulation-specific surveys (1)-(3), the students had to specify the relative amount of time they spent for completing each of the 16 tasks as well as the total absolute time for the whole Stupid Model per framework (Fig. 3). To avoid subjective impressions, the students had to distribute 800 Points in total to the 16 tasks (with a maximum of 100 points per task). These points were later recalculated to the actual time consumption per task, utilizing the total time required.

As Fig. 3 clearly indicates, visualization of charts was the most time-consuming task in EAS. We attribute this to the fact that the corresponding support-features were implemented in EAS only after the survey was taken. This is supported by the code analysis, too (cf. below, Fig. 4).

It is noticeable that the complex scheduling (task 10) seemed to be significantly more complicated to the test persons in MASON than in any of the other frameworks, although the LOC required for this were less than in EAS. Our conclusion is, that in the latter it was more obvious how to accomplish modified scheduling.

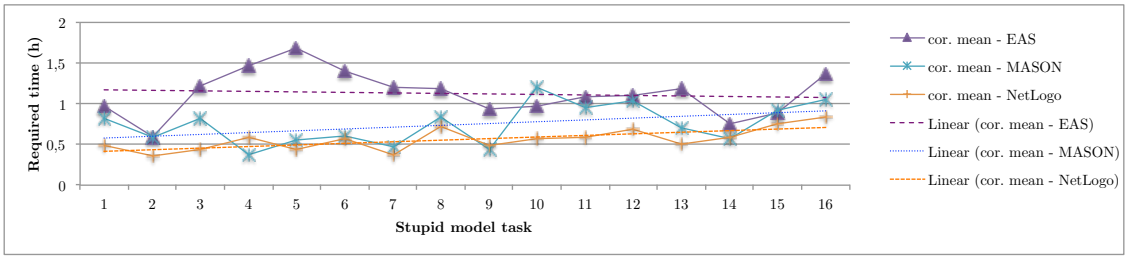
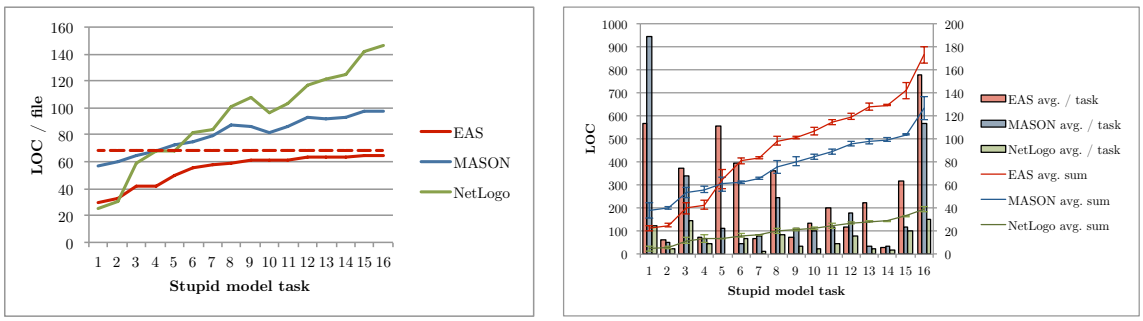


Figure 3: Reported time required to implement tasks 1-16 of the Stupid Model

Overall, time and effort were lowest in NetLogo, followed by MASON and then EAS. Still there is evidence that the cost for implementing features in NetLogo were lowest for the first (simpler) tasks and grew gradually with the rising complexity of the tasks (cf. Fig 3). A similar development could be observed for MASON, while there is no such trend obvious for EAS. This could indicate, that the architectural structure of EAS supports additions and extensions of simulations more easily. Of course it has to be considered that the Stupid Model is a very simple model even at its highest level, and that the results might not be generalizable to more complex scenarios.

Structural analysis of the code. The number of files and LOC can be related to the structuredness of code. In object-oriented (OO) languages it is a desirable goal to keep the number of LOC per file low. In (Bay 2008) a number as low as 50 LOC per class is stated as a maximum. The measure does not apply perfectly to NetLogo as it is not OO, but file size indicates structuredness here, too. In NetLogo, a program is usually read and executed from a single file. However, some of the test persons chose to introduce a second file at some point to improve structuredness by distributing the 16 subtasks on two places (leading to a lot of redundancy; nobody chose to use more than two files). In these cases, the total number of files has been counted, but the duplicate parts of the code appearing in both files have been counted only once.

A two-tailed t-test has been performed for all pairs of simulations on the mean summed-up LOC and mean summed-up number of files per test person, respectively. It indicates that all the mean values are pairwise different with a probability of at least 99.9% except for the LOC of MASON vs. EAS where the probability still calculates to more than 99.0%.



