DIVAs 4.0: A Multi-Agent Based Simulation Framework

M. Al-Zinati, F. Araujo, D. Kuiper, J. Valente, R. Z. Wenkstern
Department of Computer Science
The University of Texas at Dallas
Richardson, TX, 75080
Email: {mohammad.al-zinati, frederico.araujo, kuiper, juniavalente, rymw}@utdallas.edu

***Abstract*—In this paper we present DIVAs 4.0, a framework that supports the development of large-scale agent-based simulation systems where agents are situated in open environments. DIVAs includes high-level abstractions for the definition of agents and open environments, a microkernel for the management of the simulation workflow, domain-specific libraries for the rapid development of simulations, and reusable, extendable components for the control and visualization of simulations. We illustrate the use of DIVAs through the development of a simple simulator where virtual agents are situated in a virtual city.**

***Keywords*—Agent based simulation systems; simulation framework;**

I. Introduction

Multi-Agent Based Simulation Systems (MABS) have provided new perspectives on modeling and simulating complex problems. While traditional simulation systems have been limited to a certain class of applications, MABS have employed the powerful concepts of adaptation, emergence and self-organization to model complex, real-world problems. Many domain specific MABS have been developed over the past two decades [1, 2, 3, 4]. Even though these systems have addressed important issues in domains such as social or traffic simulations they are not reusable outside of their application area. On the other hand, the multi-agent system community has spent effort developing generic frameworks for MABS [5, 6, 7]. These frameworks provide the basic building blocks, i.e., architectures, software components and libraries for the development of a variety of agent-based simulation systems. Unfortunately, none supports the development of MABS where the environment is *open* (i.e., inaccessible, non-deterministic, dynamic and continuous). This represents a major weakness since realistic simulations require the modeling of dynamic environments that can only be partially perceived by the agents.

Over the past several years we have developed a framework for the development of large scale multi-agent based simulation systems for complex domains. The framework called DIVAs (**D**ynamic **I**nformation **V**isualization of **A**gent **s**ystems) offers reusable architectures, abstract classes, software components and libraries to support the development of enterprise-scale simulation systems. DIVAs is based on the premise that agents and environment play an equally important role in MABS. Agents are situated in an open environment that is partially perceived, and the environment is totally decoupled from agents. Such a clear separation of duties leads naturally to extensible, reusable architectures. In addition, DIVAs offers means to dynamically access and modify agent and environment properties at run-time, a unique feature that none of the existing frameworks offers.

In the following section we give an overview of related works. In Section III we give an overview of DIVAs architecture. In Sections IV-VII we discuss the various components of DIVAs, namely the Agent System, the Environment System, the Microkernel, and the GUI and Visualizer. In Section VIII, we briefly discuss how DIVAs can be used to create a simple urban city environment and in Section IX we give some experimental results. The content of this paper is related to the non-distributed version of DIVAs.

II. Related Works

Over the past years, a number of multi-agent based simulation systems (MABS) tool suites have been proposed. These include Netlogo [8], AgentSheets [9], SeSam [10]. While these systems offer an integrated graphical environment for specifying, interpreting, and executing simulations, they do not scale well to realistic, complex scenarios. They are also difficult to extend and adapt since their architectures are not easily reusable outside of their application domains.

In the remainder of this section, we restrict our discussion to those tools that best compare to DIVAs such as the generic "framework and library" platforms of Repast S [6, 11, 12], MASON [5] and GAMA [7, 13]. These platforms provide architectures, software components and libraries to design and implement a variety of agent-based simulation systems. We discuss these platforms with respect to their architecture and the type of environment they promote.

A. Architecture

A pluggable architecture is a desired feature in simulation frameworks. It supports the rapid development of simulation system by providing a flexible and easy way to integrate and/or remove self-contained modules.

Among the frameworks discussed in this section, only MASON is based on a pluggable architecture. MASON consists of three main components: 1) the Model Component corresponds to the simulation core. It provides a collection of classes

1550-6525/13 $26.00 © 2013 IEEE
DOI 10.1109/DS-RT.2013.19

for discrete event scheduler and schedule utilities; 2) the Visualization Component provides a GUI-based visualization for the simulation; 3) the Utility Component contains a set of utility classes e.g., random number generator, GUI widgets snapshot-generating facilities.

B. Environment

In our discussion on simulation environments, we focus on three main criteria: separation between agent and environment concepts, environment openness and environment structure.

1) Separation between agent and environment concepts: In modern MABS, the virtual environment plays an essential role in a simulation: it supplies the "physical" conditions for the virtual agents to exist, provides agents with information about their surrounding, enforces physical laws, etc. Researchers have stated that virtual environments should be decoupled from agents and be treated as a first class entity in MABS [14].

Among the aforementioned frameworks, only Repast S, and GAMA provide a clear separation between the environment and agent concepts and consider the environment as an important component of the simulation system.

2) Openness: In order to model realistic simulations, it is necessary for the simulated environment to be *open*: virtual agents should only access the environmental information they they can perceive; the effect of an action or event in the environment should not be known with certainty in advance; the environment should not be static but should undergo changes as a result of actions or events; and finally the environment states should not be enumerable.

None of the frameworks discussed in this section incorporates an open environment model.

3) Environment Structure: In order to develop large scale simulation systems with thousands of agents perceiving their surroundings while interacting with each other, it is necessary to design the virtual environment in such a way that both its structure and control are decentralized.

With respect to the virtual environment structure all frameworks have a decentralized environment structure where the environment is partitioned into smaller area. In Repast S, *contexts* represent containers for sets of environment objects. In MASON, the environment can be represented (optionally) as an aggregation of *fields* which associate simulated objects with locations. In GAMA, the environment is divided into *places* which store perception data and allow agents to access this data.

Even though the aforementioned frameworks have a decentralized environment structure, all of them incorporate a centralized control strategy. Centralized control creates a bottleneck for large scale real-time simulations and limits the scalability of the simulation.

