Simple Run-Time Infrastructure (SRTI):

An Accessible Distributed Computing Platform for Interdisciplinary Simulation

Szu-Yun Lin[1], Andrew W. Hlynka[2], Lichao Xu[3], Hao Lu[4], Omar A. Sediek[5], Sherif El-Tawil[6], Vineet R. Kamat[7], Jason McCormick[8], Carol C. Menassa[9], Seymour M.J. Spence[10], Atul Prakash[11], Benigno Aguirre[12]

[1]*Assistant Professor, Department of Civil Engineering, National Taiwan University, Taipei, Taiwan; email: szuyunlin@ntu.edu.tw; corresponding author*

[2]*Research Area Specialist Senior, Office of Research, University of Michigan, Ann Arbor, MI 48109, USA; email: ahlynka@umich.edu*

[3]*Ph.D., Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: lichaox@umich.edu*

[4]*Ph.D. Student, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA; email: harveylu@umich.edu*

[5]*Ph.D., Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: osediek@umich.edu*

[6]*Professor, Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: eltawil@umich.edu*

[7]*Professor, Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: vkamat@umich.edu*

[8]*Associate Professor, Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: jpmccorm@umich.edu*

[9]*Associate Professor, Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: menassa@umich.edu*

[10]*Associate Professor, Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI 48109, USA; email: smjs@umich.edu*

[11]*Professor, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, USA; email: aprakash@umich.edu*

[12]*Emeritus Professor, Department of Sociology and Criminal Justice, University of Delaware, Newark, DE 19716, USA; email: aguirre@udel.edu*

1. ABSTRACT

Distributed computing is a necessity for conducting cross-disciplinary research where field-specific computational models (simulators) are available, but have not been designed to work together. An example of this is natural hazards research. Simulators abound in the disparate fields that fall under this area, e.g. social science, engineering, economics, and health, but little progress has been made to integrate the simulators to study overarching and cross-disciplinary disaster scenarios. The reason for slow penetration of this technology is the high barrier to entry, which requires extensive knowledge of computer science and programming. Building upon an existing platform named Simple Run Time Infrastructure (SRTI v1), a new, fundamentally different version (SRTI v2) is developed to address the issues mentioned above. Designed to provide a low barrier to use, SRTI v2 is developed for users with limited programming experience and designed to simplify and streamline a user's effort to compose a distributed simulation and handle time management. To achieve this primary objective, pre-compiled components are provided including the RTI Management Server, RTI Wrapper, and a GUI. By exploiting these pre-compiled components, users can compose a scalable distributed simulation with heterogeneous computational models. To demonstrate the concepts behind SRTI, a cross-language simulation, modified and extended from a time-dependent resilience analysis of an electric power system in the literature, is presented to show the scalability and usability of SRTI. Features of the different versions of SRTI are discussed and useful features to develop in the future are outlined.

2. INTRODUCTION

Distributed simulation links independent simulators that execute on the same and/or separate processors to conduct an overarching analysis of a large-scale simulation system comprised of the simulators [1]. A common characteristics of distributed simulations is that they help users cut across disparate disciplines, in which one or more of the simulators are rooted. The need for distributed simulation exists in research areas that are broadly multi-disciplinary, such as natural hazards and defense.

Distributed simulation techniques and tools have been under development for the past three decades. Most of the tools are based on the publish-subscribe principle for message transmission between simulators. One of the earliest tools is the Distributed Interactive Simulation (DIS) platform [2] developed by the US Department of Defense. High-Level Architecture (HLA) is a set of standards [3-5] for building distributed simulation tools. Unlike DIS, where simulators deal directly with one another, HLA relies on a central manager (middleware), called Run-Time Infrastructure (RTI), to handle message transmission. The Test and Training Enabling Architecture (TENA) is similar to HLA and also uses a middleware to organize message communication. Other techniques are Distributed Data Services (DDS) [6] and Lightweight Communication and Marshalling (LCM) [7]. The distributed simulation concept also has been applied locally in laboratories as an experimental testing technique, i.e. hybrid simulation. For example, OpenFresco (the Open-source Framework for Experimental Setup and Control) [8] is a software framework designed for connecting finite element models and the physical portion of structural/geotechnical systems to facilitate various experimental setups, control, and data acquisition. Some newer tools, such as Google Protocol Buffers [9], operate like LCM. All of these platforms require extensive user knowledge in order to operate them and therefore offer a high barrier to entry for non-specialized users.

To facilitate broader usage of distributed computing, especially by lay users, the intent of this paper is to describe fundamental modifications made to an existing free, open-source distributed simulation platform (SRTI v1) that substantially reduces the barrier for use. The open source software, which has some architectural similarities to HLA, was developed to investigate hazard mitigation and multidisciplinary interactions in community resilience, but it could be repurposed for other fields. The tool is available as a pre-compiled application for quick deployment and integration. Treated as a black box accessed through API (application programming interface) function calls, a user can edit the open-source code or rewrite components to support additional programming languages.

In the following sections, the history and features of SRTI are discussed first then the SRTI framework is provided from a high-level overview to the details of its constituent components. A cross-language resilience analysis of an electric power system subjected to an earthquake is used to showcase SRTI's user friendliness, scalability and flexibility. Finally, the unique features of SRTI and potential new ones that could be developed are discussed.

3. HISTORY OF SRTI

The concept generation and technical development of SRTI have been under development since 2016 [10]. The initial approach to the development of SRTI focused on a purely data-transmission system based on the JSON format (SRTI v1), but it did not apply mandatory rules on the simulators for time management [11]. While highly flexible for composing simulations, this approach required users to embed time-related information within the transmitted messages. This additional requirement made the barrier for use quite high, as users needed to be familiar with computer programming and the inner workings of the platform.

To address the additional time management requirements and enable a plug-and-play approach to composing simulations, a new version of the SRTI (v2) that is fundamentally different in its approach was developed. Unlike the earlier version of the SRTI that required users to add code into their simulators to allow them to access SRTI, the new version simply requires users to prepare a configuration file. This configuration file can be edited in any common ASCII text editor and provides the definition of the simulators, messages, and their publish/subscribe relationships. To further facilitate ease-of-use, an optional SRTI graphical user interface (GUI) was developed to assist the used in constructing these configuration files. By using the GUI, users can describe their simulator through an interactive graphic interface and export the necessary configuration files. It also allows users to launch and execute the RTI Management Server and simulators through the same interface. Moreover, the SRTI GUI helps users inspect the content of messages and time step during the execution of the simulation. Aside from lowering the barrier for use for lay users, the use of text-editable configuration files in the new version of SRTI allows users explicit control for time-dependent simulations for which time synchronization between multiple simulators and phase changes are required.