(a) (b)

Figure 4: Measures of LOC and file count for the three simulations. (a) Average LOC / file, summed up incrementally over the 16 Stupid Model tasks (dashed line: LOC / file in EAS `users` package). (b) Total lines of code for each of the 16 Stupid Model tasks individually (bars, right axis), and summed up incrementally (lines, left axis), respectively; error bars denote SD for LOC of individual tasks (not sum).

Fig. 4(a) shows the number of LOC / file for the three frameworks summed up over the 16 subtasks. All three simulations break through the bound of 50 LOC / file. However, overall EAS needs clearly the least LOC / file in comparison to the other two. It seems that the LOC / file stagnate at around 70.

This assumption has been rechecked by calculating the average over all EAS-based scenarios implemented so far in the `eas.users` package (containing code from various users, students as well highly skilled developers, but excluding the code of this study); overall this calculated to $\frac{27,670}{405} \approx 68.3$ LOC / class, cf. dashed line in Fig. 4(a). A similar test with MASON's `sim.app` package lead to $\frac{12.738}{165} = 77.2$ LOC / class. Of course, these data has to be considered with caution as the code was produced at least partially by skilled users which is not the case for our study. NetLogo's LOC / file count obviously increases essentially linearly with the LOC which is correlated to the complexity of the task.

When comparing the absolute LOC required in total for the Stupid Model (Fig. 4(b)), NetLogo required the fewest (195; SD=12), while EAS required the most (856; SD=36.5), and MASON was in between (632; SD=49). It is notable that for tasks 5, 6 and 13 the difference between EAS and the other two frameworks is considerably high. We relate this to the lack of chart and parameter changing support in EAS, which was implemented in a later version. Ignoring these outliers, the differences between the Java-based frameworks in LOC development are not that high. From that perspective, EAS code contains a little more LOC, but has considerably less LOC / class.

Another observation is that file operations (tasks 8 and 15) were reported to be time-consuming for the first time (task 8) in all frameworks (cf. Fig. 3), but for the second time (task 15) the time consumption for EAS is significantly lower than for the other two frameworks. Relating this finding with the LOC counts for the according tasks (Fig. 4(a)), the number of new LOC required for task 15 is as high as for task 8 in EAS. This indicates that code was copied and pasted and could be seen as a hint to include better file access functionality in EAS.

In conclusion, we state that EAS is a young simulation platform that is not as elaborate as MASON and NetLogo in terms of predefined functionality, and that it lacks documentation. Therefore, more LOC and in sum more time has been needed to implement the Stupid Model in EAS. However, our findings indicate that the EAS architecture leads to code that is better structured and can be reused more easily than with the other two simulations. Relating this to the SPI concept, especially the tasks 5, 6, 8, and 13 have been implemented as plugins, while tasks 9 and 10 were mostly implemented using the scheduling capability of the master scheduler. For the first task, EAS offers a reusable plugin ("videoplugin") that can be simply attached to any simulation without requiring further implementations. In contrast, MASON needs a lot more code here which is mainly due to the rather complicated (although very flexible and elaborate) implementation of visualization in MASON.

5 CONCLUSION AND FUTURE WORK

In this paper an architectural concept called SPI for ABS frameworks has been presented as well as an ABS framework called EAS which implements the SPI. The concept is intended to improve modularity and structuredness as well as reusability of the developed code. Moreover, the goal is to not only allow implementations to have these properties, but to actively guide particularly unskilled users in an intuitive way to a well-structured implementation. To achieve these objectives, the SPI introduces an additional layer between simulation engine and simulation model allowing to place there all the functionality that is logically separable from the simulation model. The extracted modules can be attached and detached flexibly, and used in completely different scenarios, making the code reusable.

The concept has been evaluated by comparing EAS to the state-of-the-art simulation frameworks MASON and NetLogo. Six test persons, most unfamiliar with any of the simulations, implemented individually the Stupid Model using all three simulations. The results indicate that the implementations in EAS are structured significantly better in terms of LOC / class and reusability of the code than in both MASON and NetLogo. Moreover, the SPI concept has been credited by the test persons in a anonymous post-implementation survey as beneficial compared to the other simulations. When being asked to choose a simulation framework for a future assignment, EAS came of fairly well by being chosen twice, which was the same number as for MASON and NetLogo. However, the total LOC number is highest in EAS. This can be partially explained by the high number of classes required in EAS, but it is certainly also

due to the elaborateness of the two other simulations that have both been developed for about a decade or more. Especially NetLogo comes with a lot of pre-assembled components that can be integrated using a GUI without requiring any code (but being rather inflexible in return).

The SPI architecture is not bound to the EAS framework, but can be implemented within most state-of-the-art simulation frameworks. In future, a MASON version of the SPI is planned. Furthermore, a major objective is to improve EAS from a functional point of view meaning that more standard functionality should be provided. For example, after studying the main needs of the test persons during implementing the Stupid Model, a plugin for visualizing charts and another for changing parameters at runtime have already been implemented.

