中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-15-004

An Agent-Based Disaster Simulation Environment

Tzu-Liang Hsu and J. W.S.Liu

Apr. 07, 2015 || Technical Report No. TR-IIS-15-004
http://www.iis.sinica.edu.tw/page/library/TechReport/tr2015/tr15.html

Institute of Information Science, Academia Sinica
Technical Report TR-IIS-15-004

An Agent-Based Disaster Simulation Environment

Tzu-Liang Hsu and J. W.S.Liu

ABSTRACT

Agent-Based Disaster Simulation Environment (ABDiSE) is a framework that provides model elements and tools to support modeling and simulation of common types of natural disasters, including fires, floods and debris flows. The underlying disaster model is agent based: Active objects describe how agents move, attach, and interact with each other and with their environment. ABDiSE is extensible: New agent types and external simulators needed to model elements and dynamics of new disaster scenarios and define behaviors and interactions of agents can be added without requiring revision and recompilation of the framework. ABDiSE is a multi-threaded, capable of taking advantage of available computing resource to speed up simulation.

Keywords: Agent-based model, Disaster simulation, Development environment

An Agent-Based Disaster Simulation Environment

Tzu-Liang Hsu and J. W. S. Liu

1 INTRODUCTION

In recent years, agent-based modeling (ABM) and simulation have proven to be an effective tool for application domains as diverse as economical and market trend prediction; complex system design; fluid, traffic and customer flow analysis; and so on [1-6]. According to ABM, entities capable of taking actions are modeled as active objects called agents. Each *agent* is defined by a behavior specification and a set of interaction rules. During simulation, each agent assesses its situation, makes decisions and takes actions in accordance with its behavior specification and interaction rules, and through the rules, taking into account the behavior of other agents and states of the environment.

This paper presents an agent-based framework called *Agent-Based Disaster Simulation Environment,* or *ABDiSE* for short. The framework contains an extensible library of agents of various types with which one can construct agent-based models of elements that cause natural disasters (including fires, floods and debris flows) and elements of geographical areas (including buildings, trees, and roads) that are affected by disasters. The framework also contains an engine for executing the models and thus simulating the causes and developments of natural disasters for the sake of understanding and predicting their dynamics and impacts. The model elements provided by the framework build on common-sense concepts and notions. By using them, complex processes of how a small mishaps (e.g., a cinder touching a dry leave) develop into a major calamity (e.g., a forest fire) can be modeled with the desired fidelity in an intuitive but formal and executable way.

To illustrate, Figure 1 shows a screenshot of the main window of ABDiSE. The user can access from the GUI (Graphical User Interface) tools provided by an experiment manager. These tools enable the user to select and retrieve model elements from the model library and use them to construct the simulation model, set up and control the simulation experiment, and configure the simulation engine. We will explain in subsequent sections the available agent types and the design of the experiment manager and experiment management tools.

As one can see from Figure 1, the most prominently displayed tool is the *Map Explorer* in area B where a flood affected area is displayed in its entirety as an example. Built on the cross platform and open source .NET control GMAP.NET [7], the tool allows the user to control

map attributes, including map provider (Bing, Google, and so on), overlay, marker and zooming. In particular, it provides the user with an easy way to specify locations of agents in a common geographic information system (GIS) format and have them displayed on the map. In this example, small circles and squares in the area represent two types of agents which the user has already selected from the agent library to be included in the simulation model and placed them on the map as indicated. The user can also visualize in Area B the development of the disaster scenario during the simulation run.

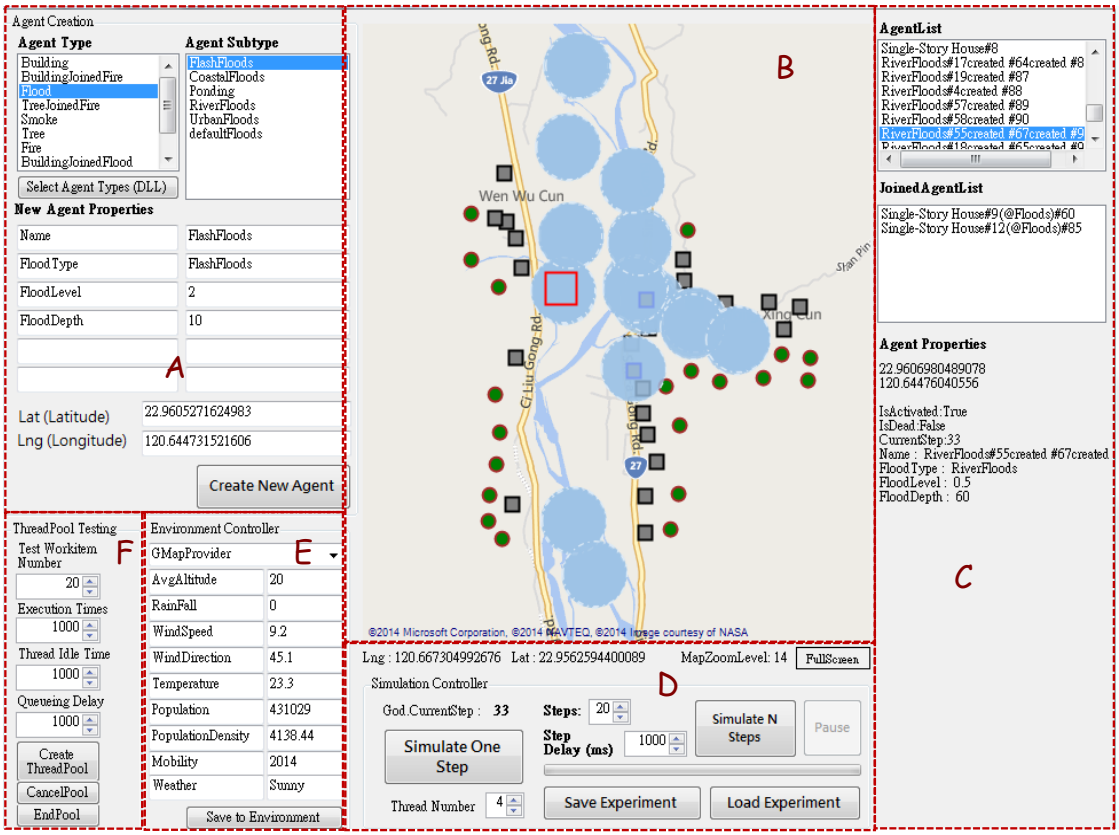

Figure 1 Main window of ABDiSE GUI

Area A is used for selecting agents to be included in the simulation model: To include an agent of a specific type that is provided by the model library, the user only needs to first select an agent type (e.g., flood) and then select a subtype (e.g., flash flood). If none of the available agent types suits the disaster scenario which the user wants to simulate, the user can add a new agent type and/or subtype by clicking the `Select Agent Types` (DLL) button: This button enables the user to add new agent type(s) or remove some existing type(s) (e.g., add fire type and remove flood type): The user adds a new agent type (e.g., tornado) to the library by programming a customized Dynamic Link Library (DLL) function. The function will be called during simulation to update the behavior of agents of the type. External simulation

programs can also be incorporated into the model in a similar way. We will return to provide further details on this extensibility feature.