4. FRAMEWORK DESCRIPTION

SRTI v2 consists of a series of software tools that not only focus on data-transfer between computer programs, but also assist in the construction of simulation scenarios. In addition to data management, SRTI v2 was designed with the purpose of managing simulator actions and time synchronization between multiple simulators inside a larger complex simulation system, while maintaining user accessibility to easily replace or edit components of the system.

A high-level view of the architecture of the SRTI v2 is shown in Figure 1~~Figure 1~~. To facilitate referral to the software, the version (v2) will be dropped in subsequent discussions. In addition to the RTI Server and RTI Lib API that existed in SRTI v1, the pre-compiled components added in SRTI v2 include the RTI Management Server, RTI Wrapper, and SRTI GUI. SRTI uses the RTI Management Server as the central node of a connected network, by which multiple instances of RTI Wrapper are connected. The RTI Wrapper handles the connection between user's simulator and the RTI Management Server. The SRTI GUI assists users in defining the configuration file for each RTI Wrapper, and allows execution of the whole simulation system from a single place.

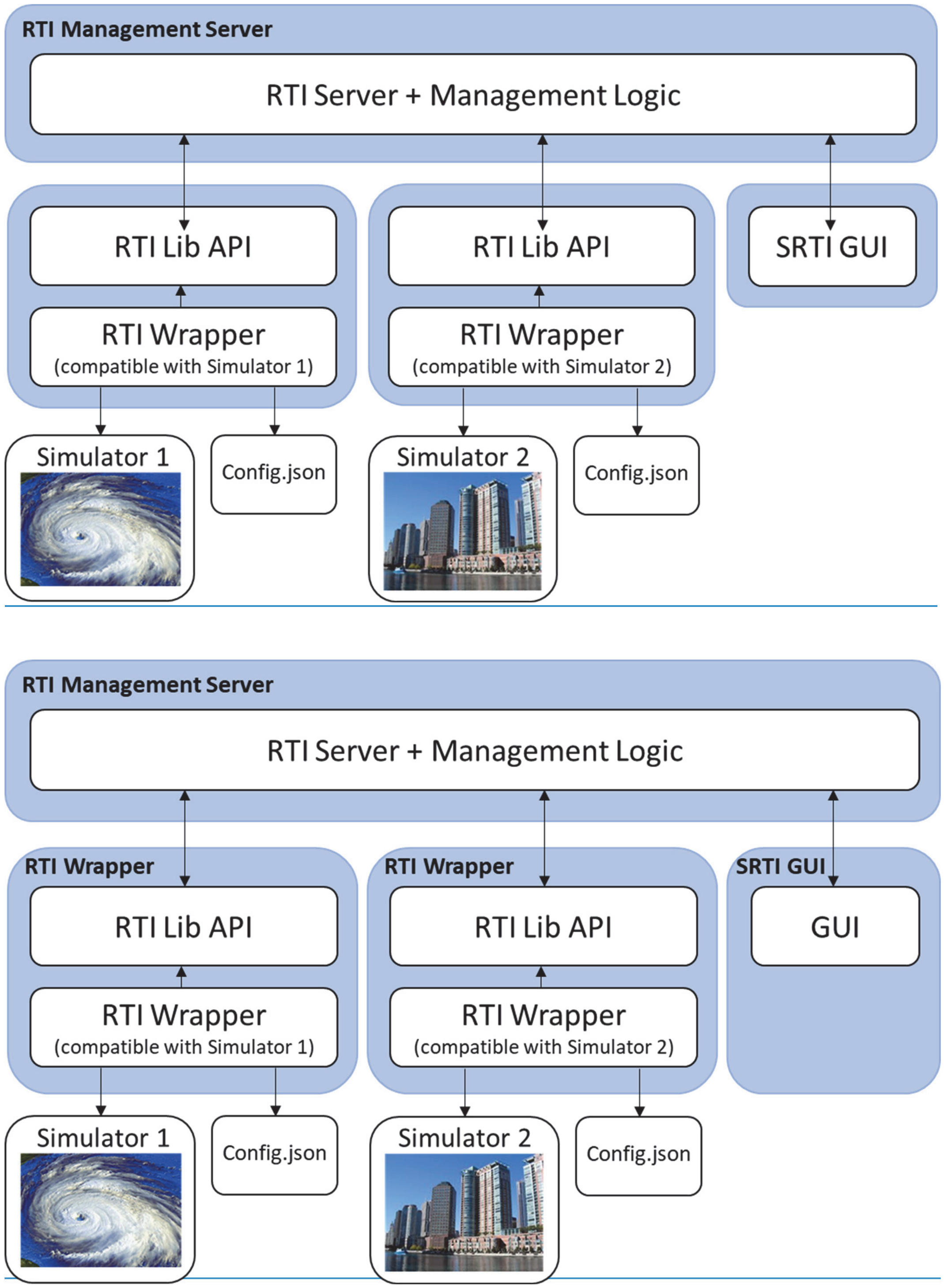


Figure 1. A high-level view of the architecture of the SRTI

RTI Management Server

Publish-subscribe data transmission pattern and message format

Connections between the RTI Management Server and the RTI Wrapper are made using sockets, allowing for connections both locally and over a network. By using a centralized 'server' instance, each simulator has a shared access point to connect to the larger simulation system, simplifying the method of initial connection. Messages are shared through a publish-subscribe model, as in the framework introduced by Lin et al. [12], where each connected simulator can choose which message(s) they should receive, by name. The RTI Management Server sends out a copy of any received message to all simulators subscribed to that message. Individual simulators can each publish one or more message types (differentiated by name). This design avoids direct integration of simulation models, which usually depends on the compatibility of programming languages and instead replaces that with a dependency on messages. It allows the user to remove or edit individual simulators from an existing simulation system, which is constructed via the SRTI (assuming that any significant messages are still sent by at least one simulator in the system).