In this paper we present DIVAs 4.0, a framework for the development of large scale simulation systems where agents are situated in an environment. The unique characteristics of the DIVAs framework are:

1) It can be used to implement a variety of simulation systems in different domains.
2) It provides a pluggable architecture with a collection of reusable abstract classes and software components that allow the rapid development of complex simulation systems.
3) It provides the necessary building blocks for the implementation of:
   a) virtual *open* environments with decentralized structure and control,
   b) and virtual agents that can dynamically perceive their surroundings through various senses while interacting and/or collaborating with one another.
4) It provides pluggable 2D and 3D visualizers.
5) It provides editors that can be integrated in simulation systems to allow the run-time property modification of simulated agents (e.g., change goal, modify sensor values) and the virtual environment (e.g., add/modify/remove environment objects, trigger external events).

To the best of our knowledge, no other existing framework offers this integrated set of features.

In the following section we give an overview of DIVAs architecture.

III. Overview of DIVAs 4.0

As shown in Figure 1, DIVAs consists of four main modules: 1) the Simulation Module; 2) the Message Transport Service; 3) the Control and Visualization Module and 4) the Data Management System.

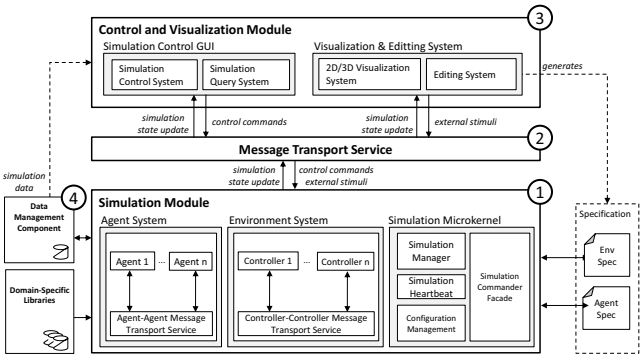

Fig. 1. DIVAs framework high-level architecture illustrating its main modules

The framework's main constituent, i.e., the Simulation Module, creates large-scale simulation instances. It consists of three subsystems. The *Agent System* creates and manages agents. The *Environment System* creates and manages

a dynamic environment where agents are situated. The *Simulation Microkernel* manages the workflow of the simulation and provides mechanisms for loading/storing agent and environment specifications from/to persistent storage (e.g., xml file).

The second main component of DIVAs 4.0 is the *Message Transport Service* (MTS). The MTS's role is to provide a messaging infrastructure to allow different elements of the simulation to communicate with each other through a common set of interfaces. For instance, it is through the MTS that simulation elements exchange messages containing simulation state updates, external stimuli, and control commands.

The interactive *Control and Visualization Module* receives information from the MTS, renders 2D and 3D images of the simulation, and provides mechanisms for users to interact with the simulation and modify simulation parameters at run-time. It consists of two subsystems: the Simulation Control GUI and the Visualization & Editing System.

The *Simulation Control GUI* allows users to control a running simulation (i.e., start/stop the simulation, adjust parameters, save/load simulation states) through the Simulation Control System. It also allows users to query detailed properties of simulation entities (e.g., existing traffic lights) through the Simulation Query System.

The *Visualization & Editing System* includes a 2D/3D Visualization System which generates 2D and 3D representations of the simulation and provides tools for users to interact with the simulation at run-time (e.g., trigger events, modify agent properties). It also includes the Editing System which allows users to specify a virtual environment (e.g., build a virtual city) and modify the virtual environment at run-time (e.g., add/edit/delete environment objects).

Finally, DIVAs 4.0 architecture defines a *Data Management System* (DMS) which is responsible for storing and processing information collected from the simulation for data analysis as well as *Domain-Specific Libraries* which allow for rapid development of simulation platforms for specific domains. Currently, DIVAs embeds models for social and traffic simulation domains.

In the following sections, we discuss the various components of DIVAs 4.0, starting with the Agent System.

IV. The Agent System

A. Concepts

As shown in Figure 2, in DIVAs, an agent consists of four main modules [15]. The *Interaction Module* handles an agent's interaction with external entities and separates environment interaction from agent interaction. An agent communicates with other agents through the Agent Communication Module. It receives environmental data (e.g., agent states, environment object states, external event information) from the Environment Perception Module. The *Knowledge Module* is partitioned into External Knowledge Module (EKM) and Internal Knowledge Module (IKM). The EKM serves as the portion of the agent's memory dedicated to maintaining knowledge about entities external to the agent, i.e., acquaintances, environment objects situated in the environment. The IKM serves as the portion of the agent's memory dedicated for keeping information that the agent knows about itself (i.e., current state, physical constraints, social limitations). The *Task Module* manages the specification of the atomic tasks that the agent can perform (e.g., walk, run). The *Planning and Control Module* serves as the brain of the agent; it uses information provided by the other modules to react to critical situations, plan, initiate tasks, make decisions, and achieve the agent's goals.

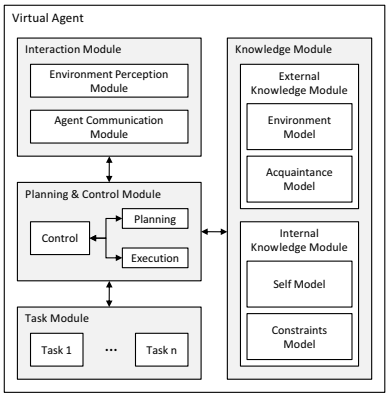

Fig. 2. Agent architecture showing the agent's main components.

B. Agents in DIVAs 4.0

In DIVAS, the `Divas-Core.Agent` package (see Figure 3) implements the high-level agent architecture discussed above and encapsulates packages that correspond to the agent's main modules. `Divas-Core.Agent` structure as well as the mechanisms that relate the various sub-packages and some components are implemented and are intended to be reused. Nevertheless, due to its generic nature, `Divas-Core.Agent` also includes abstract components that need to be instantiated. Therefore, in order to create domain-specific agents (e.g., virtual humans), a developer is required to "fill in the blanks" by either providing concrete implementations for the abstract components or by reusing the appropriate predefined components available in DIVAs libraries. DIVAs 4.0 comes with a library of components for virtual human agents and vehicle agents.