Properties of agents are stored in C# dictionary form. When the user selects an agent type for a new agent in the simulation model, the default parameter values of the agent, including the one and only `Name` property of the agent, is automatically filled in Area A, and an entry for the agent is created in the dictionary. The user can freely assign properties to the new agent except for its `Name`. Among the important properties of an agent are its coordinates that specifies its location. The user can assign the latitude (`lat`), longitude (`lng`) and altitude of a new agent by typing their values in the textboxes in Area A. (The values of `lat` and `lng` can also be entered by double left clicking its location in Area B.) After all required agent properties have been entered, the user then clicks `Create New Agent` button. Clicking it creates a new agent according to the agent parameters entered in Area A. If the create operation is successful, the new agent marker will appear in Map Explorer after the entry of the agent in the dictionary is committed.

At any point in time, the upper panel in Area C lists all the agents included in the simulation model. The lower panel displays the properties of a selected agent (e.g., the one marked by the red square in B and highlighted in C in Figure 1.)

Within ABDiSE, simulation is time driven: The engine executes one step at a time, and the update method of each agent is executed once in each step. Using buttons and text boxes in Area D, the user can control the length of the time for each step, as well as the number of time steps to execute during the current simulation. A special model element is environment, which is included in every simulation model to take into account of the effect of environmental condition on the development of natural disasters. Text boxes in Area E allow the user to set environment properties (e.g., wind speed and direction, rainfall rate, etc.), and thus, provide the user with control over environment during simulation. Finally, text boxes in Area F allow the user to configure and test the simulation engine, in particular, whether the multi-threaded engine executes correctly and performs well.

The contributions of the work described here include the open source framework ABDiSE for disaster simulation. Currently, simulators of natural disasters (e.g., [8-16]) are by and large built one at a time, each for a specific type of disaster, locale, disaster situation, and so on. ABDiSE aims to enable simulators for disasters of different types at different places and likely scenarios to be easily constructed from model elements. Specifically, the framework

supports common GIS data format and provides easy to use GUI and tools to input information and model elements of the simulation world. Any time after a disaster scenario is setup and simulation progresses, the user can save the current state of the simulation world as a checkpoint and the initial condition for a later experiment. Extensibility is a distinguishing feature of ABDiSE. Agents and simulators needed to model the dynamics of new disaster types and scenarios can be added without modification and recompilation of ABDiSE core controller: When a user wants to add one or more agent types, he/she only needs to write the new agent classes, compile them to DLL (dynamic link library) files, and restart ABDiSE. ABDiSE core controller will dynamically load the DLL functions and thus enable the user to simulate new disaster scenarios modeled by the new type(s) of agents.

The remainder of the paper is organized as follows. Section 2 presents related work on disaster simulators and agent-based models and simulation. As it will be evident, they differ from ABDiSE both in purposes and in design and functionalities. Section 3 describes ABDiSE agent-based model elements. Section 4 describes the design and architecture of the framework. Section 5 describes the implementation of the ABDiSE simulation engine and the interaction between ABDiSE Core and GUI. Section 6 first presents as case studies examples illustrating the use of ABDiSE for modeling scenarios of past disasters. It then summarizes the paper and discusses future work.

2 RELATED WORK

As stated in the previous section, there are numerous disaster simulators for specific types of disasters. An example is the advanced fire simulator Fire Dynamics Simulator (FDS) [8-10], which was created at the US National Institute of standards and Technology (NIST). The combination of FDS simulation [8] of smoke and heat transport based on computational fluid dynamic models and visualization support from Smokeview program [9, 10] can provide answers to critical questions on causes of rapid spread of fire and smoke within a building and time required for residents to escape from the building. Similar to FDS, ABDiSE breaks a large-scale simulation into computation of the behavior and interactions of elements within each of the small grid cells. The incorporation of solutions of Navier-Stokes equations used in FDS into ABDiSE is possible and is a part of our future work.

Flood simulation can provide data and information on where an inundation may occur, and its arrival time, flood wave speed and depth for various scenarios, and so on, critical to

effective flood emergency and reclamation management. Numerous flood simulators are now available for such proposes. Examples include Monticello Dam simulation and generation of inundation map [11] based on the MIKE21 model (the Danish Hydraulic Institute 2-D hydrodynamic model [12]) and simulation of Amazon flood pulse based on the 2-D hydrodynamic model LISFLOOD-FP to quantify and predict the exchange between the Amazon main stem and its floodplains [13, 14]. Simulation has also been an approach used to study long term effects of climate change and to create high-resolution, static sea-level rise image of specific area (e.g., Manhattan in New York) for a specific sea level rise value [15, 16]. This is a simple way to estimate flood disaster damage. Similar to these simulators, ABDiSE also supports GIS file format and scenarios creation with Google Earth Map. The default flood simulation is based on the altitude attribute in a way similar to the one used in [15, 16]. Again, the user can incorporate more advanced flood models in ABDiSE by editing the behavior rules of flood agent type and rules of individual flood agent instances and structuring and compiling flood simulation programs into DLLs and use them for update methods of the new flood agents.

Being a framework of tools and a runtime support system for the construction and execution of agent-based models of natural disasters, ABDiSE more closely resembles many existing toolkits for the development of agent-based applications, in terms of its approach, goals and design. There is no universal agreement on the exact definition of the term agent in the context of agent-based modeling [17, 18]. Among all the definitions, the one most appropriate for ABDiSE is the one defined for control systems [19]: An agent is an encapsulated computer system or system component that is situated in some environment and can act flexibly and autonomously to meet its design objectives [20]. ABDiSE captures the core ideas of common agent-based models. Agents in the framework have the typically attributes of agents. In addition to interactions such as contention for space and other common interactions mentioned in [18], agents in ABDiSE can attach to each other and become joined agents. Joined agents are models of entities such as burning trees, flooded homes, etc. Section 3 will present the definition of attach and its use in modeling dynamics of disasters.

Specifically, the ABDiSE framework was motivated by many existing toolkits for developing agent-based models and simulators [6, 21], including Repast (Recursive Porous Agent Simulation Toolkit) [22], AnyLogic [23], and Natural Disaster Complex Systems Framework [24]. Repast is a widely used open-source, cross-platform, agent-based modeling

and simulation toolkit. The toolkit supports flexible models of living social agents (i.e., agents that are permeable, interleaved, and mutually defining) and models of belief systems, agents, organizations, and institutions as recursive social constructions. Repast includes a variety of agent templates and examples and provides a variety of two-dimensional agent environments. It allows users to dynamically access and modify agent properties, agent behavioral equations, and model properties at run time. Moreover, Repast includes libraries of genetic algorithms, neural networks, random number generation and specialized mathematics, as well as supports for social network modeling. ABDiSE has a narrower focus. It specifically aims to make modeling the causes and dynamics of natural disasters conceptually intuitive without loss of rigor and fidelity, and the model base extensible. It adopts some of the ideas from Repast, including dividing disaster events into smaller natural elements. ABDiSE also allows users to access and modify agent and environment properties dynamically at run time.