Messages are formatted using standard JSON. Each message includes the following important key-value pairs: "name" (the name of the message), "content" (data the simulators share and receive), "source" (the name of the simulator it came from), "timestamp" (the computer system's time when the message was sent), and "vTimestamp" (the virtual timestep for the simulation system). These (and other components used to handle additional features internally through the SRTI system) are used by the RTI Management Server and the RTI Wrapper to manage the system and to convey data in the expected order. The RTI Wrapper helps prepare messages from each simulator to match this format, which is independent of the system platform and programming language of the simulators, allowing the users to focus on the data content itself. Several additional internal messages in this format are exclusively for the RTI Management Server and RTI Wrapper for managing simulator actions and time synchronization between multiple simulators.

Time synchronization and simulation step/stage control

The RTI Management Server includes added logic at the server level that helps manage features specific to virtual simulation. This logic was added based on needs associated with possible simulation system requirements. One example is the concern of time synchronization: if simulator logic is dependent on the concept of time, but one simulator executes faster than others do in the simulation system, how can they be kept synchronized? The RTI Management Server keeps track of an internal virtual timestep that is shared in each message. This value does not increase until the RTI Management Server receives a "finish" confirmation from every active simulator. When this occurs, the virtual timestep is increased by one, and the RTI Management Server sends a message request to each RTI Wrapper to "start" the next step. This process repeats, preventing any single simulator from proceeding to the next step too early, while allowing for some computational parallelization where the user deems it possible.

This type of synchronization permits some additional variance to be controlled at the RTI Management Server-level. For example, it allows simulators to run in order before others in a certain step. It also supports the communication between simulators with heterogeneous time steps (i.e. step intervals can vary from simulator to simulator). Moreover, it supports complex simulation systems with different stages during execution, where a different set of simulators may run at each stage and depend on the data from the previous stage. The RTI Management Server controls the activation and deactivation of the simulators at different stages. Control over the time step and simulation stage can be automated through the RTI Management Server by many controlling parameters, either defined inside a configuration file by users before the start of a system execution or inside subsequent messages during execution. The RTI Wrapper plays an important role in communicating these features to the RTI Management Server.

RTI Wrapper

The RTI Wrapper is an important part of SRTI. As discussed in [13], in most existing distributed simulation platforms and data passing tools, the user must modify the internal code of their simulator to connect to a larger system using specific instructions. While this feature remains in the SRTI to allow for customizability for experienced users, most users will utilize the RTI Wrapper. The RTI Wrapper handles both the connection to the RTI Management Server and to the user's simulator code itself. It accomplishes this task by reading and writing variable data from the simulator during execution and sharing it with the larger simulation system. In ideal situations, the simulator code does not need to be modified at all, and even the RTI Wrapper itself will not need to be edited or recompiled.

To accomplish its function, the RTI Wrapper primarily makes use of an object-oriented programming concept called "reflection" [14-16]. This concept allows one computer program to inspect public variables and functions inside another program and make use of them by name. It typically allows references with names defined at runtime, after compilation. Therefore, the RTI Wrapper can be pre-compiled and distributed online for users to simply download and run without compiling locally. The names of relevant variables and functions inside the user's simulator can be defined externally inside a configuration file and then read as input at the beginning of the RTI Wrapper's execution. The configuration file is saved in JSON format, and consists of a large set of parameters that can be defined by the user. A complete list of these parameters, including value type and definition, can be found in the documentation on the SRTI GitHub site [17].

There are advantages to using reflection in this solution. An RTI Wrapper is convenient to share with users, and ready to use with pre-compiled simulators. However, reflection across multiple languages is not a common feature, so unlike the RTI Management Server, the RTI Wrapper must be re-written in a new language for every computer language it needs to support. If new stability changes are made in the SRTI's continued development, all versions of the RTI Wrapper must be updated, which becomes an issue for development scalability. Currently, RTI Wrapper versions for Java, Python, Matlab and NetLogo [18] have been prepared for public access.

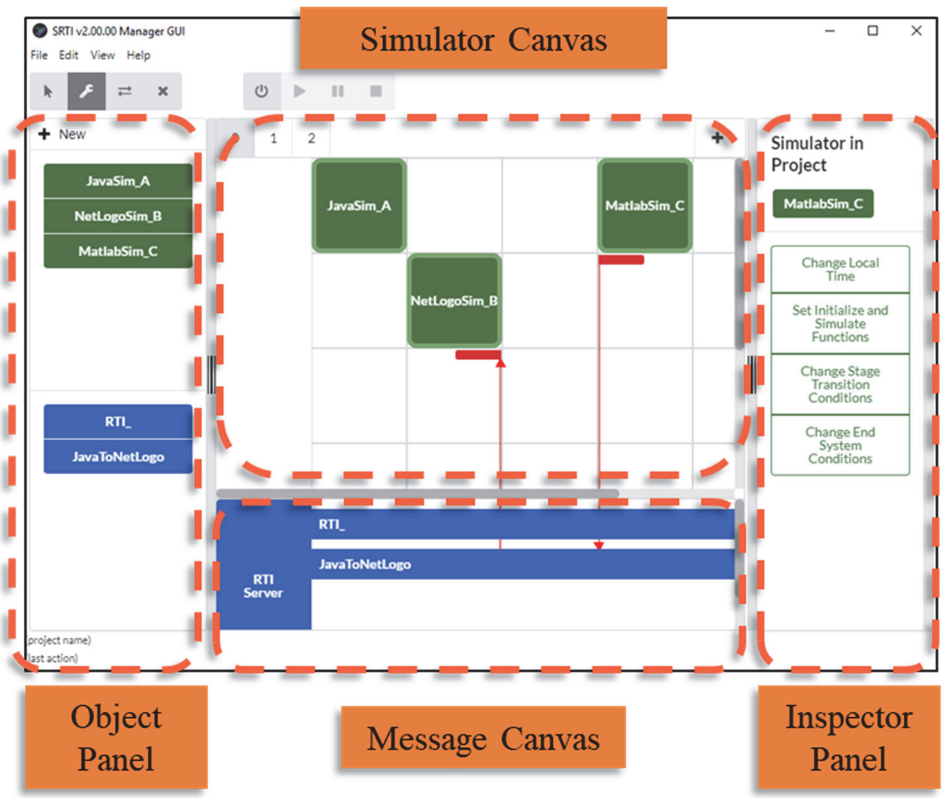
Additionally, some compiled languages do not support reflection, such as C, C++ and Fortran, making this approach difficult. Without reflection, an RTI Wrapper's source code would have to be modified to import and access a simulator's functions, meant to work exclusively with that one simulator. This prevents a pre-compiled version from being released for certain languages and is different from previous versions of SRTI, which allowed support for virtually any computer language.

