Simulation of an Organization of Spatial Intelligent Agents in the Visual C#.NET Framework

Reza Nourjou and Michinori Hatayama

***Abstract*—Problem: This paper investigates the simulation of a community of spatial intelligent agents, an important issue for the implementation and evaluation of distributed algorithms. The research question is how to simulate this organization using Visual C#.NET.**

Objective: This paper intends to use the C# programming language to develop a simulator. In a simulated environment, a social agent needs to communicate with other agents and respond to received messages.

Method: Our approach is to integrate the thread and delegate methods provided by the .NET framework. We enhance the architecture of a spatial intelligent agent in order to embed several in a simulated society. A basic C# program is presented to demonstrate how to implement a contract net protocol (CNP) among three simple agents.

Results: Two results were achieved: 1) the responses and actions of three agents during a simulated CNP and 2) a simulated environment that contains distributed spatial intelligent agents that can interact with each other and a human user.

Conclusion: The proposed methodology and presented code provide a flexible and efficient framework for C# developers to develop, simulate, and evaluate a society of advanced software agents using the .NET platform.

***Index Terms*—simulator, C#.NET, multi-agent systems, community/society, software agents, development, communication, code, message, GIS.**

I. Introduction

GICoordinator, a GIS-based intelligent assistant system designed for disaster emergency response, helps an IC (Incident Commander) to coordinate a team of field units, especially in urban search and rescue operations [1], [2]. This spatial intelligent system supports the IC with intelligent algorithms for action planning and task scheduling to centrally coordinate a team in a dynamic and spatial environment [3] , [4]. In addition, it uses a spatial database to manage geographic and location-based information and GIS functions to support the development of these spatial intelligent algorithms [5]. The C# programming language was used to implement the core of the GICoordinator [2]. The IC is equipped with a computer that runs GICoordinator, and a simple version of this agent has been developed for field units [6].

A difficult challenge arises in this domain when there are several teams, each with their own IC. In order to maximize a joint objective, the teams have to cooperate and coordinate their actions, and this is the main responsibility of ICs. As a result, distributed ICs form an organization or society and must coordinate their decisions with each other in a decentralized way. With regard to this community, the GICoordinator of an IC should provide two essential capabilities: 1) to communicate and interact with other agents to share data and coordinate decisions, 2) to use decentralized algorithms to coordinate distributed decisions. A human and GICoordinator is called a human-agent team in this paper, and Fig. 1 illustrates an organization in which three human-agent teams are embedded. Each IC is equipped with a GICoordinator that is run on a tablet computer. Distributed instances of GICoordinator can communicate with each other via message (data) passing.

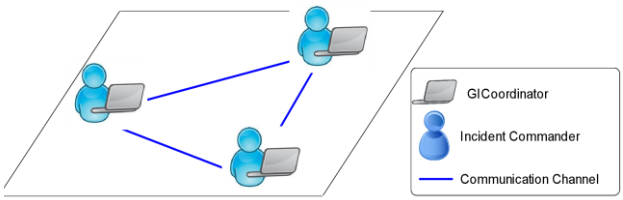

Fig. 1. An organization of three human-agent teams.

We require a proper framework to enable us to implement and evaluate the two key capabilities of the GICoordinator before real world applications can be developed. Hence, it is necessary to simulate a society of advanced agents. Building this framework is important because it enables us to easily run several GICoordinators within it, observe their behaviors, and test and refine the GICoordinator. When developing this framework, we must also consider that a single GICoordinator has already been developed.

Agent-based modelling (ABM) and simulation is a relatively new approach to modelling systems composed of autonomous, interacting agents [7]. The simulation of an organization of GICoordinators is important for developing and evaluating their organizational performance. A simulator provides us with an efficient framework in which we can run a number of GICoordinators in a virtual world, study their interactions and behaviors when solving distributed optimization problems, and test the efficiency of distributed coordination algorithms. It enables us to easily refine their architecture and re-design and re-implement better algorithms.

In intelligent systems technologies, the .NET framework can be used to develop agents such as GICoordinator. In this paper, we investigate how to build a simulator using the C# programming language on the .NET framework. Our requirements (and assumptions) are as follows:

Manuscript received December 2, 2013; revised February 17, 2014.

R. Nourjou is with the Graduate School of Informatics, Kyoto University, Japan (e-mail: nourjour@imdr.dpri.kyoto-u.ac.jp).