AnyLogic [23] is a multi-method simulation modeling tool developed by the AnyLogic Company for modeling and simulating supply chains and logistics; passenger flows in airports and subway stations; population, housing and transport infrastructure of a city, and so on. Models can dynamically read and write data to spreadsheets or databases during a simulation run, as well as charting model output dynamically. In addition to agent-based modeling, AnyLogic also supports discrete event simulation and system dynamics. AnyLogic simulation consists of stock and flow diagrams and process flowcharts. ABDiSE does not have these capabilities. Rather it focuses on simulation of natural disasters: The framework offers easy to use tools for agent model extensions, simulation state saver/loader, and run-time editing of agent and environment properties.

Natural Disaster Complex Systems Framework [24] provides tools for modeling and simulation of natural disasters. In addition to elements for modeling natural disasters, these tools also have elements for modeling and defining organizational structures and the policies that must be taken into account to simulate real-life emergency management activities. Similar to this framework, ABDiSE is also structured to facilitate simulation and observation at different levels of abstraction and details the development of natural disasters. The current versions of ABDiSE do not support the modeling of organizational structures and policies, however. We plan to add to ABDiSE models of standard operating procedures (SOPs) and human beings to enable the effectiveness of SOPs in simulated disaster scenarios to be evaluated via simulation [25] in the near future.

3 MODEL ELEMENTS

This section presents an overview of the ABDiSE model elements. As stated earlier, they capture common-sense concepts in natural disasters. The goal is for disaster scenario models built from the elements to be conceptually intuitive, easy to understand and use. The model elements are implemented as C# classes. Subsequent sections on architecture and implementation of agents and the simulation engine will provide further details.

3.1 Simulation World, Environment and God

Throughout any simulation experiment setup and carried out within ABDiSE, there is one and only one simulation world. In the narrowest sense, the term *world* refers to the geographical area specified by the user for the experiment at set up time. The world may be divided into regions, each of which has a specified boundary. For sake of concreteness without loss of generality, our discussion assumes that agent interactions are in a grid topology [17, 18] except for where it is stated otherwise.

Environments The simulation world has one or more environments. Each environment is defined by a set of parameters for one or more regions. In general, the values of environment parameters are functions of space and time. At each point in space and time, the values of environment parameters specify attributes / conditions that affect the behaviors of all agents around that point in space and time. The user can choose to provide the parameter values for all or selected grid points within the world (or a region) and instants or intervals of time. The environment parameters can also be defined by functions of space and time. For example, at time t and a point (i.e., a location) with longitude *log*, latitude *lat*, and altitude *alt*, the cumulative rain fall R, wind speed W, and wind direction D are given by functions $R = F(log, lat, alt, t)$, $W = G(log, lat, alt, t)$, and $D = H(log, lat, alt, t)$, respectively.

As in real-life, the simulation world has a global environment. Some regions may have local environments that differ from the global environment. In a region with a local environment, the behavior of every agent depends on the local environment.

God Agent The model for every simulation experiment has one and only one special agent called `God`. As we will soon see that through this agent, ABDiSE provides the user with capabilities control over all model elements during each simulation experiment. Conceptually, `God` can create, and no other agent has this capability. The simulation world, environments and agents running during the experiment are set up (i.e., created and initialized) by this agent.

During any experiment, `God` has the records of all environments and agents in the simulation world. To simulate a disaster scenario, the user starts from creating the instance of `God` agent. This step is in fact automatically completed by ABDiSE GUI: After the GUI pops up, `God` of the simulation world is ready to carry out user's wishes as specified by the user via the GUI.

Figure 2 lists `create` and other capabilities of `God`. It is often convenient to create some agents from the start but keep them inactive until some later time instant(s) or upon the occurrence of some conditions. The update method of an agent is executed during the current simulation step only if the agent is active at the time. The `Activate` method is for this purpose. Using `Affect` and `Control`, the user can alter the simulated scenario in ways that cannot be easily or conveniently defined by behavior specifications and interaction rules of individual agents or changes in environments. Examples include a sudden raise in flow volume and ambient temperature to simulate the effect of an upstream dam break and an explosion nearby, respectively.

`Create`:	This is the capability to create and initialize agents and define simulation world and environments in the world.
`Activate:`	This method activates specified agents.
`Affect:`	This method changes environment parameters and agent attributes in arbitrary ways, including non-causal ways.
`Control`:	This method causes an agent to change behavior/state in arbitrary ways, including ways not defined by the behavior-change methods of the agent.

Figure 2 Core capabilities/methods of `God` agent

3.2 Agents

Hereafter, when there is no possibility of confusion, we will use the term agent and agent instance interchangeably. As stated earlier, an agent is an active object. It can interact with the environment and with other agents. Similar to agents in existing agent-based models, agents in ABDiSE interact with each other during simulation according to rules governing their interactions under various conditions. All entities that interact in the simulation experiment can be modeled as agents. (Even markers in Map Explorer represent agents.) So, the user cannot simulate any disaster scenario without creating or importing one or more agents.

Major Types and Agent Properties In ABDiSE, every agent other than `God` belongs to one of two major types: `NaturalElementAgentType` and `AttachableObjectAgentType`. Agents of natural element types include cinder, smoke, fluid and so on. Disasters are typically caused by this type of agents. Agents of attachable type are affected by disasters. Examples include

tables, buildings, trees and cars. Disasters are due to their interactions with some of natural element agents. Figure 3 lists some of the essential properties of each agent instance in ABDiSE. Of which major type an agent instance belongs is indicated by its flags `IsNaturalElementAgent` and `IsAttachableObjectAgent`. Clearly, only one of these flags can be true. We will return to explain other properties shortly.

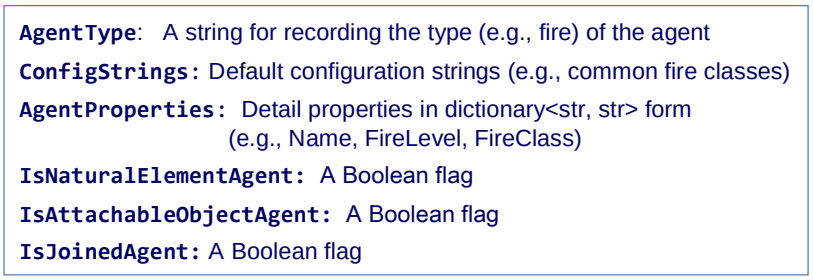

Figure 3 Essential properties of agent

Attach Method `Attach` is one of agent methods/rules. The `Attach` rule is applicable to a natural element agent and an attachable object agent (or a joined agent). The result produced by `Attach` is a *joined agent*. The natural element agent, or the attachable agent, or both may disappear, and the new joined agent instance thus created inherits their attributes and status. Depending on the type of the joined agent, the agent may also have its own methods/rules. The `Attach` rule enables us to model events that cause disasters and the development of disasters. As far as we know, no other agent-based model has this rule.