In the remainder of this section we discuss the various packages.

1) Interaction Module: Figure 4 shows the `Divas-Core.Agent.Interaction` package. This package consists of two sub-packages `Perception` and `Communication` that include the components necessary to implement the Environment Perception Module and the Agent Communication Module in DIVAs' agent architecture (see

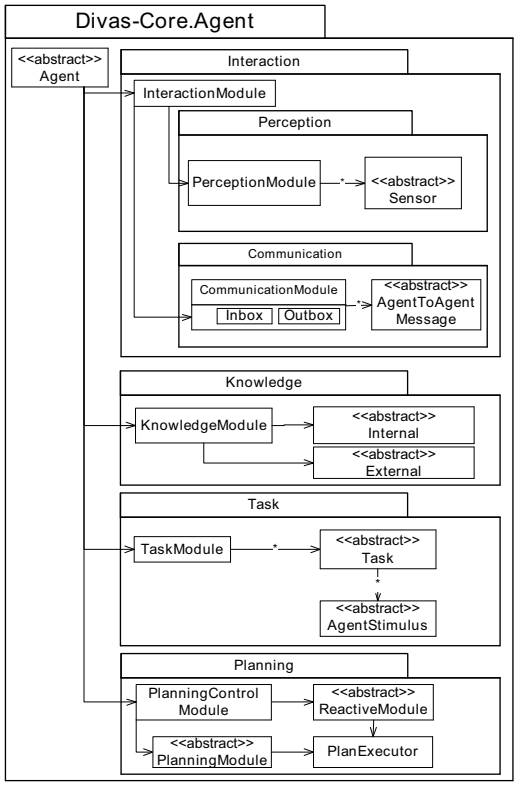

Fig. 3. Divas-Core.Agent design package

Figure 2).

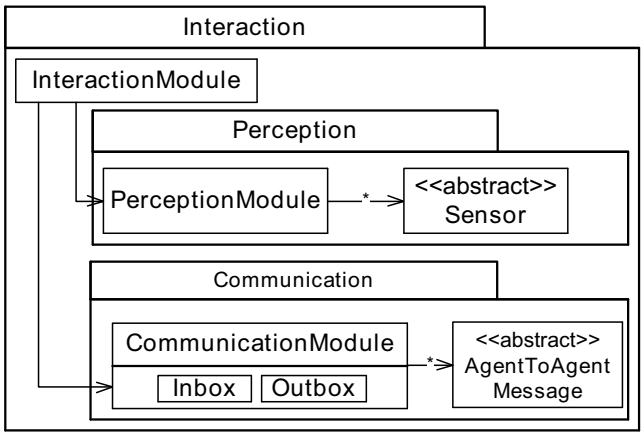

Fig. 4. Divas-Core.Agent.Interaction package

`PerceptionModule` implements the mechanisms necessary for an agent to make use of sensors to perceive its environment. It is associated with abstract `Sensor`s that need to be defined for domain specific virtual agents. For instance, in the case of a virtual human agent, a developer may elect to reuse the vision, auditory or olfactory sensors available in the DIVAs virtual human component library or define a new sensor. More details on agent perception in DIVAs can be found in [16, 17, 18].

The `CommunicationModule` handles all the agent-to-agent communications. Agents communicate with each other by exchanging `AgentToAgentMessage`s. Each `AgentToAgentMessage` contains information about the sender and receiver agent, message type, level of priority, time the message was sent, and the message itself. The `CommunicationModule` provides each agent with an `inbox` and an `outbox`. Outgoing messages are processed by a messaging service that sends messages asynchronously through the *Agent-Agent Message Transport Service*. The `CommunicationModule` is fully implemented and only requires instances `AgentToAgentMessage`s to execute.

2) Knowledge Module: The `Divas-Core.Agent.Knowledge` package shown in Figure 5 includes the components that correspond to Knowledge Module in the DIVAs' agent architecture.

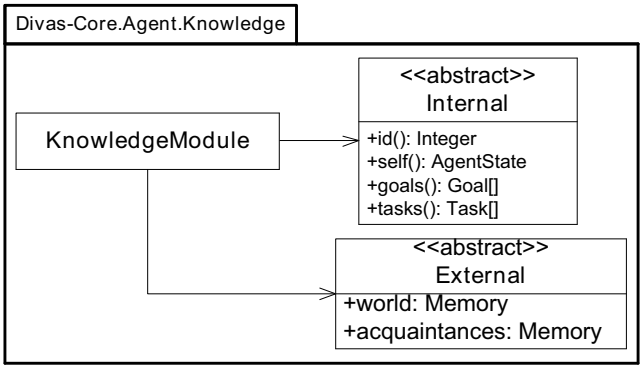

Fig. 5. Divas-Core.Agent.KnowledgeModule package

The `Internal` package consists of knowledge the agent knows about itself such as its state, its goals, and its constraints. This knowledge can be predefined or acquired at run time. For example, if we wish to endow a virtual human agent with the knowledge that a bomb is life-threatening, this information will be stored in `+self()` at initialization time.

The `External` package consists of knowledge acquired by perceiving the environment via the `PerceptionModule`, such as events triggered in the environment, nearby agents, or environment objects.

3) Task Module: The `Divas-Core.Agent.Task` package shown in Figure 6 includes the components that implement the Task Module in the DIVAs' agent architecture. This package consists of a set of abstract atomic `Task`s associated with `AgentStimuli`.