M. Hatayama is with the Disaster Prevention Research Institute (DPRI), Kyoto University, Japan (e-mail: hatayama@imdr.dpri.kyoto-u.ac.jp).

- communication between agents is done by sending and receiving messages
- the simulator and embedded agents are run on the same computer
- each agent has access to a central spatial database
- a human user can select any agent and interact with it
- an interaction between an agent and human user is done via the agent's interface
- a description of the distributed coordination algorithms is beyond the scope of this paper
- we can initiate and run any number of agents
- there are at least three agents
- all agents have GIS functions
- C# .NET is used to create this simulator
- an agent needs to be able to send a specific message to a specific agent
- an agent needs to appropriately respond to received messages

In order to build the simulator, we need to apply an efficient tool to achieve the requirements given these assumptions. Although there is much literature on agent-based modeling software, toolkits, and programming languages, unfortunately they do not thoroughly address the problems stated in this paper.

This paper presents a methodology for the development of a simulator for an organization of distributed GICoordinators using C# .NET according to the requirements defined above. In this simulated society, a GICoordinator can communicate with other agents, respond to received messages, and interact with a human user.

II. Background

Agent-based modelling and simulation (ABMS) is a relatively new approach to modelling complex systems composed of interacting, autonomous "agents." Agents have behaviors, often described by simple rules, and interactions with other agents that in turn influence their behaviors. ABMS research can be traced back to investigations into complex systems. A typical agent-based model has three elements: 1) a set of agents, 2) a set of agent relationships and methods of interaction, and 3) the agents' environment. A developer must identify, model, and program these elements to create an agent-based model [7].

In general, two types of simulation/modelling systems are available to develop agent-based models: toolkits and software [8]. Toolkits are simulation/modelling systems that provide a conceptual framework for organizing and designing agent-based models. They provide appropriate software libraries that include pre-defined routines and functions specifically designed for agent-based modelling. In addition, the object-oriented paradigm allows the integration of additional functionality not provided by the toolkit. Some toolkits include Swarm, MASON, Repast, OBEUS, and AnyLogic.

In addition to toolkits, software such as StarLogo, NetLogo, and OBEUS are available for developing agent-based models, and can simplify the implementation process. For example, using simulation/modelling software often avoids the need to develop an agent-based model via a low-level programming language (e.g., Java or C++). In particular, software for ABM is useful for the rapid development of prototype models. However, modelers using software are restricted to the design framework implemented by the software. For instance, some ABM software only has limited environments (e.g., raster only) in which to model, or agent neighborhoods may be restricted in size. Furthermore, a modeler is constrained to the functionality provided by the software (unlike in ABM toolkits, modelers are unable to extend or integrate additional tools), especially if the toolkit is written in its own programming language (e.g., NetLogo). Most simulation packages claim they are object-oriented or use Java as development language.

An agent-based model could be programmed completely from scratch using a low-level programming language such as Python, Java, or C. However, development from scratch can be prohibitively expensive, given that this requires the development of many services already provided by specialized agent modelling tools. Most large-scale agent-based models use specialized tools, toolkits, or development environments because of their usability, ease of learning, cross-platform compatibility, and the need for sophisticated database connection capabilities, graphical user interfaces, and GIS [7]. In particular, the use of toolkits can reduce the burden modelers face when programming the parts of a simulation that are not content-specific [8].

III. Methodology

Fig. 2 presents the structure of the simulation based on the requirements defined in this paper. Our focus is on simulating the organization in the .NET framework. To implement this, we integrated the thread and the delegate methods included in the C# .NET framework. These technologies enable us to easily refine the architecture of the GICoordinator to achieve our purpose.

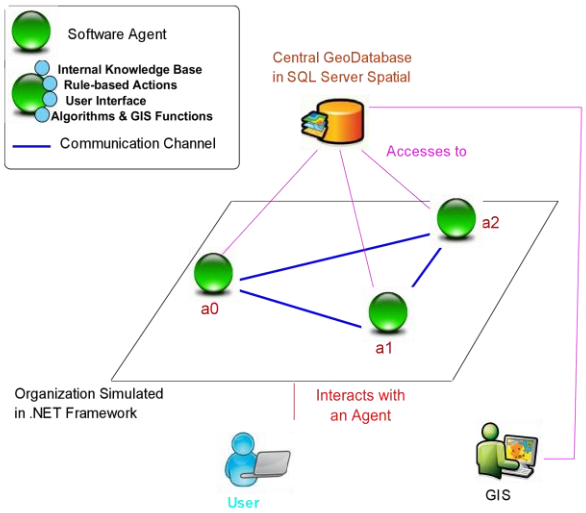

Fig. 2. Structure of the simulation based on the requirements defined in this paper