An example is a cinder, which is an agent instance of the natural element type, attaches itself to a table, which is an attachable object. The behavior of the table-joined-with-cinder agent changes according to a method of the table agent used to model and simulate the start of a fire from the table and subsequent development of the fire. Another example is water (a fluid) attaches to a mud-sand mixture. Mud-sand with water attached becomes debris that flows according to a debris flow law or as computed by a debris-flow simulator when given the amount of water attached. In a forest fire scenario, a cinder attaches to a tree. A new tree-joined-with-cinder joined agent instance is created. The behavior of the joined agent is computed by a wild fire simulator.

Joined Agent Again**,** a joined agent instance is composed of a natural element agent instance and an attachable agent instance or a joined agent instance. The type of a joined agent is decided by the two agents involved in the attach method. For example, fire attaches to tree creates tree-joined-with-fire joined agent instance. For convenience, we denote the joined agent by tree@fire. (In other words, we replace “joined with” by the symbol @.) Another

example is building@smoke, which means building-joined-with-smoke. The fact that an agent is a joined agent is indicated by the fact that the value of its `IsJoinedAgent` flag is true.

The user can customize `JoinedAgent` type according to the existing agent types. As examples, suppose that there are fire and smoke agents, which are of natural element agent type, and building and tree, which are attachable object agent type. Possible combinations include: building@fire, building@smoke, tree@fire, tree@smoke. The user can also customize rules and properties for each joined agent type by customizing the update method of the type. For example, if the user needs to use car@tornado in a scenario, he/she can create car agent and tornado agent and invokes the attach method of tornado.

4 ARCHITECTURE AND IMPLEMENTATION

ABDiSE is written in C#. It has two versions. Version 1.0 was implemented primarily for the proof-of-concept purpose. Version 2.0 was designed and implemented to provide extensibility: It allows the user to add new agent types and extend the model base without having to revise and recompile the entire framework and simulation engine.

4.1 Structure

Figure 4 shows the functional structure of both versions. In addition to libraries of reusable agents, agent-based models, various tools and simulators, the framework provides three major components: graphical user interface (GUI), experiment manager and simulation engine. Parts of the GUI was described earlier in Section 1: The user sets up, controls simulation experiments and visualizes simulation results via the GUI. The component is implemented using Microsoft Windows Form.

The Experiment Manager helps the user to build an agent-based model of the disaster and sets up the experiment. To assist the user in this process, the experiment manager provides select, load and build tools, which are accessible to the user via buttons and text boxes in Area *A* shown in Figure 1. The Experiment Manager also allows the user to change the fidelity of the model by adjusting simulation parameters such as spatial and time granularities and choices of simulators. The user can configure the engine by adjusting the number of threads used to run the simulation. The Model Builder loads agent data from agent library to create agent instances. It also loads models, tools, simulation information and facts from libraries and fact database.

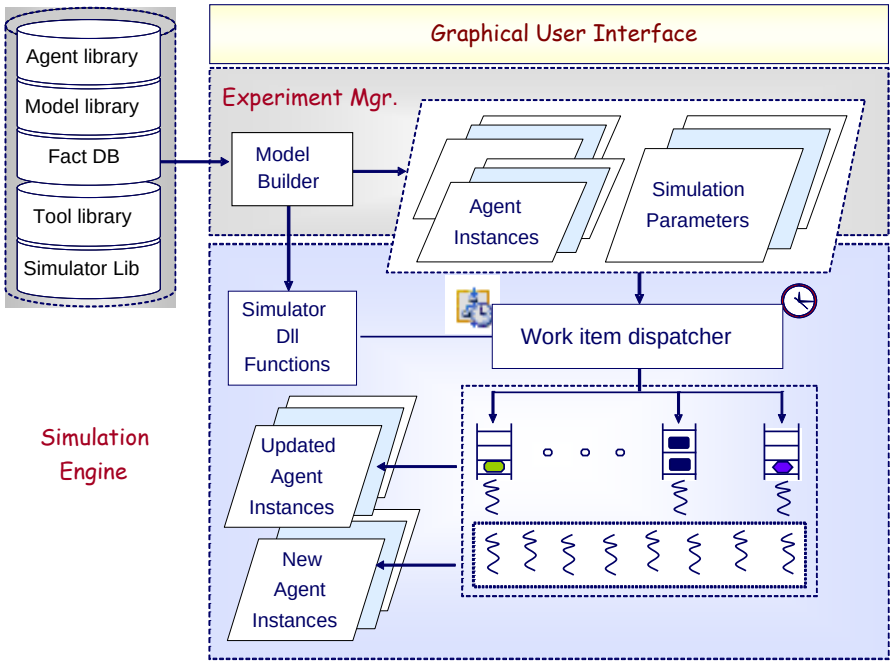

Figure 4 ABDiSE structure and major components

During a simulation experiment, the behavior rules of every activated agent instance are executed during each time step. The rules (i.e., simulation programs that compute the dynamic behavior of individual agents and joined agents) are programmed in `Update` methods of agent classes. They are wrapped within work items and invoked as DLL functions. The Simulation Engine uses a pool of worker threads to execute the work items. Specifically, at the start of each time step, the simulation engine checks the agent list of `God` for activated agents. It encapsulates the `Update` method of each activated agent into a work item, and queues the work item into the work item queue in the thread pool. Worker threads in the thread pool dequeue and execute work items. In this way, the states of agent instances and their interactions are updated. In the process, new agents may be created and existing ones may disappear according to their rules, all in time-driven manners.

4.2 Extensibility

In ABDiSE version 1.0, types of agents are fixed at compile time. Adding new agent types and behaviors requires the user to edit and recompile the code of the simulation environment. This shortcoming was removed in version 2.0 by providing the simulation engine with the ability to load agent types dynamically at initialization time. Specifically, in version 2.0, each agent type is defined by a class. The class inherits the abstract class `Agent` and implement methods that override abstract methods of `Agent`. Agent classes are compiled as DLL

functions. When ABDiSE 2.0 framework starts, the dynamic loader allows the user to selectively load available agent classes and associated .dll files from the agent library. The user can create new agent types by creating new agent classes for the types and compiling the new classes as DLL functions. Like existing agent classes, the new classes also inherit the abstract class `Agent`. Their .dll files are also loaded along with the .dll files of existing agent classes. The next section will provide further details on the mechanism.