Aside from reflection, the RTI Wrapper also assumes simulators are written with object-oriented design, or a similar alternative, such that specified variables and functions can be accessible. This design is not supported by all languages, or it may require simulators to be written in a specific manner to that users may not be accustomed. Also, if important variables and simulators are 'private' rather than 'public' (a generally good practice in computer programming), and if the simulator is closed-sourced (unable to be modified and recompiled), then it may not be able to connect with the SRTI without extreme adaptation by the end user.

The RTI Wrapper utilizes user-defined parameters, including (but not limited to) values to determine how to publish and subscribe to messages from the RTI Management Server, how the data corresponds to simulator variables, when to execute, and what simulator functions to execute at each virtual timestep. The configuration file can be edited by hand in a text editor or through the SRTI GUI. The RTI Wrapper can handle translating and sending simulator variables in the following formats: Boolean, integer, double (decimal-precision numbers), strings (text), and multi-dimensional arrays of any of the previously listed formats.

SRTI GUI

The last component for the SRTI is the GUI. The SRTI GUI (Figure 2~~Figure 2~~) has two primary functions: to help define the configuration file for each RTI Wrapper and to allow execution of the simulation system from a single place (within the GUI itself). Unlike the RTI Management Server and Wrapper, the SRTI GUI is optional. A simulation system's configuration files can be edited by hand, and individual RTI Wrapper instances (for each simulator) can be started manually. However, the SRTI GUI greatly improves the workflow.



(a)

(b)

Figure 2. Screenshot of a simple project inside the SRTI GUI: (a) overview; (b) important details

The GUI consists of an “Object Panel” on the left, a “Canvas” in the middle, and an “Inspector Panel” on the right. The Object Panel allows the user to define new simulators or messages to a given project, and then adds them to the Canvas to define how they will be executed. The Inspector Panel allows access to modify finer details for each individual simulator. Properties such as “publishing,” “subscribing” and “execution order” can be intuitively displayed with the canvas. Some features, such as describing a RTI Wrapper folder location, are defined through text-box input. The user is responsible to define the terminal command that will launch the Wrapper and allow simulator access for your corresponding language and operating system. First-time users will need to study the documentation of the SRTI [17] carefully to understand what properties exist and how to define them, with or without the GUI.

Project files for the GUI can be saved and reloaded. Individual simulators and messages in the project are copied and saved into individual files, allowing a user to import a previous simulator into a new project. A menu command allows for generating and outputting the expected RTI Wrapper configuration files in the corresponding file directories (same directory as each Wrapper and simulator). The user can choose to start the simulation system entirely from within the GUI, using “Play-mode” buttons at the top to launch the RTI Management Server (and each RTI Wrapper), start the simulation, pause and resume the simulation, or stop the simulation. Minimal feedback is provided in the GUI to show the system running, which simulator is currently in the process of finishing a step, and what recent messages were sent. Like the SRTI itself, distributed simulations across a local area network (LAN) or a wide area network (WAN) can be supported through the GUI, although extra steps are required to make an external computer accessible (or else, to connect its simulators manually to a launched RTI Management Server without support from the GUI). Generally, editing and running the system locally on one machine is recommended for using the GUI as an all-in-one system. The GUI can be used to edit and output configuration files, but the user would need to copy those files to desired locations on high-performance servers and execute them one-by-one, using a terminal command for each of them. Some minor changes to each configuration file might be necessary. Since the GUI is an optional component of SRTI, an experienced user can still overcome any limitations the GUI has for different systems.

While the RTI Management Server was written in the Java programming language, the GUI was written in JavaScript, using the open-source Electron framework to compile it into a desktop app. This GUI is not necessarily the only one that can or will be produced for the SRTI, either internally or by a third-party. The SRTI GUI is treated as an additional simulator from the perspective of the RTI Management Server, one that subscribes to all messages (at this time, a RTI Wrapper or RTI Lib API was not written for the JavaScript language, so the necessary socket connection is hard-coded into the GUI app). It is meant to serve as an example, with source code available for modification and with the ability for users to create their own GUI from scratch to replace it. This approach is in line with the philosophy of the SRTI: to be open-source, convenient to use and straightforward to extend, either through new apps or through support of new languages.

5. EXAMPLE: CROSS-LANGUAGE SIMULATION

In this section, a cross-platform simulation of an electric power system subjected to an earthquake is conducted using SRTI (v2). To demonstrate the scalability and flexibility of SRTI, the time-dependent analysis of three interdependent lifeline systems in

[19] has been simplified to consider the electric power system only (Figure 3~~Figure 3~~). A Visualization Simulator written in NetLogo has been added to show the ability of the SRTI to connect simulators across multiple languages. Figure 4~~Figure 4~~ shows the simulation framework of the example, where the simulators written in Matlab are adapted from the authors' previous work [19]