REFERENCES

Barcelo, J., E. Codina, J. Casas, J. L. Ferrer, and D. Garcia. 2005. "Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems". *Journal of Intelligent and Robotic Systems* 41:173–203.

Bay, J. 2008. *The ThoughtWorks Anthology: Essays on Software Technology and Innovation*, Chapter Chapter 6: Object calisthenics. O'Reilly Series. Dallas, TX: Pragmatic Bookshelf.

Burbeck, S. 1992. "Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)". http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

Casas, J., J. L. Ferrer, D. Garcia, J. Perarnau, and A. Torday. 2010. "Traffic Simulation with Aimsun". In *Fundamentals of Traffic Simulation*, edited by J. Barcelo, Volume 145 of *International Series in Operations Research and Management Science*, 173–232. Springer New York.

Daniels, M. 2000. "Integrating simulation technologies with swarm". In *Proceedings of the Workshop on Agent Simulation: Applications, Models, and Tools*, edited by C. M. Macal and D. Sallach, 1–16.

König, L., S. Mostaghim, and H. Schmeck. 2009. "Decentralized Evolution of Robotic Behavior using Finite State Machines". *Int. Journal of Intelligent Computing and Cybernetics* 2 (4): 695–723.

Krasner, G. E., and S. T. Pope. 1988, August. "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80". *Journal of Object-Oriented Programming* 1 (3): 26–49.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. "MASON: A Multiagent Simulation Environment". *SIMULATION* 81 (7): 517–527.

Minar, N. 1996. *The swarm simulation system : a toolkit for building multi-agent simulations*. Santa Fe Institute.

Cynthia Nikolai and Gregory Madey 2009, March. "Tools of the Trade: A Survey of Various Agent Based Modeling Platforms". http://jasss.soc.surrey.ac.uk/12/2/2.html.

Steve Railsback and Steve Lytinen and Volker Grimm 2005. "StupidModel and extensions: A template and teaching tool for agent-based modeling platforms". http://condor.depaul.edu/slytinen/abm/StupidModel/.

Railsback, S. F., S. L. Lytinen, and S. K. Jackson. 2006, September. "Agent-based Simulation Platforms: Review and Development Recommendations". *Simulation* 82 (9): 609–623.

Reenskaug, Trygve 1979a, December. "Models-Views-Controllers". http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

Reenskaug, Trygve 1979b, May. "Thing-Model-View-Editor – an Example from a planningsystem". http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf.

Siebers, P.-O., and U. Aickelin. 2008. "Introduction to Multi-Agent Simulation". *CoRR* abs/0803.3905.

Sklar, E. 2007. "NetLogo, a Multi-agent Simulation Environment". *Artificial Life* 13 (3): 303–311.

Standish, R. K. 2006. "Going Stupid with EcoLab". *Simulation* 84 (3): 611–618.

AUTHOR BIOGRAPHIES

LUKAS KÖNIG is a PhD Student and a Research Associate at Karlsruhe Institute of Technology (KIT), Germany. He graduated from University of Stuttgart, Germany, with a degree in Computer Science in

2007. He started research in evolutionary swarm robotics in 2006 and research in agent-based simulation in 2007. He is particularly interested in improving the synergies between academic research and teaching by using advanced simulation techniques. His email address is lukas.koenig@kit.edu.

DANIEL PATHMAPERUMA is a PhD Student and a Research Associate at Karlsruhe Institute of Technology (KIT), Germany. He studied computer science and did his diploma at the KIT. After working as a developing engineer for the Orbster GmbH in Karlsruhe until in July 2009, he joined the KIT again in the group of Hartmut Schmeck. Besides coordinating the smart grid and smart traffic projects *MeRegio*, *MeRegioMobil* and *iZEUS*, he works on different simulation projects, especially in the context of mobile robotics. His email address is daniel.pathmaperuma@kit.edu.

FELIX VOGEL is a PhD Student and a Research Associate at KIT, Germany. He graduated from University of Munich, Germany, with a degree in Media Computer Science in 2007. He started research in personalizing web forms in web applications in 2008, and he is working on a project to improve teaching through simulation since 2011. His email address is felix.vogel@kit.edu.

HARTMUT SCHMECK is a Full Professor of Applied Informatics at KIT, Germany. His current major research interest is on self-organization and adaptivity in complex technical systems with a special focus on the challenges of future traffic and energy systems. He is a Key Member of the *Organic Computing* Initiative and has been the Coordinator of the German Priority Programme on Organic Computing. He is the Scientific Spokesperson of the KIT-Focus COMMputation addressing the inherent combination of communication and computation which is a characteristic feature of smart application systems. His email address is hartmut.schmeck@kit.edu.