In addition to enhancing ABDiSE with extensibility, we also restructured the code according to the Model-View-Controller (MVC) architectural pattern [26] in order to improve maintainability. Figure 5 shows how the major components of ABDiSE 2.0 fit in the MVC pattern. According to the pattern, classes are divided into three parts: `model`, `view` and `controller`. `Model` includes agents, joined agents, `God`, and environments. They directly manage data, logic and rules. Classes in `view` handle button events and passes user inputs to the controller. `Controller` manipulates data in the model. The `controller` also creates agents by dynamically loading DLL functions, based on rules of the `God` agent. The core controller uses a pool of worker threads to execute work items, as stated earlier. When the `model` is updated, the `view` is refreshed to display updated information.

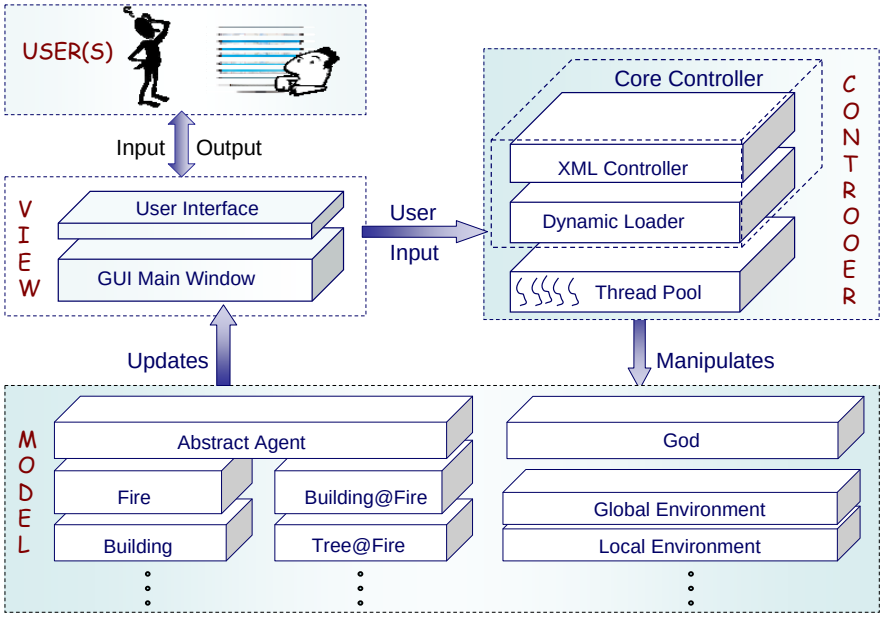

Figure 5 ABDiSE 2.0 major components

4.3 Important Classes

Again, ABDiSE is written in C#. Attributes of each agent instance are updated during simulation by the update method of the class implementing the agent. Documentation of ABDiSE was generated by Doxygen [27] from source code and comments.

The top-left part of Figure 6 lists important classes in ABDiSE 2.0. They use three namespaces: `Model, View` and `Controller`.

```
Partial List of ABDiSE 2.0 classes
      ABDiSE.Program
      ABDiSE.Model.Definitions
      ABDiSE.Model.Environment
      ABDiSE.Model.God
      ABDiSE.Model.AgentClasses.Agent
      ABDiSE.Model.AgentClasses.ConfigStrings
      …
      ABDiSE.Model.AgentClasses.SubTypeStrings
      ABDiSE.Model.AgentClasses.Building
      ABDiSE.Model.AgentClasses.BuildingJoinedFire
      ABDiSE.Model.AgentClasses.Fire
      ABDiSE.Model.AgentClasses.Smoke
      …
      ABDiSE.Controller.CoreController
      ABDiSE.Controller.ThreadPool.SimpleThreadPool
      ABDiSE.Controller.ThreadPool.SimpleThreadPool.WorkItem
      ABDiSE.View.MainWindow
      ABDiSE.View.SelectDLLForm
      ABDiSE.View.GMapMarkerCircle

public class God {
   public int CurrentStep = 0;
   public Environment[ ] WorldEnvironmentList;
   public Agent[ ] WorldAgentList;
   public God () {
        this.WorldEnvironmentList =
                    new Environment[MaximumEnvironments];
        this.WorldAgentList = new Agent[MaximumAgents];
    }
   public MethodReturnResultsAddToEnvironmentList
                                              (Environment en)
   public void ClearDeadAgent()
   public MethodReturnResults AddToAgentList
                                              (Agent targetAgent)
   public MethodReturnResults CheckAgentAttachment
                                              (Agent targetAgent)
    ...
   private MethodReturnResults activate (Agent target)
   private MethodReturnResults affect
                    (Agent target, Dictionary<string, string> controls)
   private MethodReturnResults control
                    (Agent target, Dictionary<string, string> controls)
}
```

```
static class Program
{
   static void Main( ) {
        CoreController CoreController = new CoreController();
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new MainWindow(CoreController));
   }
}

public abstract class Agent {
   public string AgentType;
   public CoreController CoreController;
   public int CurrentStep = -1;
   public Dictionary<string, string> AgentProperties;
   public ConfigStrings ConfigStrings;
   public bool IsNaturalElementAgent = false;
   public bool IsAttachableObjectAgent = false;
   public bool IsJoinedAgent = false;
   public bool IsDead = true;
   public bool IsActivated = false;
   public Environment MyEnvironment;
   public GMapMarkerCircle Marker;
   public PointLatLng LatLng;
   private Object agentLock = new Object();
   public Agent(
        CoreController CoreController,
        Dictionary<string, string> Properties,
        PointLatLng LatLng,
        ABDiSE.Model.Environment AgentEnvironment
    );
    public void ThreadPoolCallback(Object threadContext);
    public abstract MethodReturnResults Attach(Agent B);
    public abstract void SetMarkerFormat();
    public abstract ConfigStrings SetDefaultConfigStrings();
    public abstract void Update();
    public MethodReturnResults AgentDistance(Agent target);
    public MethodReturnResults MoveByWind();
}
public class ConfigStrings {
     public string ClassShortName;
     public string ClassFullName;
     public List<SubTypeStrings> SubTypes;
     public List<string> Keys;
}
```

Figure 6 Parts of ABDiSE classes and class `Program`, `God` and `Agent`

Program, Definition and Environment The simplest among all classes is `ABDiSE.Program`. As one can see from its definition in the upper-right part of Figure 6, the class has only one method, `Main`. It is the thread function of the main and only thread when the ABDiSE framework starts. As the result of executing the method, an instance of `CoreController` and an instance of `MainWindow` are created. In the process of creating the `MainWindow` instance, `Application.Run` initializes the C# Windows Form.

`ABDiSE.Model.Definitions` class includes the necessary definitions of constant data types in ABDiSE. For example, this class defines basic movement distances of agents and possible distances within which a natural element agent may attach to an attachable agent or joined agent. The class also defines enumerations such as `MethodReturnResults`, which record results returned by ABDiSE methods.