The `TaskModule` aggregates a set of abstract tasks which are associated with abstract stimuli. In a domain-specific simulation, its role is to load concrete tasks for virtual agents. Therefore for domain-specific agents, it is necessary to

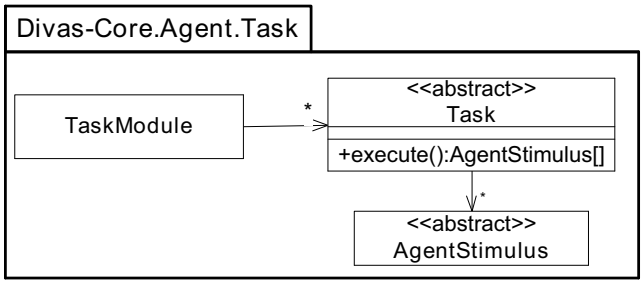

Fig. 6. Divas-Core.Agent.Task package

provide concrete implementation for each task the agent can perform and define the stimuli resulting from the execution of this task. For example, in order to implement a human virtual agent, a developer may define a concrete `MoveTask` and its associated stimulus, i.e., $newPosition(x, y, z)$ or import a list of predefined tasks (e.g., `TurnTask`, `OpenDoorTask`, etc.) and their corresponding stimuli from the DIVAs library.

4) Planning and Control Module: The `Divas-Core.Agent.Planning` package shown in Figure 7 includes the components needed to implement the Planning and Control Module in the DIVAs' agent architecture. This package consists of `PlanningControlModule`, `ReactiveModule`, `PlanningModule`, and `PlanExecutor`.

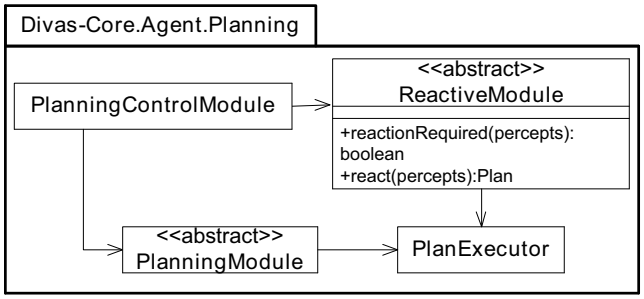

Fig. 7. Divas-Core.Agent.PlanningControl package

As mentioned earlier, `PlanningControlModule` serves as the brain of the agent. Its main role is to determine when to execute the Reactive or Planning modules. For instance, in critical situations, such as an explosion, this module will execute the `ReactiveModule`, whereas in other cases, the `PlanningModule` will be used to plan new goals, tasks, or decide on a set of actions to perform.

For domain-specific virtual agents, it is necessary to either provide concrete implementations of `ReactiveModule` and `PlanningModule` or reuse existing modules from DIVAs library.

As a result of the execution of either module, a `Plan` is generated. This plan consists of a set of `Tasks` the agent has decided to perform (e.g., open a door and exit a room, flee from an explosion). This plan is then executed by the predefined `PlanExecutor`. This results in a set of `AgentStimuli` which are handled by the environment.

C. Illustrative Example: Implementing a Virtual Human Agent in DIVAs 4.0

Figure 8 shows a representation of a concrete implementation of a human agent using reusable component obtained from the DIVAs library.

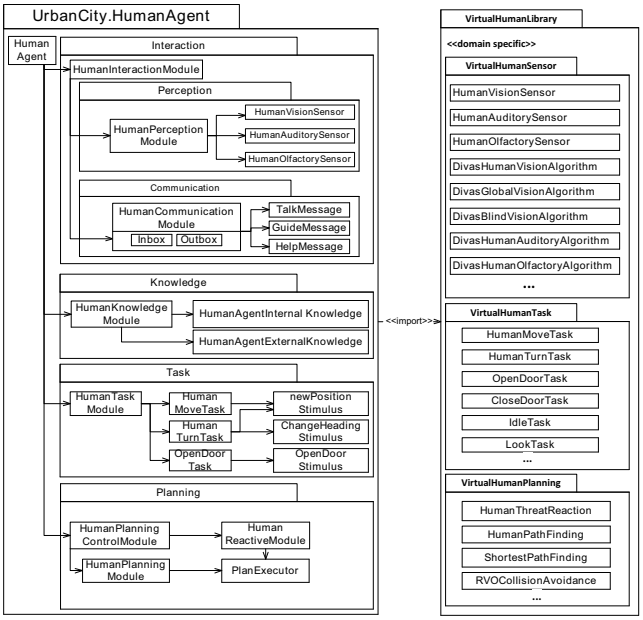

Fig. 8. Concrete Virtual Human Agent

```
public abstract class VirtualAgent<S extends AgentState,
KM extends KnowledgeModule<S>, IM extends InteractionModule,
PCM extends PlanningModule<KM, TM>, TM extends TaskModule<KM>>
{
   protected KM          knowledgeModule;
   protected IM          interactionModule;
   protected PCM         planningModule;
   protected TM          taskModule;

   public VirtualAgent(S state)
   {
      knowledgeModule = createKnowledgeModule(state);
      interactionModule = createInteractionModule(knowledgeModule);
      taskModule = createTaskModule(knowledgeModule);
      planningModule = createPlanningModule(knowledgeModule,
         taskModule, interactionModule);
   }

   protected abstract KM createKnowledgeModule(S state);
   protected abstract IM createInteractionModule(KM knowledgeModule);
   protected abstract PCM createPlanningModule(KM knowledgeModule,
      TM taskModule, IM interactionModule);
   protected abstract TM createTaskModule(KM knowledgeModule);
   protected abstract Stimuli generateStimuli();

  ...
}
```