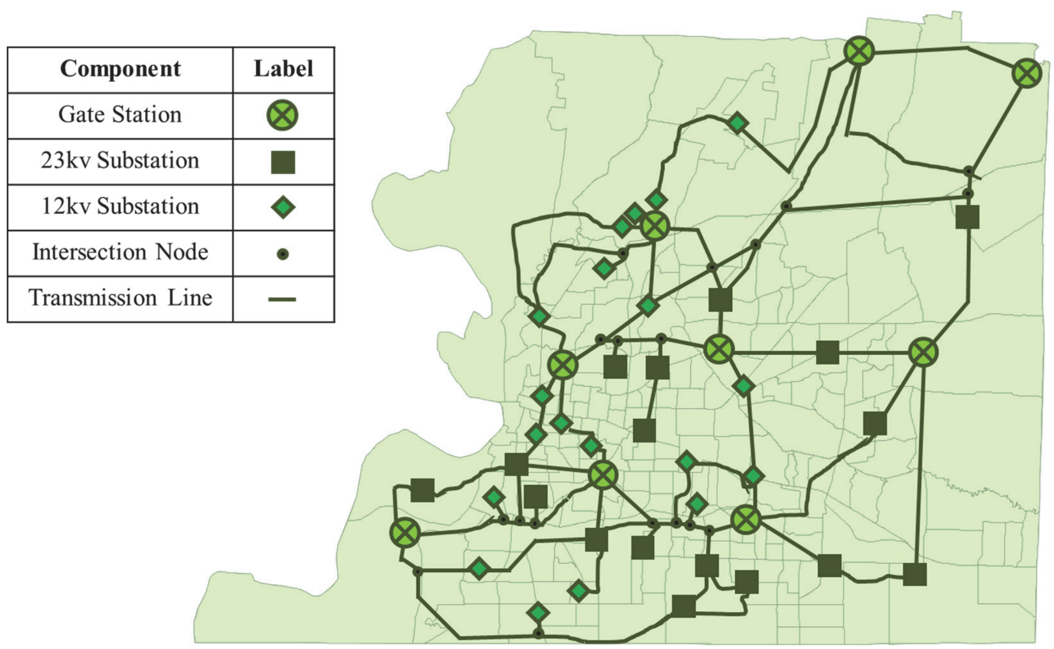


Figure 3. Electric power system in Shelby County, Tennessee {Adapted from [19].}

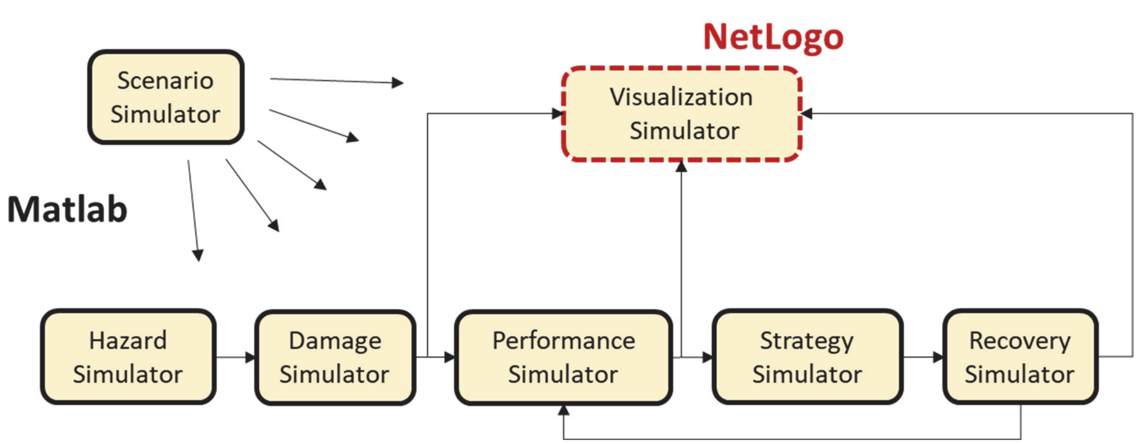


Figure 4. Simulation framework of cross-language example

Using the SRTI and GUI to conduct the simulation, the disaster event and post-disaster recovery effort are represented by seven simulators that may run at one or both of the disaster and recovery phases, named Stage 0 and Stage 1, as shown in Figure 5~~Figure 5~~ and Figure 6~~Figure 6~~. The time step of the disaster phase and recovery phase is taken as 0.01 second and one day, respectively.

Figure 5. Simulators running du Stage 0: disaster phase

Figure 6. Simulators running during Stage 1: recovery phase

During Stage 0 (i.e. disaster phase), the Scenario Simulator provides the coordinates and connectivity of the electric power system. The Hazard Simulator estimates ground motion magnitudes at the location of components, where the EQ2 case in [19] is adopted for considering an earthquake with a 5% probability of exceedance in 50 years.

The Damage Simulator obtains the physical damage of components using fragility functions from HAZUS-MH [20]. The Performance Simulator assesses the connectivity loss [19, 21] of the electric power system. The Recovery Simulator and Visualization Simulator subscribe to the messages from the Damage Simulator and Performance Simulator to update the damage state of components and the system performance, but they neither carry out calculations nor publish messages in this stage.

During Stage 1, (i.e. recovery phase), the Strategy Simulator distributes limited recovery resources (15 units/day) to the damaged components in order of their priority, namely, the P strategy in [19]. The recovery priority of the damaged components in the electric power system is as follows: supply nodes, demand nodes, and links/pipelines. The Recovery Simulator estimates the progress of the physical reconstruction of the damage components based on whether they are allocated recovery resources at the current time step. The Performance Simulator keeps updating the system performance of lifelines based on the recovery progress and the Visualization Simulator subscribes to the messages from the Recovery Simulator and Performance Simulator to update the reconstruction progress and system performance.

The active simulators and messages in each stage are listed in Table 1~~Table 1~~ and Table 2~~Table 2~~. The message "RTI_" referenced in the tables refers to a handful of private variables (including current timestep and stage) inside the "RTI Wrapper," which has been adapted to use the same subscription pattern as normal messages for simplification. Simulators can decide whether to subscribe the "RTI_" message depending on their needs, e.g., dependency on timestep or stage. Except for the Visualization Simulator, which is implemented using NetLogo, the other simulators are written in Matlab language. The detailed topological configuration of the lifeline system and the methodology of each simulator have been well documented in [19]. This article focuses primarily on the description of the procedure using the SRTI and GUI.

Table 1. Active simulators and messages in Stage 0

Simulator	Published Message	Subscribed Message
Scenario	Scenario	RTI_
Hazard	GroundMotion	Scenario, RTI_
Damage	Damage	Scenario, GroundMotion, RTI_
Performance	Performance	Scenario, Damage, RTI_
Recovery	-	Scenario, Damage, RTI_
Visualization	-	Scenario, Damage, Performance, RTI_

Table 2. Active simulators and messages in Stage 1

Simulator	Published Message	Subscribed Message
Performance	Performance	Recovery, RTI_
Strategy	Strategy	Performance, RTI_
Recovery	Recovery	Strategy, RTI_
Visualization	-	Recovery, Performance, RTI_

As described in the documentation of the SRTI [17], the pre-compiled SRTI files can be downloaded from the public GitHub Site. The download should include the RTI Management Server, RTI Wrapper, and SRTI GUI, all of which can execute without explicitly installation (uncompressing files and setting system paths, etc.). In addition to saving the RTI Management Server and GUI in a file directory on their local machine, the user needs to prepare individual folders for each simulator that contain the executable file of the simulator and the corresponding Wrapper. For instance,