As stated in Section 1, an environment is defined by a set of parameters that specify attributes / conditions of the simulation world (or a region) affecting the behaviors of all agents in the world (or in the region). The class `ABDiSE.Model.Environment` provides the data structures of these parameters. Each agent is affected by only one environment, i.e., the environment of the smallest region that contains the location of the agent. Each environment has a list for recording references / pointers of agents that are in its own region. The reference to the environment is also kept by agents affected by it. These pointers accelerate the search between agents and environments.

Specifically, the data structure of environment parameters is the `dictionary<string, string>` named `EnvProperties`. After ABDiSE framework starts and the instance of `CoreController` is created, the controller automatically creates an instance of global environment with parameters specified in the data structure `Environment`. Currently, the elements of the structure include the `avgAltitude, rainFall, windSpeed, windDirection, temperature, and population, and weather`.

God Class The left part of Figure 6 lists parts of the `God` class. The class implements the one and only one special agent instance of `God` of the simulation world. As a special agent, `God` has four abilities: create, activate, affect and control. We can see from Figure 6, however, that `God` class does not have a `create` method. The reason is that for sake of extensibility and maintainability of the code, creating and initializing simulation world is done by `ABDiSE.Model.Program` in version 2.0. In this way, the simulation world is created automatically when ABDiSE framework starts. Agent creation is handled in turn by `CoreController.` The user can have agents or joined agents created one at a time by entering values of each agent instance via the GUI, which passes the values to `CoreController`. The dynamic loader in `CoreController` then uses the parameters to create a new instance of the specified agent type by loading the associated .dll file.

In contrast to create, `activate, affect` and `control` are methods of `God` class. Each agent has a Boolean flag: `IsActivated`. An agent is active, and its update method is executed during the current time step, only when the simulation engine finds its `IsActivated` flag being true at the start of the time step. So, applying `activate` method to a specified agent instance simply turns on the `IsActivated` flag of the agent instance.

Both of `affect` and `control` methods can change attributes of agent instances and environment. The user can edit the values of environment properties using text boxes in Area

E of the GUI. Thus directed by the user, these methods are invoked to carry out the changes. In the current version, the GUI does not provide function for editing attributes of agents.

Class `God` keeps track of all environments and agent instances of the simulation world. `WorldAgentList` and `WorldEnvironmentList` are for this purpose. The user can call `AddToEnvironmentList` and `AddToAgentList` methods to record a specified environment or agent instance in the respective list.

Each agent instance has a `IsDead` Boolean flag, which when true indicates that the instance no longer exists, i.e., is dead. `ClearDeadAgent` is a bookkeeping method of `God`. It is invoked at the end of each time step to delete all dead agent instances. Finally, the `CheckAgentAttachment` method of `God` can help an agent instance to check whether there are other agent instances suitable for attachment.

Agent Class We can see from Figure 6 that `ABDiSE.Model.AgentClasses.Agent` is an abstract class. Every agent type (e.g., fire, flood, and building) and every joined agent type (e.g., tree@fire and building@flood) is defined by a class that inherits from this class. The major type of an agent instance is indicated by the values of its Boolean flags: The difference between an agent instance and a joined agent instance is the values of their Boolean flag `IsJoinedAgent`. Similarly, the values of the Boolean flags `IsNaturalElementAgent` and `IsAttachableObjectAgent` of an agent instance indicate to which one of two major agent types (i.e., natural element type or attachable type) the instance belongs.

During simulation, each agent instance records its properties and environment at its location (i.e., at latitude and longitude `LatLng`) in the attribute `CurrentStep`. The string attribute `AgentType` is used to record agent type of the instance (e.g., `AgentType` = "`Flood`"). The create agent instance method has as an input parameter a `dictionary<string key, string value>`. The data structure is used to record `AgentProperties`. At the time of its creation, values of `key` and `value` of the initial dictionary is editable in area *A* of `MainWindow.`

The abstract methods `SetDefaultConfigStrings`, `SetMarkerFormat`, `Update` and `Attach` need to be overridden and implemented by methods of children classes for specific types of agents. Take `Fire` agent type as an example. The child fire agent type class `ABDiSE.Model.AgentClasses.Fire` defines the specific behavior rules of fire agents in the `Update` method of the class. Class `Fire` also has the related fire-building and fire-tree simulators in `Attach` method. The user can customize the image, color, size and shape of map

markers for fire agent instances shown in GUI by editing the `Fire.SetMarkerFormat`.

The `SetDefaultConfigStrings` method is used to save data of fire agent attributes. The data structure is described in class `ConfigString`, shown at bottom-right part of Figure 6. The data in this method are filled in Area **A** of the GUI automatically. The user can define attributes of different types of fire disasters in this method.

Similarly, classes for other model elements such as `Cinder, Smoke, Water, Building, Tree, Building@Fire, Building@Water` and so on inherit the abstract class `Agent`. They use the same namespace `ABDiSE.Model.AgentClasses`. The differences among all the agent classes and agent instances are given by values of `AgentType`, `AgentProperties`, Boolean flags `IsJoinedAgent`, `IsNaturalElementAgent`, and `IsAttachableObjectAgent` in addition to the rules in the four methods.

Finally, `ThreadPoolCallback` method and `agentLock` object is used in multi-threaded simulation. The next section will describe how they are used.

More on Update and Attach `Update` and `Attach` are core methods of `Agent` class. In general, properties of each agent instance change through time (e.g., fire intensity decreases and smoke moves with wind). The behavior rules dedicating the changes are defined by the `Update` method that is executed during each simulation step. When the simulation engine is multi-threaded, the method is wrapped into a work item, and the work item is inserted in the `Workitem` queue and executed by a worker thread. When the engine is single-threaded, the method is executed by the main thread.

`Attach` method defines the rules for creating new joined agents. This method is always executed by main thread. At the start of each time step, each activated natural element agent instance (again, take a fire for example) calls `God.CheckAgentAttachment(theFireAgent)` to check whether any of the attachable object agent instances nearby are suitable for attachment. Then the fire agent instance executes the `Attach` method `Fire.Attach`. The method gets one suitable agent instance at a time from the return value of `AgentDistance (oneNearbyAgent)`: When `AgentDistance` returns as an acceptable result a target agent instance, the `Attach` method checks the type of the target agent. As an example, suppose that the target agent is of a tree type. Because the rule for “fire attaches to tree” exists and is legal, the `Attach` method calls the dynamic loader in `CoreController` with properties of fire and tree agents as parameters to create an instance of `Tree@Fire`. On the other hand, if the `Attach` finds no rule for `Fire` attach to the target agent or properties of the specific agent

instances are such that the existing rule cannot be applied, the method moves on to get the next target agent until all suitable agents found during the current time step have been processed.