Fig. 9. Code for Abstract Agent

```
public class HumanAgent extends VirtualAgent<HumanAgentState, HumanKnowledgeModule,
      HumanInteractionModule, HumanPlanningModule, HumanTaskModule>
{
   public HumanAgent(HumanAgentState state)
   {
      super(state);
   }

   @Override
   protected HumanKnowledgeModule createKnowledgeModule(HumanAgentState state)
   {
      return new HumanKnowledgeModule(state);
   }

   @Override
   protected HumanInteractionModule createInteractionModule
                        (HumanKnowledgeModule knowledgeModule)
   {
      return new HumanInteractionModule
                        (new HumanPerceptionModule(knowledgeModule),
        new SimpleAgentCommunicationModule(getId()));
   }

   @Override
   protected HumanPlanningModule createPlanningModule
         (HumanKnowledgeModule knowledgeModule, HumanTaskModule taskModule,
   HumanInteractionModule interactionModule)
   {
      HumanPlanGenerator planGenerator = new HumanPlanGenerator
    (knowledgeModule, taskModule, interactionModule, pathFinding);
      HumanPlanExecutor planExecutor = new HumanPlanExecutor(knowledgeModule);
      HumanReactionModule reactionModule = new HumanReactionModule
    (knowledgeModule, taskModule, interactionModule, pathFinding);
      return new HumanPlanningModule(planGenerator, planExecutor,
       knowledgeModule, reactionModule);
   }

   @Override
   protected HumanTaskModule createTaskModule
                     (HumanKnowledgeModule knowledgeModule)
   {
      return new HumanTaskModule(knowledgeModule);
   }
...
}
```

Fig. 10. Code for Virtual Human Agent

As shown in Figure 8, a `HumanAgent` is a concrete implementation of the abstract DIVAs' core `Agent`. For example, the abstract virtual human perception module is implemented by importing predefined human sensors (e.g., vision, auditory, olfactory) from the `VirtualHumanLibrary`. The same applies to `HumanTaskModule`. In regards to the `HumanPlanningModule`, the developer can either implement his/her own plan strategy and path finding or reuse modules from the VirtualHumanLibrary.

For the sake of conciseness, in Figure 9 and 10 we show portion of the code for abstract agent and virtual human agent. It is straightforward to notice that `HumanAgent` is a specialization of `Agent`.

V. The Environment System

A. Concepts

In DIVAs, virtual agents are situated in a large *virtual environment* which is *open*, i.e., inaccessible, non-deterministic, dynamic, and continuous [19].

DIVAs is based on the premise that in order to manage a large environment efficiently, it is necessary to partition the space into smaller defined areas called *cells* (see Figure 11). Each cell is managed by a special-purpose agent called *cell controller*. A cell controller is responsible for being aware

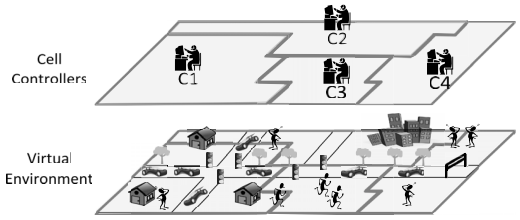

Fig. 11. Decentralized virtual environment showing the partitioning of the environment into *cells*, each cell being managed by a *cell controller*.

of the virtual agents located in its defined area; interacting with local virtual agents to inform them about changes in their surroundings; and interacting with adjacent cell controllers to inform them of external events and their propagation. It is important to note that cell controllers are design-specific agents that do not have counterparts in domain-specific applications.

B. Environment in DIVAs 4.0

The `Divas-Core.Environment` package shown in Figure 12 includes the main components of DIVAs environment. The abstract `Environment` is divided into `cells`. Each cell carries a portion of the environment state and aggregates environment entities such as `Agent`s, `EnvironmentObject`s, and `EnvironmentEvent`s, i.e., externally triggered events that may influence the agents and objects situated in the environment.

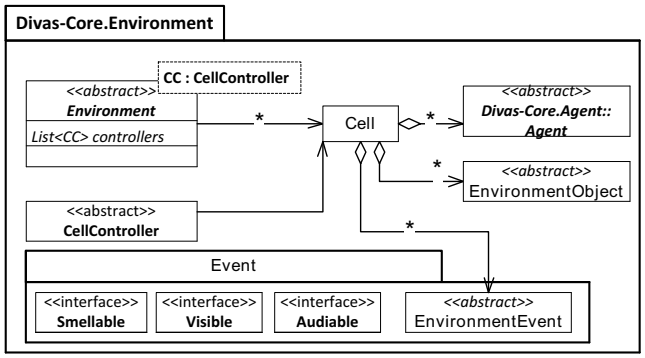

Fig. 12. Divas-Core.Environment package

1) Processing Agent Stimuli: As mentioned in Section IV-B3, when a virtual agent situated in a cell, say C_1 decides to execute a task, it produces stimuli (i.e., action intents) that are synchronously passed on to the cell controller CC_1 (i.e., the cell controller responsible for managing the cell in which the agent is located). In addition, any external stimulus (e.g., explosion) triggered during that simulation cycle and affecting C_1 is communicated to CC_1. The cell controller combines all stimuli, resolves conflicting intents (e.g., two agents want to be in the same (x, y, z) position at the same time) and returns to each agent an updated state that is legal with respect to the law of the environment. All special cases involving, for example, agents crossing boundaries, or agents stepping

into an obstructed position in an adjacent cell are handled in this phase. The abstract `CellController` class include the workflow for agent stimuli processing. Nevertheless, tasks such as stimuli combination of conflict resolution need to be defined by the developer for domain-specific environments.

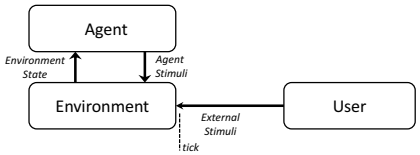

Fig. 13. Model showing agent-environment interactions and external stimuli.

2) Propagating Events: Figure 14 shows the `Event` package. Each concrete event in DIVAs extends the abstract `EnvironmentEvent`, which contains common attributes and abstract methods for events (e.g., `eventID`, `age`, `origin`, `intensity`). In addition, concrete events may realize one or more of the following marker interfaces: `Visible`, `Audible`, and `Smellable`. The effects of a concrete event over the environment vary depending on which interface the event is marked with.