- **/Simulator_A/**

        - SRTI_JavaWrapper_v2_00_00.jar
        - Simulator_A.jar
    - **/Simulator_B/**
        - SRTI_NetLogoWrapper_v2_00_00.jar
        - Simulator_B.nlogo
    - **/Simulator_C/**
        - SRTI_Wrapper_v2_00_00.m
        - Simulator_C.m
    - **…**

After preparing the file system, the next step is to launch the SRTI GUI.exe whose layout is shown in Figure 2~~Figure 2~~. Simulators/messages can be created and defined through the GUI. Then, users can click the objects on the Object List to add well-defined simulators or messages to the Canvas for a given stage. Action toggle buttons can be used to define the publish/subscribe relationships between simulators and messages, as shown in Figure 5~~Figure 5~~ and Figure 6~~Figure 6~~. All the defined simulators, messages, and publish/subscribe relationships can then be saved as a .project file to be imported and edited in the future. User will find the other files with extensions as described in Table 3~~Table 3~~ after saving a project. These files are coded in ASCII-text data format, typically representing a JSON object and can be read and edited in most text editors. Among them, the definition file of a simulator or message (i.e. .simdef and .mesdef) can be imported into a new project independently to prevent re-defining from scratch.

Table 3. Description for different file extension

File Extension	Description
.project	Represents a single large JSON object that defines the full project. 1 of these files exists per project.
.simdef	Optional output file that describes a single simulator referenced in the .project file.
.mesdef	Optional output file that describes a single message referenced in the .project file.

The user can either run the simulation system within the SRTI GUI, or separately using a command line prompt to open each simulator one by one. To run the simulation directly within the GUI, the user needs to click "Export Execute Files" from Menu Bar, and then click on the "Power On" button (in the Execute Buttons section) to launch an instance of the RTI Management Server and the individual simulators. After allowing for a few seconds for everything to finish opening and ensuring all simulators are connected to the RTI Management Server successfully, the other buttons in the Execute Buttons section are available to click. "Play" will start/resume the simulation system, "Pause" will pause the simulation system, and "Stop" will close the RTI Management Server and the simulators.

Other features of the SRTI GUI include being able to display the content of messages and system timestep in the inspector panel to better trace the system's execution outside the individual simulators. In addition, while the simulation is running, the color of the simulator objects on the Canvas will change according to their status, as shown in Figure 7~~Figure 7~~. Blue indicates simulators that are waiting for the messages that they subscribe to, and red indicates simulators running their internal calculation. Figure 8~~Figure 8~~ shows the progress of seismic damage and post-disaster recovery of the power system plotted by the NetLogo Visualization Simulator, where the blue, green, yellow, orange, and red colors indicate the non-, minor, moderate, extensive, and complete damage state of the lifeline components. Figure 9~~Figure 9~~ shows the time history of system performance plotted by the NetLogo Visualization Simulator.

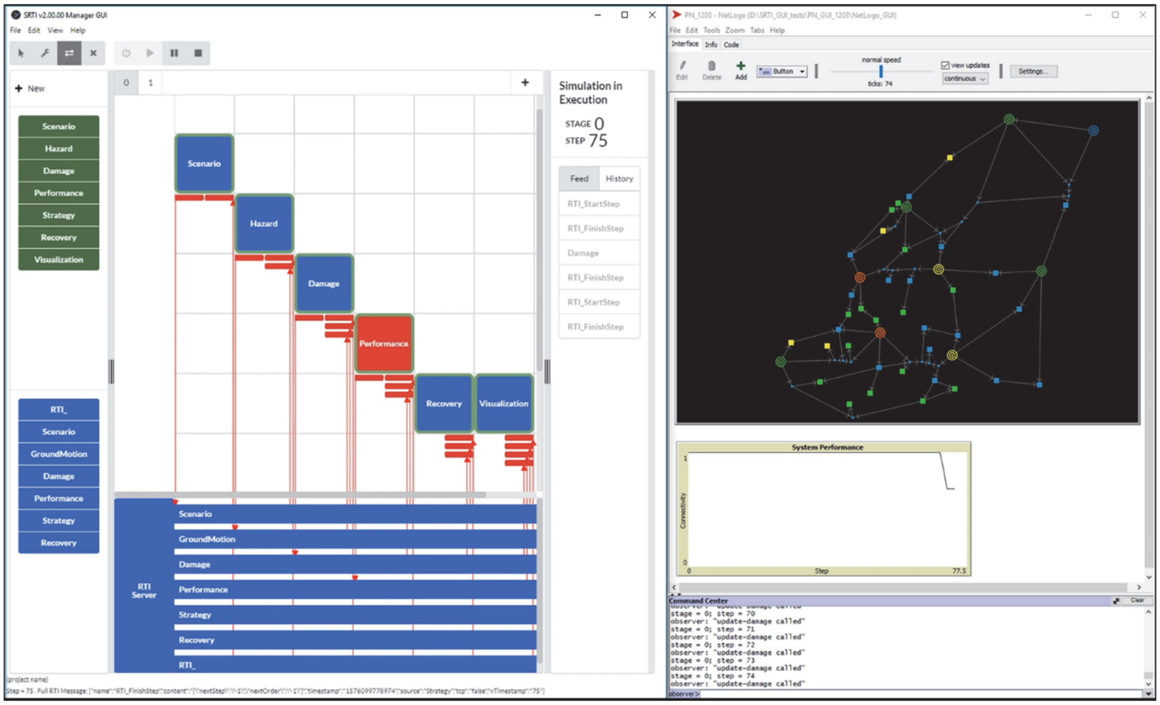


Figure 7. Screenshot of SRTI GUI and NetLogo Visualization Simulator while the simulation is running