A delegate is a type that defines a method signature. When a delegate is instigated, its instance can be associated with any method with a compatible signature. The method can be

invoked (or called) through the delegate instance. Delegates are used to pass methods as arguments to other methods. Event handlers are simply methods that are invoked through delegates. A custom method can be created and a class such as a Windows control can call this method when a certain event occurs. Threading enables the C# program to perform concurrent processing so that more than one operation may be done at a time [9].

C# supports the parallel execution of code through multithreading. A thread is an independent execution path, able to run simultaneously with other threads. Hence, it is possible to write applications that perform multiple tasks at the same time. Tasks with the potential of holding up other tasks can execute on separate threads, a process known as multithreading or free threading [9].

To demonstrate this, a basic program was developed in C# to show how to implement a contract-net protocol (CNP) among three simple agents in a simulated organization (given in Listing 1). CNP is a task-sharing protocol in multi-agent systems consisting of a collection of nodes or software agents that form the "contract net." When a node receives a composite task (or for any reason cannot solve its present task) it breaks the problem down into sub-tasks and announces the sub-task to the contract net that acts as a manager. Bids are then received from potential contractors and the winning contractor(s) are awarded the job(s) [10].

The role of agent "a0" is to announce an auction. Because the roles of "a1" and "a2" are to bid for this auction, they do not need to communicate with each other. Hence, we have not defined (established) communication between these two agents. This makes our code clear and easily understood.

```
using System;
using System.Threading;
using System.Collections.Generic;

namespace SIMULATOR
{
    delegate void d_Send_f0t1(message msg);
    delegate void d_Send_f0t2(message msg);
    delegate void d_Send_f1t0(message msg);
    delegate void d_Send_f2t0(message msg);

    class Simulation
    {
        static void Main(string[] args)
        {
            Agent0 a0 = new Agent0("a0");
            Agent1 a1 = new Agent1("a1");
            Agent2 a2 = new Agent2("a2");

            a0.f_Send_t1 = new
d_Send_f0t1(a1.f_Recieve_f0);
            a0.f_Send_t2 = new
d_Send_f0t2(a2.f_Recieve_f0);

            a1.f_Send_t0 = new
d_Send_f1t0(a0.f_Recieve_f1);

            a2.f_Send_t0 = new
d_Send_f2t0(a0.f_Recieve_f2);

            Thread T_a0 = new Thread(a0.Run); T_a0.Start();
            Thread T_a1 = new Thread(a1.Run); T_a1.Start();
            Thread T_a2 = new Thread(a2.Run); T_a2.Start();

            Thread.Sleep(30000);
        }
    }
    class Agent0
    {
        public string agentId;
        public int count;
        private List<message> Messages;
        public d_Send_f0t1 f_Send_t1;
        public d_Send_f0t2 f_Send_t2;

        public Agent0(string id) { agentId = id; }
        public void Run()
        {
            Thread T0 = new Thread(f_ReactiveRules);
            T0.Start();

            message msg = new message
            {
                from = agentId,
                to = "ALL",
                subject = "Announcement-a-contract",
            };
            count = 0;
            Messages = new List<message>();
            var T1 = new Thread(() => f_Send_t1(msg));
T1.Start();
            var T2 = new Thread(() => f_Send_t2(msg));
T2.Start();
        }
        public void f_Recieve_f1(message msg)
        {
            Console.WriteLine(agentId + ": f= " + msg.from +
" :Bid= " + msg.content);
            Messages.Add(msg);
            count = count + 1;
        }
        public void f_Recieve_f2(message msg)
        {
            Thread.Sleep(1000);
            Console.WriteLine(agentId + ": f= " + msg.from +
" :Bid= " + msg.content);
            Messages.Add(msg);
            count = count + 1;
        }
        private void f_ReactiveRules()
        {
            /* Monitor the enviroment and Do something for
Example*/
            while (true)
            {
                if (count != 2) { Thread.Sleep(1000);
continue; }

                List<message> Winners = new List<message>();

                if (Messages[0].content >
Messages[1].content)
                { Winners.Add(Messages[0]); }
                else if (Messages[0].content <
Messages[1].content)
                { Winners.Add(Messages[1]); }
                else if (Messages[0].content ==
Messages[1].content)
                { Winners.Add(Messages[0]);
Winners.Add(Messages[1]); }

                message award = new message
                {
                    from = agentId,
                    subject = "Award",
                };
                foreach (message winner in Winners)
                {
                    if (winner.from == "a1")
                    {
                        var T1 = new Thread(() =>
f_Send_t1(award)); T1.Start();
                    }
                    if (winner.from == "a2")
                    {
                        var T2 = new Thread(() =>
f_Send_t2(award)); T2.Start();
                    }
                }
                break;
            }
        }
    }
    class Agent1
    {
        public string agentId;
        public d_Send_f1t0 f_Send_t0;

        public Agent1(string id) { agentId = id; }
        public void Run()
        {
            /* Do something */
        }
        public void f_Recieve_f0(message msg)
        {
```

```
            Thread.Sleep(1000);
            Console.WriteLine(agentId + ": f= " + msg.from +
" :Sub= " + msg.subject);

            if (msg.subject == "Announcement-a-contract")
            {
                message msg2 = new message
                {
                    from = agentId,
                    to = "a0",
                    subject = "Bidding",
                    content = (new Random()).Next(1, 3)
                };
                var T1 = new Thread(() => f_Send_t0(msg2));
T1.Start();
            }
            if (msg.subject == "Award")
            {
                Console.WriteLine(agentId + ":: A Nice
Day");
            }
        }
    }
    class Agent2
    {
        public string agentId;
        public d_Send_f2t0 f_Send_t0;

        public Agent2(string id) { agentId = id; }
        public void Run()
        {

        }
        public void f_Recieve_f0(message msg)
        {
            Thread.Sleep(1000);
            Console.WriteLine(agentId + ": f= " + msg.from +
" :Sub= " + msg.subject);

            if (msg.subject == "Announcement-a-contract")
            {
                message msg2 = new message
                {
                    from = agentId,
                    to = "a0",
                    subject = "Bidding",
                    content = (new Random()).Next(1, 3)
                };
                var T1 = new Thread(() => f_Send_t0(msg2));
T1.Start();
            }
            if (msg.subject == "Award")
            {
                Console.WriteLine(agentId + ":: A Nice
Day");
            }
        }
    }
    class message
    {
        public string from { get; set; }
        public string to { get; set; }
        public string subject { get; set; }
        public int content { get; set; }
    }
}
```