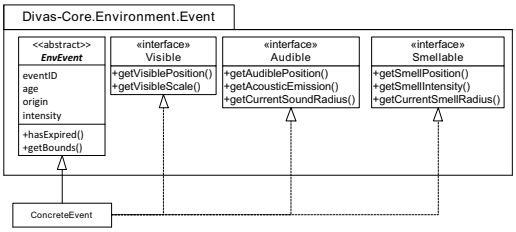

Fig. 14. Event package

Each cell controller propagates the events it contains. As simulated events propagate over time and space, their effects may cross cell boundaries and span multiple cells. In this case, interaction between neighboring cell controllers are required to ensure that the propagation is handled properly. For each event in a given cell, the controller computes the new state of the event at the current simulation cycle. Then, if an event has expired, the controller removes it from the cell state and sends a message to other cells to inform them that an event has expired and must be removed. Otherwise (if the event has not expired), the controller checks if the event is leaving its cells' boundaries; if it is the case, then the controller forwards the event influences to the appropriate adjacent controller.

C. Illustrative Example: Implementing an Urban City Environment in DIVAs 4.0

We first start by defining, `UrbanCityCellController`, a concrete implementation for the abstract `CellController`. `UrbanCityCellController` implements event propagation mechanisms, conflict resolution and stimuli combination algorithms for physical environments. Then we create the `UrbanCityEnvironment` by binding the

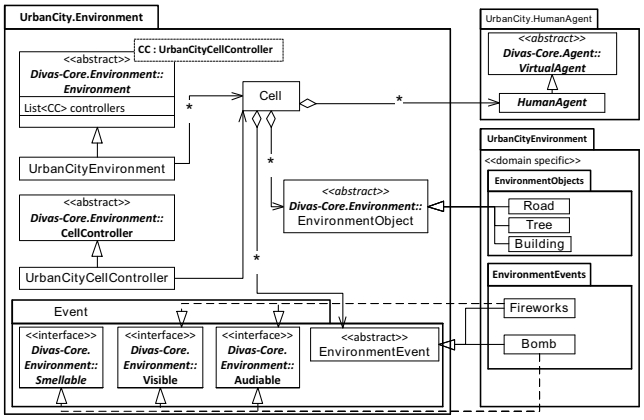

Fig. 15. Urban City Environment

`UrbanCityCellController` to the abstract `Environment`. We proceed by either implementing or importing environment objects such as buildings, roads and trees, and do the same for environment events. Figure 15 shows a representation of a concrete implementation of an urban city environment.

VI. The Microkernel

DIVAs encapsulates its most important core services in a *microkernel*. Core services include the simulation heartbeat, communication infrastructure, configuration management, and thread management.

As illustrated in Figure 16, the microkernel package `Divas-Core.Microkernel` aggregates several core components of the simulation framework. `SimCommanderFacade` provides a uniform interface to simulation services (e.g., start/stop simulation, add agent, edit environment object). `CommModule` provides the mechanisms for sending and receiving messages through the MTS. `ConfigurationManager` handles run-time configuration changes to simulation parameters and user settings such as minimum cycle time or default visible distance. `IDManager` is responsible for assigning unique IDs to simulated entities. The `SimulationCore` is an abstract class which orchestrates the workflow of the simulation.

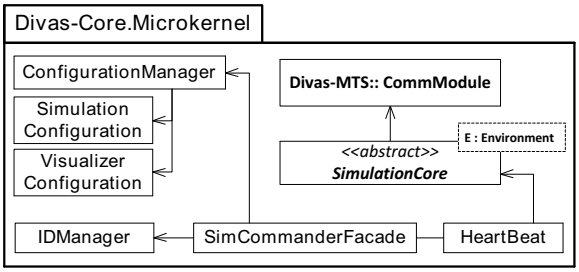

Fig. 16. Simulation microkernel package

`Heartbeat` acts as a time keeper and discrete event generator for the simulation. In DIVAs the `Heartbeat` ticks at the completion each of two distinct phases: the *environment* phase

and the *agent* phase. The execution of the environment and the agent phases constitutes the *simulation cycle* (see Figure 17).

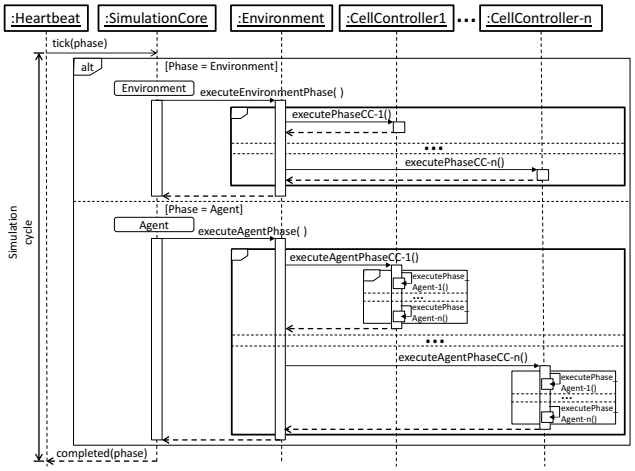

Fig. 17. Simulation Cycle

During the environment phase, the `Environment` object which is responsible for managing threads allocated to cell controllers triggers the concurrent execution of cell controllers. In this phase, each controller 1) *deliberates*, i.e., combines agent and external stimuli, resolves conflicts and determines the new state of its cell; then 2) *reacts*, i.e., publishes its cell's new state for data collection and visualization.

During the agent phase, the `Environment` object requests that each cell controller triggers the execution of its agents. In this phase, agents 1) perceive their new environment state, 2) deliberate to determine their next course of action; and 3) execute tasks which are in turn converted into stimuli and passed on to the environment. A new simulation cycle starts.

VII. The GUI and Visualizer

The user can interact with the simulation at run-time using the Divas-GUI and Divas-Visualization components.