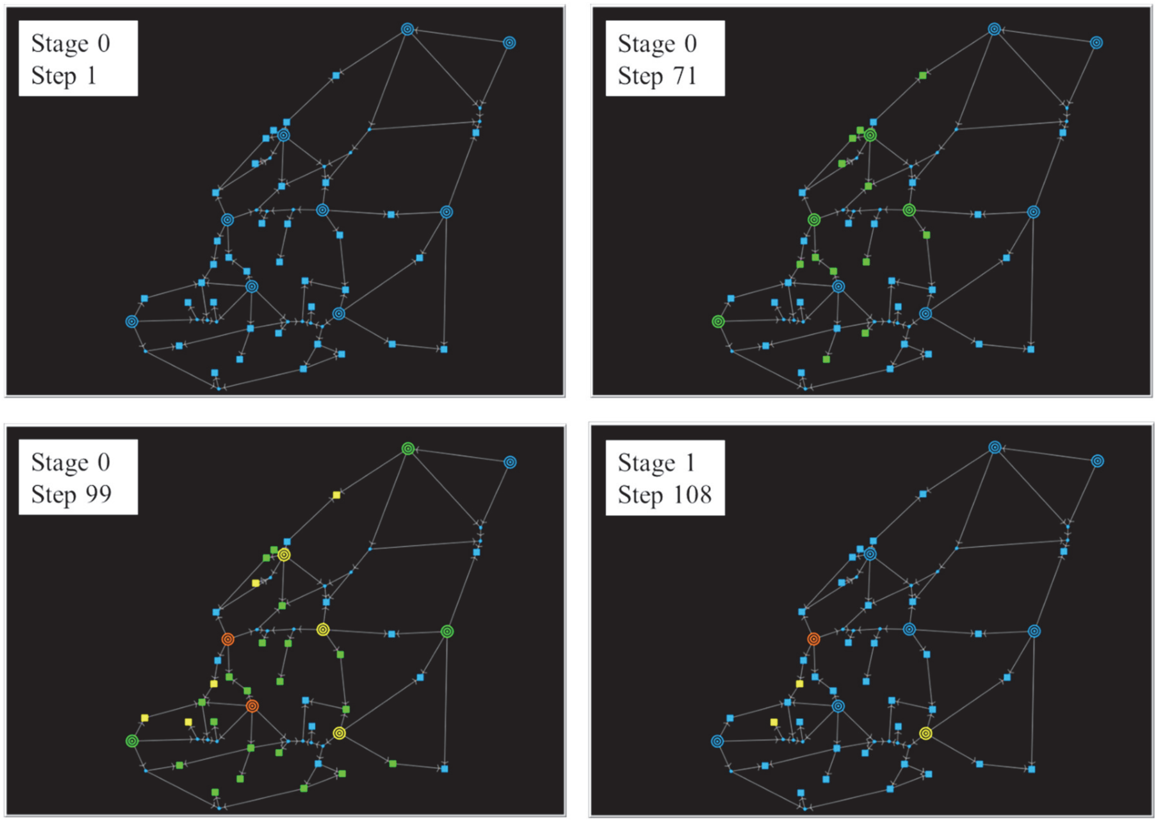


Figure 8. Progress of seismic damage and post-disaster recovery of electric power system

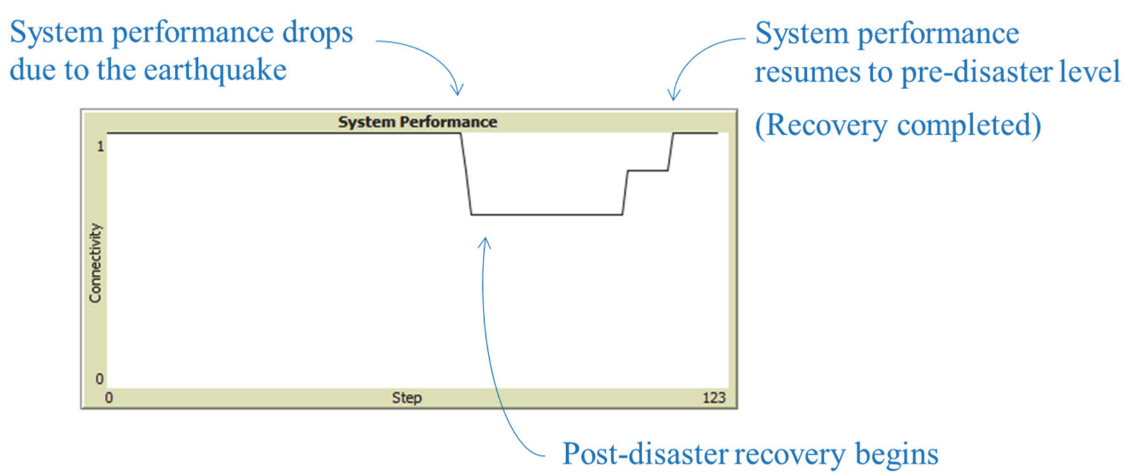


Figure 9. System performance plotted by NetLogo Visualization Simulator

6. DISCUSSION AND FUTURE OPPORTUNITIES

The concept of having a management simulator (Manager) to control the larger system (timesteps, executing specific simulators, etc.) is an intuitive method for designing simulations and controlling their complexity. Using the earlier versions of the SRTI (v1) [11, 22-24], a user would have had to send messages from a simulator to the RTI Server to a user's management simulator, a long communication channel for a simple process. Putting the management logic within the RTI Management Server reduces the number of nodes the message(s) have to travel to, providing faster overall system execution despite speed not being an intended design factor of the SRTI. The downside of this management logic is the difficulty of creating a generalized management system: SRTI v2 can handle most design preferences, but does not allow unlimited control like the earlier versions of SRTI (1).

To better support plug-and-play simulation, it is recommended that each individual simulator remain independent of each other. If one simulator is removed, or if something entirely different is added, the remaining simulators should still function as expected in that new context. Dependencies on time are possible, but also discouraged, so that the SRTI configuration files can more easily adapt different rates of time among different simulators. The case study in this article does not follow this mindset; it is not necessary, but is highly recommended. Simulation design varies greatly among different users, and these guidelines may be unintuitive for some of them to follow.

The data content from each simulator is not absolute, and cannot represent everything that an additional (future, as-of-yet unknown) simulator might require. For certain research areas, creating a standard, strictly defined message format might be necessary to help designers who want to extend a system with new simulators. This approach is against the open concept of the SRTI, but such rules can be defined and enforced while using the SRTI as middleware. Finding a balance between these lines of thought will invoke further discussion across different fields.

The time management and synchronization of virtual simulations is commonly associated with artificial time. However, there is no reason a ‘real-time’ simulator (utilizing sensors in real space) cannot be used, or possibly a hybrid model of both virtual and real simulators inside a system. This type of simulation may require sending messages at different frequencies, an option that the SRTI fully supports.