5 CORE CONTROLLER AND SIMMULATION ENGINE

Figure 7 shows parts of the code of `ABDiSE.Controller.CoreController` and `CoreController` constructor. During initialization of ABDiSE 2.0, the constructor of `CoreController` creates the one and only one instance of `God`; initializes the XML file controller; creates instance(s) of `Model.Environment` with parameters provided by the user via GUI; and launches the dynamic loader; and when the user commands to create agents via the GUI, uses the dynamic loader to load agent instances (i.e., the associated .dll files) from the agent library dynamically.

```
public class CoreController  {
    public God God;
    public SimpleThreadPool STP;
    /// Configuration strings from agent types in DLL.
    public List<ConfigStrings> ConfigStrings;
    /// All loaded DLL classes
    public ArrayList Classes;
    /// Types of agent classes: for xml save/load
    public Type[ ] AllTypes;
     /// XML save/load functions
    public XMLController XMLController;
    public CoreController ( )
    public ArrayList GetAllTypesFromDLLstring (string dllName)
    public ArrayList GetAllTypesFromClass (string dllName,
                                           string className)
    public object RunClass (string dllName, string className,
                            string methodName)
    public object CreateDLLInstance (string className,
                                     params object[] args)
    public void EnableMarkerAnimation()
    public void DisableMarkerAnimation()
    public void DeselectMarkers()
    public void StartThreadPool (int ThreadsNum,  int IdleTimeout,
                                 int ExecuteTime)
}
```

```
public CoreController ()
{
    this.God = new God();
    this.XMLController = new XMLController(this);

    ABDiSE.Model.Environment SimulationWorld =
                        new ABDiSE.Model.Environment ( … );
    God.AddToEnvironmentList (SimulationWorld);
    string DLLName = "AgentDLL";

    Classes = GetAllTypesFromDLLstring (DLLName);
    ConfigStrings = new List<ConfigStrings>();

    for (int ii = 0; ii < Classes.Count; ii++)
    {
        ConfigStrings configStr = (ConfigStrings)RunClass (
                                    DLLName,
                                    Classes[ii].
                                    ToString (),
                                    "SetDefaultConfigStrings“ );
        ConfigStrings.Add(configStr);

    }
}
```

Figure 7 Parts of class `CoreController` and `CoreController` constructors.

For sake of concreteness, we assume hereafter that the user has chosen via text boxes in area D of the `MainWindow` to run the multi-threaded simulation engine. As the next step during start up, the `CoreController` creates the thread pool of the simulation engine with user specified number of threads. `ABDiSE.Controller.ThreadPool.SimpleThreadPool` is a custom thread pool. It includes a pool of worker thread(s), a FIFO queue for holding work items waiting to be executed, and methods of controlling worker thread(s). Details on the thread pool and the `WorkItem` classes and key parts of their code can be found in [28].

During simulation, `CoreController` interacts with all models elements, including agents,

environments, and `God` and the GUI, while the simulation engine uses the worker threads in the thread pool to execute work items. Again, the dynamic behavior of individual agents and joined agents are defined by update methods of their respective types, and update methods of all activated agent instances in the simulation world are wrapped within the work items, invoked as DLL functions and executed by worker threads during a simulation step.

Several parts of the update method need to be marked as critical sections and guarded by locks. An example is the code for agent attachment: When the `attach` method returns successfully, the attachment will affect the agent instances being attached. Both of them must be locked until the attachment operation ends. Similarly, `God.WorldAgentList` must be locked by agent creation code when adding a new agent instance to the list, and each agent instance must be locked during its state update.

Lock contention is reduced and time order is preserved by dividing each simulation step into two parts. The second part starts after the first part completes. In the first part, the update methods of all activated joined agent instances are executed. The update method of each joined agent updates its state and may create new agent instances. The update methods of all agent instances that are not of joined agent types are executed in the second part of the time step. As stated earlier, the method first tries to find suitable agent instances nearby and attach to them if possible. When attachment of an agent instance succeeds, there is no need to update the state of the agent instance. In other words, the state of any agent instance is updated only when the instance does not attach to another agent instance. At the end of the second part of a step, when update methods of all agent instances have completed, the God instance executes `ClearDeadAgent` method to remove all dead agent instances.

6 SUMMARY AND FUTURE WORK

The previous sections described the agent-based models and tools provided by the agent-based disaster simulation environment ABDiSE for construction and execution of the agent-based models to simulate several common types of natural disasters. Models available in the agent library of the latest version, ABDiSE 2.0, includes agent classes/DLL functions for modeling fire, smoke, flood, tornado, buildings, trees, and joined agents of them. In addition to model elements, the framework provides the user with several easy-to-use tools, including tools for agent creation, environment control, simulation control, and an interactive GIS map, which the user can access via `MainWindow` of the GUI. `XMLController` is for

saving/loading the state of agents in the simulation world at any point during a simulation run to be used as a checkpoint and as an initial state of simulation experiments to be done later

In case studies for purposes of assessing the usefulness and usability of the model elements and tools, we constructed from available model elements in the agent library several customized simulators of disasters at locations of well-known past disasters, including a simulator of the devastating flood at Xiaolin Village in Taiwan caused by Typhoon Morakat in 2009 [29]; a simulator of an earthquake-triggered fire along Van Ness Avenue, San Francisco, CA, in the midst of the region hard hit by the 1906 San Francisco earthquake [30] and a simulator of frequent bushfires in New South Wales, Australia [31]. The screen shot shown in Figure 1 was taken during the early part of a simulation run of the flood simulator. (Again, the small squares and circles in Area B represent houses/buildings in the affected area.) Similarly, Figure 8 shows the screen shots generated by the San Francisco fire simulator on the left and the South Wales bushfire simulator on the right showing the quick spread of fire (in red) and smoke (in dark cloud shapes) from a few buildings and trees.

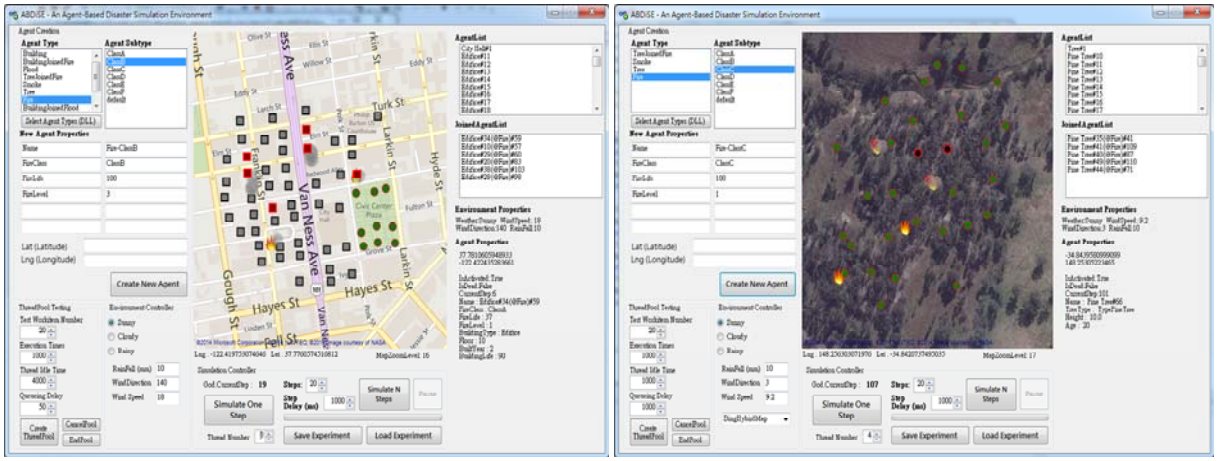

Figure 8 Screen shots generated by fire simulators
Left: Fire developing at a few buildings along Van Ness Avenue in San Francisco
Right: Bushfire spreading from a few trees)

Thus far, we have focused primarily on ABDiSE models and tools using which a user with can easily construct customized simulators of different scenarios for diverse types of disasters at different locales. Our experimentation has demonstrated that we have achieved this goal. The accuracy of the simulators requires significant improvement, however. This can be done by incorporating accurate and detailed fire and flood simulation programs in the update methods of joined agents as discussed in Section 2. Indeed, the most important future work is to provide a wide selection of accurate simulation programs for various disasters.