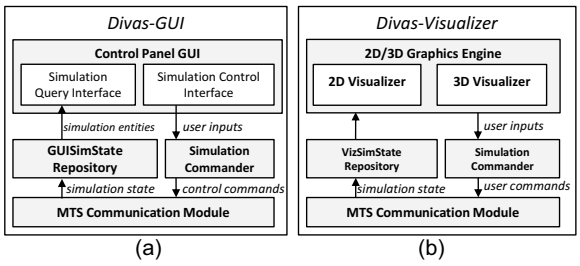

Fig. 18. User-Simulation interaction components featuring (a) Divas-GUI and (b) Divas-Visualization

Figure 18 (a) shows the architecture of Divas-GUI. At every simulation cycle, the updated simulation state received through the MTS is stored in the *GUI Repository* and is used to update displayed simulation statistics (e.g., agent count, objects count) and control information (e.g., simulation cycle time, simulation cycle number). Users can submit queries about agents (e.g., find agent #131) or environment objects (e.g., display information about all buildings in the environment) through the *Simulation Query Interface*. In addition, users can use the *Simulation Control Interface* to control the simulation (e.g., start/stop, pause, save). User's inputs are converted into control command messages which are transmitted to the Divas-Core through the MTS.

Figure 18 (b) shows the architecture of Divas-Visualization. As in Divas-GUI, updated simulation states received through the MTS are stored in the *VizSimState Repository* and are used to create 2D and 3D representations of the simulated entities. Through the 2D or 3D visualizer, users can interact with the simulation at run-time (e.g., trigger an event) or edit properties of the environment (e.g., add environment object, edit environment object). User interaction inputs are converted into user command messages by the `Simulation Commander` and transmitted to the Divas-Core through the MTS as external stimuli messages.

Figure 19 shows the snapshot of a 3D visualizer developed using the Divas-Visualizer component. The tool box in the right provides an interface for the user to add virtual agents, environment objects and trigger events, while the dialog box in the left allows users to change individual agent properties at run-time.

Fig. 19. A 3D visualizer created using the DIVAs-Visualization component showing a 3D visualization of an urban traffic simulation.

A. Illustrative Example: Reusing DIVAs 4.0 Visualizer

To reuse the visualizer, it is necessary to provide models for the simulated entities. In DIVAs, these models are referred to as Visualized Objects. In the case of the urban city, we import 3D models for virtual humans, environment objects such as buildings, trees and roads, and external events such as bombs and fireworks. Then we customize the visualizer dialog panel to visualize human agent properties such as location, velocity, sense intensity or field of view. As mentioned earlier, the purpose of the Visualizer Commander is to allow users of the simulation to trigger predefined external events during the execution of the simulation. Therefore, it is necessary to specify what events will be used during the simulation.

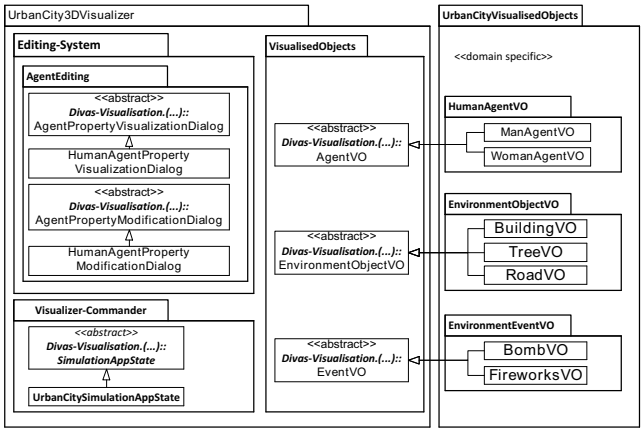

Fig. 20. Urban City 3D Visualizer

VIII. Illustrative Example

In this section, we give a brief overview of the steps necessary to develop a simple simulator using DIVAs 4.0. The urban city is an open environment thats consists of buildings, roads, trees, benches, etc. and includes virtual human agents (see Figure 21). The virtual agents are capable of perceiving their surroundings through vision, auditory and olfactory sensors. They execute complex path-finding and collision avoidance algorithms to move within the environment. In addition, they interact with other agents, plan and deliberate to achieve their goals (e.g., move to location, search for agents).

Fig. 21. Urban City Simulation

In order to develop the urban city simulator, it is necessary to define concrete implementations of `DIVAs-Core.Agent` and `DIVAs-Core.Environment`. This is achieved by following the steps discussed in Sections IV-C and V-C. Then, we integrate `DIVAs-Core.Microkernel` with `UrbanCityEnvironment` bound to the environment template `E` (see Section VI). This is followed by the customization and the integration of the simulation visualizer and the GUI (see Section VII-A).

A demo on the development and execution of the urban city simulator is available at http://mavs.utdallas.edu/projects/divas

IX. Scalability Evaluation

In this section we run the urban simulator discussed in Section VIII and evaluate the simulation performance with respect to scalability. The virtual urban city environment consists of 814 environment objects (e.g., commercial buildings, houses, traffic lights, roads). The environment is partitioned into 64 equally sized cells. Agents are scattered in various areas and walk randomly in the city.

In order to evaluate the scalability of the simulator, we make use of the simulation cycle time, i.e., the time elapsed in each simulation cycle measured in milliseconds. More precisely, for each set of agents placed in the environment, we record the average simulation cycle time every 30 seconds. The real-time requirement for the simulation is 150 milliseconds. This corresponds to the longest time the visualizer can display the simulation without delay.

The urban city simulator used to run the experiment was executed on a multicore PC (Intel Core i7 X980 CPU (3.33GHz), 6.00 GB, 64-bit Windows 7) and implemented in Java (version 1.7.0, 64-bit).

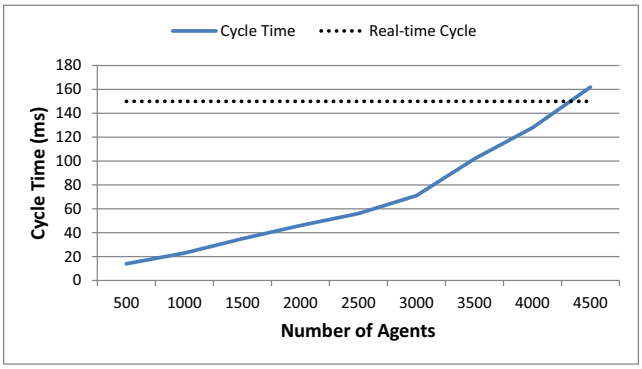

Fig. 22. Average simulation cycle time for a 64-cells environment.