The choice to have the RTI Management Server be in control of executing the time management and message distribution, rather than having the RTI Wrapper be responsible for direct contact and connections with other Wrappers, helps ensure correct synchronization and simplicity in implementation at the expense of potential speed optimizations.

The SRTI’s different versions have each been designed with such scalability and usability as its primary goals, ignoring the common goal of execution speed. Although there are some common strategies to increase performance such as optimizing data formats by local compilation or running simulators on the same local machine (not allowing network communication), these approaches conflict to the original goals of the SRTI. Therefore, systems that require frequent transfer of large packets of data should not use SRTI, unless they are capable of absorbing the extra execution time from this bottleneck. The SRTI is designed as a black-box system; an alternative version of the tool can be developed with efficiency in mind (using different protocols for communication, and modifying ability to function across a network, are a couple strategies to achieve this) while maintaining similar API functions, to retain a similar level of usability.

7. CONCLUSIONS

SRTI is a real-time data transmission solution for distributed computational simulations that supports time-dependent simulations. The original version of SRTI (v1) focused purely on data transmission between simulators and was not explicitly designed to cater to lay users. Using a fundamentally different architecture, a new version of SRTI (v2) was built to provide a low barrier to entry for user with limited programming experience, or for teams that are unable to modify their simulators to function natively as a single system. The new version was developed to address this issue through the use of a generalized RTI Wrapper design that does not require users to edit programming code. In the new version, the RTI Management Server is responsible for controlling of simulation step and stage. The RTI Wrapper handles the connections between the RTI Management Server and user's simulator, and the message parsing. The SRTI GUI helps users prepare the configuration files for the RTI Wrapper, and allow users to launch simulators and execute simulation via the same graphic interface. The scalability and usability of the SRTI were demonstrated through a cross-language simulation of seismic damage and post-disaster recovery of a lifeline system, exploiting the SRTI GUI. The SRTI encourages thoughtful simulator design free of strict dependencies and fosters interdisciplinary collaboration for computational simulation.

ACKNOWLEDGEMENTS

This research was supported by the University of Michigan and the US National Science Foundation (NSF) through grants ACI-1638186.

TABLE OF FIGURES

TABLE OF TABLES

REFERENCES

1. Fujimoto, R. *Parallel and distributed simulation*. in *2015 Winter Simulation Conference (WSC)*. 2015. IEEE Press, Piscataway, NJ.
2. IEEE, *IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulations Applications. Entity Information and Interaction.* IEEE Std 1278-1993, 1993.
3. IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-- Federate Interface Specification*. 2010, IEEE: New York, NY.
4. IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-- Framework and Rules*. 2010, IEEE: New York, NY.
5. IEEE, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-- Object Model Template (OMT) Specification*. 2010, IEEE: New York, NY.
6. OMG, *Data Distribution Service (DDS), Version 1.4*, O.M. Group, Editor. 2015, Object Management Group.
7. LCM. *Lightweight Communications and Marshalling (LCM)*. 2018 [cited 2018 Nov. 1]; Available from: https://lcm-proj.github.io/.
8. PEER. *OpenFresco (the Open-source Framework for Experimental Setup and Control)*. 2019 [cited 2019 December 1]; Available from: https://openfresco.berkeley.edu/.
9. Google. *Google Protocol Buffers*. 2019 [cited 2019 Jan. 1]; Available from: https://developers.google.com/protocol-buffers/.

10. ICoR. *Interdependencies in Community Resilience (ICoR) Project*. 2019 [cited 2019 Dec. 15]; Available from: https://icor.engin.umich.edu/.

11. Lin, S.-Y., et al., *A Distributed Computational tool for Natural Hazards Simulation*, in *17th World Conference on Earthquake Engineering, 17WCEE*. 2020: Sendai, Japan.

12. Lin, S.-Y., et al., *Framework for Modeling Interdependent Effects in Natural Disasters: Application to Wind Engineering* Journal of Structural Engineering, 2019. **145**(5): p. 04019025.

13. Xu, L., et al., *Distributed Simulation Platforms and Data Passing Tools for Natural Hazards Engineering: Reviews, Limitations, and Recommendations.* Advanced Engineering Informatics, under review, 2020.

14. Smith, B.C., *Procedural Reflection in Programming Languages*, in *Dept. of Electrical Engineering and Computer Science*. 1982, Massachusetts Institute of Technology: Cambridge, Massachusetts.

15. Smith, B.C., *Reflection and semantics in a procedural language*. 1982, Massachusetts Institute of Technology: Cambridge, Massachusetts.

16. Ibrahim, M.H., *REFLECTION IN OBJECT-ORIENTED PROGRAMMING.* International Journal on Artificial Intelligence Tools, 1992. **01**(01): p. 117-136.

17. SRTI. *Simple Run Time Infrastructure (SRTI)*. 2019 [cited 2019 Jan. 15]; Available from: https://github.com/ICoR-code/SRTI.

18. Wilensky, U. *NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.* 1999 [cited 2019 April 26]; Available from: http://ccl.northwestern.edu/netlogo/.

19. Lin, S.-Y. and S. El-Tawil, *Time-Dependent Resilience Assessment of Seismic Damage and Restoration of Interdependent Lifeline Systems.* Journal of Infrastructure Systems, 2020. **26**(1): p. 04019040.

20. FEMA, *Earthquake loss estimation methodology: Technical manual*. 2003, National Institute of Building for the Federal Emergency Management Agency: Washington, DC.

21. Albert, R., I. Albert, and G.L. Nakarado, *Structural Vulnerability of the North American Power Grid.* 2004.

22. Lin, S.-Y. and S. El-Tawil, *Time-Dependent Computation of Multiscale Interdependencies between Lifeline Systems Subjected to Seismic Events*, in *International Conference in Commemoration of the 20th Anniversary of the 1999 Chi-Chi Earthquake*. 2019: Taipei, Taiwan.

23. Abdelhady, A.U., et al., *A Distributed Computing Platform for Community Resilience Estimation*, in *13th International Conference on Applications of Statistics and Probability in Civil Engineering, ICASP13*. 2019: Seoul, South Korea.

24. Sediek, O.A., S. El-Tawil, and J. McCormick, *Quantifying the Seismic Resilience of Communities: A Distributed Computing Framework*, in *International Conference in Commemoration of the 20th Anniversary of the 1999 Chi-Chi Earthquake*. 2019: Taipei, Taiwan.