Listing 1. Visual C# .NET Code for the Implementation of a CNP in the Simulated Organization of Three Simple Social Agents.

To establish a communication network among three agents, four delegates were declared. As Fig. 3 illustrates, there are two methods for setting up a communication network among agents.

1) An agent can communicate with another agent via an independent delegate. The network of GICoordinator instances is decentralized and the number of nodes is fixed, similar to a peer-to-peer network.
2) There is a shared delegate that enables all other agents to send their message to this agent.

The following code shows a delegate declaration that is used by agent a1 to send its messages to agent a0.

```
delegate void d_Send_f1t0(message msg);
```

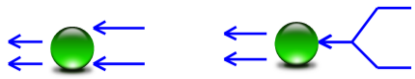

Fig. 3. Two possibilities for configuration of a communication network.

Communication is used by agents to share or exchange data and information. To do this, the class "message" is declared to encode three kinds of data. A number of sophisticated classes may be designed for sending different types of data.

To present agents, three classes were declared. The architecture of an agent includes: 1) several delegates for sending its messages to others, 2) several functions for receiving messages from others, and 3) other properties and methods. The three agents were then created.

To setup a communication channel between two agents, e.g., between agents a0 and a1, the following code is necessary. It states that a0 can send a message to a1 if it calls the function "f_Send_t1" and a1 receives a message from a0 through the function "f_Recieve_f0" In fact, a0 executes a method of a1.

```
a0.f_Send_t1 = new d_Send_f0t1(a1.f_Recieve_f0);
```

To create three autonomous agents, the newly created agents are run in three threads. The following code shows a thread that has been created for the "run" method of a0:

```
Thread T_a0 = new Thread(a0.Run);
T_a0.Start();
```

A proper delegate should be run by an agent to send a message to a certain agent. The agent a0 executes the following delegate in order to send a message to the agent a1.

```
var T1 = new Thread(() => f_Send_t1(msg));
T1.Start();
```

The function "f_ReactiveRules" of a0 enables this agent to autonomously monitor and scan the environment. In this code, this function continuously scans the total number of bids. If this agent receives two proposals, a simple sub-algorithm is run to select the winner(s).