The results given in Figure 22 show an increase in the simulation cycle time as the number of agents augments. This is to be expected since agents' execution consumes significant computational resources. However, the simulator was able to handle $\approx$ 4500 virtual agents before reaching the real-time requirement constraint of 150 milliseconds. The high number of agents handled by the simulator is the result of the decentralized structure and control of the environment.

X. Conclusion

In this paper we presented DIVAs 4.0, a framework for the development of large-scale agent-based simulation systems where agents are situated in open environments. DIVAs provides architectures and abstract classes for the definition of agents and open environments, a microkernel for the management of the simulation workflow, domain-specific libraries for the rapid development of simulations, and reusable, extendable components for the control and visualization of simulations. We illustrated the use of DIVAs through the development of a

simple simulator where virtual agents are situated in an open environment representing a virtual city. The results show that the simulator is capable of executing a very large number of agents in simulated real-time.

Even though most of the building blocks for DIVAs 4.0 have been developed, more needs to be done. For example, new domain specific libraries to be defined; a graphical agent specification tool has to be developed; and various environment structures (e.g., self-organizing) need to be investigated.

Acknowledgments DIVAs is supported by Rockwell-Collins grant 5-25143. The DIVAs project has been under investigation for several years. This work would not have been possible without the contributions of Travis Steel, Renee Steiner, and Gary Leask.

REFERENCES

[1] N. Pelechano, J. Allbeck, and N. Badler, "Controlling individual agents in high-density crowd simulation," in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. San Diego, California: Eurographics Association, 2007, pp. 99–108.

[2] K. Uno and K. Kashiyama, "Development of simulation system for the disaster evacuation based on multi-agent model using GIS," *Tsinghua Science & Technology*, vol. 13, no. 1, pp. 348–353, October 2008.

[3] S. Sharma, H. Singh, and A. Prakash, "Multi-agent modeling and simulation of human behavior in aircraft evacuations," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 44, no. 4, pp. 1477–1488, 2008.

[4] S. Sharma, "Simulation and modeling of group behavior during emergency evacuation," in *Proceedings of the IEEE Symposium on Intelligent Agents*. Nashville, Tennessee: IEEE, 2009, pp. 122–127.

[5] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: A multiagent simulation environment," *Simulation*, vol. 81, no. 7, pp. 517–527, 2005.

[6] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos, "The repast simphony runtime system," in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Chicago, Illinois, 2005, pp. 151–158.

[7] E. Amouroux, T.-Q. Chu, A. Boucher, and A. Drogoul, "Gama: An environment for implementing and running spatially explicit multi-agent simulations," in *Proceedings of the 10th Pacific Rim International Conference on Multi-Agent Systems (PRIMA 2007), Bangkok, Thailand, November 21-23, 2007*, also in Lecture Notes in Computer Science, vol. 5044, pp 359-371, Springer Heidelberg, 2009.

[8] "Wilensky, u. 1999. netlogo," http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL., Accessed June 2013.

[9] A. Repenning, A. Ioannidou, and J. Zola, "Agentsheets: End-user programmable simulations," *Journal of Artificial Societies and Social Simulation*, vol. 3, no. 3, 2000, Retrieved June 28, 2013 from http://jasss.soc.surrey.ac.uk/3/3/forum/1.html.

[10] F. Klügl, R. Herrler, and M. Fehler, "Sesam: implementation of agent-based simulation using visual programming," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. Hakodate, Japan: ACM, 2006, pp. 1439–1440.

[11] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos, "The repast simphony development environment," in *Proceedings of the Agent 2005 Conference on Generative Social Processes, Models, and Mechanisms*, Chicago, Illinois, 2005, pp. 159–166.

[12] T. R. Howe, N. T. Collier, M. J. North, M. T. Parker, and J. R. Vos, "Containing agents: Contexts, projections, and agents," in *Proceedings of the Agent 2006 Conference on Social Agents*, Chicago, Illinois, 2006, pp. 107–113.

[13] P. Taillandier, D. Vo, E. Amouroux, and A. Drogoul, "Gama: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control," *Principles and Practice of Multi-Agent Systems*, vol. 7057, pp. 242–258, 2012.

[14] D. Weyns, H. Dyke Parunak, F. Michel, T. Holvoet, and J. Ferber, "Environments for multiagent systems state-of-the-art and research challenges," in *Environments for Multi-Agent Systems*, ser. Lecture Notes in Computer Science, D. Weyns, H. Dyke Parunak, and F. Michel, Eds. Springer Berlin Heidelberg, 2005, vol. 3374, pp. 1–47.

[15] R. Z. Mili (Wenkstern), R. Steiner, and E. Oladimeji, "DIVAs: Illustrating an abstract architecture for agent-environment simulation systems," *Multiagent and Grid Systems, Special Issue on Agent-oriented Software Development Methodologies*, vol. 2, no. 4, pp. 505–525, 2006.

[16] T. Steel, D. Kuiper, and R. Wenkstern, "Virtual agent perception in multi-agent based simulation systems," in *Proceedings of IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (IAT-10)*, vol. 2. Toronto, Canada: IEEE, 2010, pp. 453–456.

[17] D. Kuiper and R. Z. Wenkstern, "Virtual agent perception in large scale multi-agent based simulation systems," in *The 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*. Taipei, Taiwan: IFAAMAS, 2011, pp. 1235–1236.

[18] D. M. Kuiper and R. Z. Wenkstern, "Virtual agent perception combination in multi agent based systems," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems (AAMAS 2013)*. Minnesota, USA: IFAAMAS, 2013, pp. 611–618.

[19] S. Russell and P. Norvig, *Artificial Intelligence A modern approach*, 1995.