ABDiSE 2.0 not only allows the user to import external simulators but also to add new agent types without having to modify and compile the source code of the framework: It will be straightforward to add agent types needed to model scenarios such as infectious diseases, terrorist attacks and major traffic accidents. It will also be straightforward to add agents that model people in different roles during disasters and thus make ABDiSE applicable for modeling and simulating SOPs in response to common types of disasters [25].

The source code of ABDiSE 2.0 is released under the GPL license. The concept of agent-based model supported by the framework and ABDiSE code are easy-to-understand. A user willing to revise the source code can enhance the framework along many directions. For example, in current version, the `God.CheckAgentAttachment()` method is invoked by every activated natural element agent during each simulation step. The method confines the search for agent instances suitable for attachment among attachable object agent instances. Consequently, multi-level attachment is not supported. (In other words, it is not possible for joined agents to attach to another agent and other joined agent.) This limitation can be easily removed by modifying this method and behavior rules in attach methods to include joined agent instances as attachable objects without changing ABDiSE 2.0 system architecture.

ACKNOWLEDGEMENT

This work was supported by the Taiwan Academia Sinica thematic project OpenISDM (Open Information Systems for Disaster Management).

REFERENCES

[1] E. Bonabeau, “Agent-based modeling: methods and techniques for simulating human systems”, Proceedings of National Academy of Science,, May 14, 2002.

[2] J. M. Epstein, et al., “Combining computational fluid dynamics and agent-based modeling: a new approach to evacuation planning,” Plos ONE, 6(5), May 2011

[3] D. Helberg and S. Balictti, “How to do agent-based simulation in the future: from modeling social mechanism to emergent phenomena, to interactive system design,” SFI Working paper No. 11-06-024, 2011.

[4] S. Bandini, et al., “Agent-based modeling and simulation: an informatics perspective,” Journal of Artificial Societies and Social Simulation, Vol.12, No. 4, 2009.

[5] N. Gilbert, Agent-Based Models, Sage Publications, 2007.

[6] “Comparision of agent-based modeling software,” at

http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software

[7] “GMap.NET homepage,” at http://greatmaps.codeplex.com/

[8] “Fire simulation with FDS” , at http://ejrh.wordpress.com/2011/04/14/fire-simulation-with-fds/

[9] “Fire Dynamics Simulator (FDS) and Smokeview (SMV),” https://code.google.com/p/fds-smv/

[10] Harrington Group, “Fire Dynamics Simulator (FDS) and Smokeview (SMV) – Bringing Fire Analysis to Life”, at http://www.hgi-fire.com/blog/fire-dynamics-simulator-fds-and-smokeview-smv-bringing-fire-analysis-to-life

[11] Michael Sebhat and Tom Heinzer, “The Development of an ArcInfo Interface to the National Weather Service DAMBRK Model” http://proceedings.esri.com/library/userconf/proc97/proc97/to600/pap581/p581.htm

[12] MIKE21 User Guide and Reference Manual, Danish Hydraulic Institute, 1996

[13] Paul Bates and Dr. Mark Trigg, et al., “Amazon modeling,” http://www.bris.ac.uk/geography/research/hydrology/research/flooding/amazon/

[14] LISFLOOD-FP, http://www.bris.ac.uk/geography/research/hydrology/models/lisflood

[15] Leszek Pawlowicz, “High-Resolution Sea Level Rise Flooding Animations in Google Earth”, http://freegeographytools.com/2007/high-resolution-sea-level-rise-flooding-animations-in-google-earth

[16] “Animated Sea Level Rise in Manhattan”, https://www.youtube.com/watch?v=RUNsV0ofX-s

[17] Charles M. Macal and Michael J. North, “Introduction to Agent-based Modeling and Simulation”, MCS LANS Informal Seminar, November 29, 2006, at http://www.mcs.anl.gov/~leyffer/listn/slides-06/MacalNorth.pdf

[18] Charles M. Macal and Michael J. North, “Introductory Tutorial: Agent-Based Modeling and Simulation”, Proceedings of the 2011 Winter Simulation Conference.

[19] Nicholas R. Jennings and Stefan Bussmann, “Agent-Based Control Systems – Why Are They Suited to Engineering Complex Systems”, IEEE Control Systems Magazine, June 2003

[20] M.Wooldridge, “Agent-based software engineering,” Proc. Inst. Elec. Eng., vol. 144, pp.

26-37, 1997.

[21] Rob Allan. “Survey of Agent Based Modeling and Simulation Tools”, at http://www.grids.ac.uk/Complex/ABMS/

[22] “Repast”, at SourceForge http://repast.sourceforge.net/repast_3/index.html and http://en.wikipedia.org/wiki/Repast_(modeling_toolkit)

[23] “AnyLogic official website”, at http://www.anylogic.com/

[24] Karam Mustaphaa, Hamid Mcheicka, Sehl Melloulib, “Modeling and Simulation Agent-Based of Natural Disaster Complex Systems”, EUSPN-2013, Procedia Computer Science 21, 148-155, 2013.

[25] C. Y. Wu, “Approaches to Model People and SOP in Disaster Scenarios,” MS thesis, Department of Computer Science, National Tsing Hua University, Taiwan, April 2015.

[26] Model–view–controller architectural pattern, at ABDiSE - Agent-Based Disaster Simulation Environment.doc

[27] Doxygen (tool) at http://www.stack.nl/~dimitri/doxygen/

[28] T. L. Hsu, “An Agent-Based Disaster Simulation Environment,” MS thesis, Department of Computer Science, National Tsing-Hua University, Taiwan, December 2014

[29] Typhoon Morakot, at http://en.wikipedia.org/wiki/Typhoon_Morakot

[30] 1906 San Francisco earthquake, http://en.wikipedia.org/wiki/1906_San_Francisco_earthquake

[31] 2013 New South Wales bushfires”, http://en.wikipedia.org/wiki/2013_New_South_Wales_bushfires