IV. Results

Two results were achieved in this study. After running the code presented in Listing 1, we achieved the result shown in Fig. 4. It presents the responses and actions of three agents during a simulated CNP.

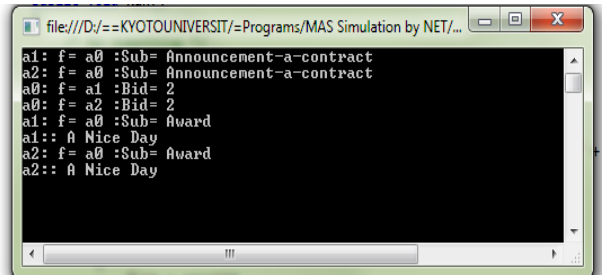

Fig. 4. The result of running the code presented in Listing 1

Fig. 5 presents the second result, a simulator that runs

multiple instances of GICoordinator. We enhanced the code shown in Listing 1 to achieve the the structure shown in Fig. 2 and requirements stated in Section I. Each GICoordinator has its own user interface that allows the human user to select and interact with it. Agents send messages containing data and information that are required by agents to coordinate their actions and decisions. In addition, a geographic information system (ArcGIS [11]) provides geographic information managed by a geodatabase [12].

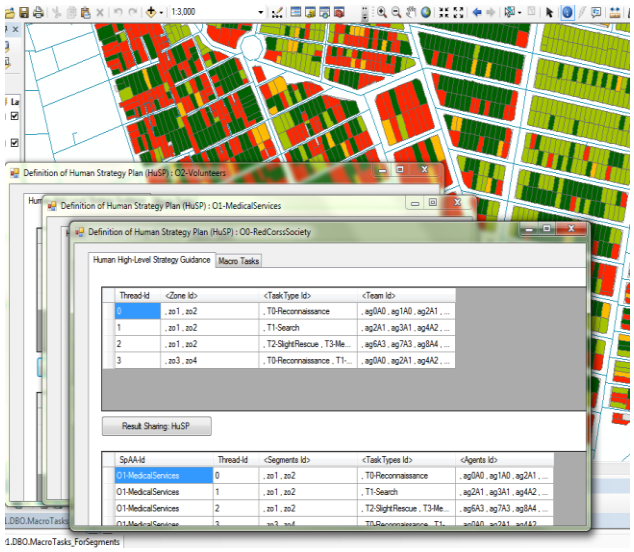

Fig. 5. The simulator that runs three instances of GICoordinator with their user-interface and the GIS that is used by human

V. Conclusion

This paper described a simulator that runs multiple instances of an agent, the GICoordinator that was developed in earlier work by the authors. The goal was to simulate the operation of multiple agents that are distributed. The agents are addressable and communicate through message passing implemented with C# delegate methods. The thread methods in C# allow for the parallelization of agent operations and their communications. Demonstration code of the contract-net protocol (CNP) was also presented.

This paper presented C# code that can simulate a virtual society of distributed agents using the .NET framework. This basic code demonstrated how to implement a communication network among social agents. This code can be modified by other software engineers and C# developers to satisfy their requirements.

The proposed methodology provides a flexible framework to develop, simulate, and evaluate multi-agent systems on the .NET platform. It enables work on decentralized algorithms for distributed GICoordinators by embedding a number of these agents in a simulated environment and demonstrating their actions.

For mutual communication between two agents, we dedicated two delegates such that each agent has its own communication channels. It is possible to specify a delegate for an agent and share it so that other agents can send messages through the delegate (or communication channel). Multithreading methods allow us to execute several agents and actions simultaneously.

It remains a difficult challenge for a software engineer to develop and run a simulator with many agent instances. All communication challenges among agents must be specified and defined.

Future work will be to design and develop distributed algorithms to coordinate distributed decisions among ICs in disaster emergency response operations. This simulated framework will be an essential tool for this research.

Acknowledgment

R. Nourjou is grateful for the financial support of GCOE-HSE of Kyoto University, which enabled him to be a visiting scholar at the Information Sciences Institute of University of Southern California and the Robotics Institute of the Carnegie Mellon University from December 2011 to November 2012.

References

[1] R. Nourjou, M. Hatayama, and H. Tatano, "Design requirements of spatial intelligent coordinators for incident commanders," in *Proc. the 11th International Conference on Information Systems for Crisis Response and Management*, 2014.

[2] R. Nourjou, M. Hatayama, S. F. Smith, A. Sadeghi, and P. Szekely, "Design of a GIS-based assistant software agent for the incident commander to coordinate emergency response operations," Cornell University Library, 2014.

[3] R. Nourjou, S. F. Smith, M. Hatayama, and P. Szekely, "Intelligent algorithm for assignment of agents to human strategy in centralized multi-agent coordination," *Journal of Software*, 2014.

[4] R. Nourjou, S. F. Smith, M. Hatayama, N. Okada, and P. Szekely. "Dynamic assignment of geospatial-temporal macro tasks to agents under human strategic decisions for centralized scheduling in multi-agent systems," *International Journal of Machine Learning and Computing (IJMLC)*, vol. 4, no. 1, pp. 39-46, 2014.

[5] R. Nourjou, P. Szekely, M. Hatayama, G.-A. Mohsen, and S. F. Smith, "Data model of the strategic action planning and scheduling problem in a disaster response team," *Journal of Disaster Research*, vol. 9, no. 3, pp. 381-399, 2014.

[6] R. Nourjou, M. Hatayama, and H. Tatano, "Introduction to spatially distributed intelligent assistant agents for coordination of human-agent teams' actions," in *Proc. 2011 IEEE International Symposium on in Safety, Security, and Rescue Robotics (SSRR)*, IEEE, 2011, pp. 251-258.

[7] C. M. Macal and M. J. North, "Tutorial on agent-based modelling and simulation," *Journal of Simulation 4*, no. 3, pp. 151-162, 2010.

[8] C. J. E. Castle and A. T. Crooks, "Principles and concepts of agent-based modelling for developing geospatial simulations," CASA Working Paper 110, Centre for Advanced Spatial Analysis (UCL), London, UK, 2006.

[9] Microsoft, Visual C#. [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/kx37x362.aspx

[10] R. Davis and R. G. Smith, "Negotiation as a metaphor for distributed problem solving," *Artificial intelligence 20*, no. 1, pp. 63-109, 1983.

[11] T. Ormsby, "Getting to know ArcGIS desktop: Basics of Arcview, ArcEditor, and ArcInfo: Exercise data," ESRI Press, 2001.

[12] A. MacDonald, "Building a geodatabase," ESRI press, 2001.

Reza Nourjou was born in Urmia, Iran in 1979. He received the B.Sc. degree in geomatics engineering and the M.Sc. degree in geographic information systems (GIS) from the K.N.Toosi University of Technology, Iran, in 2002 and 2007, respectively. In 2006, he joined the Risk Management Research Center, International Institute of Earthquake Engineering and Seismology (IIEES), Tehran, Iran as a GIS expert and a research assistant. Since 2010, he has been a Ph.D. candidate in graduate school of Informatics, Kyoto University, Japan.

His current research interests include multi-agent systems, automated planning and scheduling, intelligent agents, game theory, artificial intelligence, distributed algorithms, and GIS. He is a member of Association for the Advancement of Artificial Intelligence.

He was the recipient of the first prize in the GIS competition ranked first

in the national student competition, Iran in 2001. He received the Japanese Government Scholarship from October 2009 to March 2013 for the Ph.D. program. He was a visiting scholar at the Information Sciences Institute of the University of Southern California and the Robotics Institute of Carnegie Mellon University from November 2011 to November 2012.

Michinori Hatayama was born in Osaka, Japan in 1968. He received the B.S. and M.S. degrees in control engineering from Osaka University in 1992 and 1994 respectively, and the Ph.D. degree in computational intelligence and systems science from Tokyo Institute of Technology in 2000. In 2002, he became an assistant professor and from 2004 is an associate professor in Disaster Prevention Research Institute, Kyoto University, Kyoto, Japan.

His research carrier in disaster risk management area began from the Great Hanshin-Awaji Earthquake in Kobe in 1995. After that, his group developed an original temporal GIS named DiMSIS and applied it to disaster response activity and disaster risk management in some local governments and regional communities. This software was used in real sites by officials in Kobe (1995), Turkey (1999 and 2000), Niigata (2004) and Tohoku (2011).

His current research interests include ICT based disaster risk assessment system, disaster response system, rescue activity support, and disaster planning support system.