Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

STATE-BASED MODELING AND SIMULATION TOOLKIT FOR DISCRETE-EVENT SYSTEMS: STATE GRAPH SIMULATOR

Donghun Kang

Department of Industrial and Systems Engineering
Korea Advanced Institute of Science and Technology (KAIST)
Daehak-ro 291, Yuseong-gu
Daejeon, 305-701, REPUBLIC OF KOREA

ABSTRACT

There have been many simulation packages that support the state-based modeling formalisms for modeling and simulation of a discrete-event system. However, most of them requires a programming task to define the dynamic behavior of the system so that the modelers who are not experts to the programming have a difficulty in the learning and use. Also, they lack of a integrated capability to support the model development process. This paper presents an integrated model development environment for a state-based modeling formalism, which is named state graph simulator to provide all the functionality of the model development process of graphical and tabular modeling, interactive simulation, output analysis, model verification capabilities. The state graph simulator aims to support the rapid model building of a state graph, one of state-based modeling formalisms, and the rapid development of a domain-specific simulator.

1 INTRODUCTION

In a state-based modeling formalism, the dynamics of a discrete-event system (DES) is described in terms of the *states* of the resources that resides in the system. The state-based modeling formalism is originated from the classical *finite state machine* that was used for modeling the behavior of sequential circuits (Mealy 1955). Since then, there have been a number of state-based modeling formalisms proposed in the literature, as well as a number of simulation packages that follow the state-based modeling formalisms developed.

Among the state-based modeling formalisms, DEVS (Zeigler 1976) and timed automata (Alur and Dill 1994) are regarded as the theory of modeling, and each forms an independent research community. *Timed automata* is a finite state machine extended with a finite set of real-valued clock where two types of clock constraints (*guard* and *invariant*) are placed on a state transition and a state node (Cassandras and Lafortune 2010). Also, *DEVS* (Discrete Event System Specification) is a finite state machine extended with a *lifepan* for each state in order to accommodate the timing and hierarchical modeling concept (Zeigler 1976).

A number of simulation packages and toolkits have been introduced from the academia to support the state-based modeling and simulation of a DES. Especially, DEVS-based simulation packages have been widely developed, such as DEVSJava (Sarjoughian and Zeigler 1998), CoSMos (Sarjoughian and Elamvazhuthi 2009), DEVSim++ (Kim et al. 2011) and so on. Most of state-based simulation packages require the users to program for defining the simulation models, which is a difficult task to the non-expert developers. To mitigate this difficulty, a few simulation packages are introduced with the support of graphical modeling and the simplification of the model development process: DEVS-Suite (Kim et al. 2009), VLE (Quesnel et al. 2007), and CD++ (Bonaventura et al. 2013). The graphical modeling capability provided in these software packages is limited only to support the graphical modeling of the coupled models while the internal behavior is left to program by the users. Also, each function for the development process is separated into the independent software tools, which increases the loads of learning and use for the users.

The *state transition diagram* is a graphical specification for the state-based modeling formalism, which was firstly introduced with the finite state machine. As the dynamic behavior of a system becomes complex, the number of states and the state transitions also increases, which makes it difficult to draw a concise and readable state transition diagram. In that case, another specification for the state-based modeling formalism, *state transition table*, is more suitable to specify the dynamic behavior in a table showing what state will move to, based on the current state and other inputs. Among the aforementioned simulation packages, no simulation package supports the tabular specification.

This paper will present an integrated model development environment for a state-based modeling formalism, which is named *state graph simulator* to provide all the functionality for the model development process with graphical and tabular modeling capability while reducing the dependency on the programming expertise. With the state graph simulator, the users can model a DES with the graphical and tabular specifications, simulate the specified model with the interactive user interface, conduct the output analysis with the automatic data collection and visualization, and verify the model with a graphical method, sequence diagram. The state graph simulator not only supports the rapid model building, but also provides the rapid development of a domain-specific simulator (or simulation-based decision-support system) with the automatic code generation and simulation components.

In the following sections, Section 2 describes a state-based modeling formalism, *state graph*. Section 3 presents the state graph simulator that follows the state graph modeling formalism presented in Section 2 and provides the rapid model building and development capabilities. Section 4 describes an illustrative example of constructing a domain-specific simulator with the state graph simulator and Section 5 has the summary and discussion.

2 STATE GRAPH

The *state graph* is one of state-based modeling formalism that provides a well-defined set of graphical conventions with a formal syntax for unambiguous understanding among the modeling experts and the simulation algorithm for executing the state graph models (Choi and Kang 2013).

Figure 1 presents the state graph model of a single server system. In general, the single server system consists of a `Buffer` and a `Machine`, but a `Job Generator` is introduced to supply the job entities into the system. The state graph model consists of a *composite state graph model* represented in the *object interaction diagram* and a set of *atomic state graph models* represented in the *state transition diagrams*.

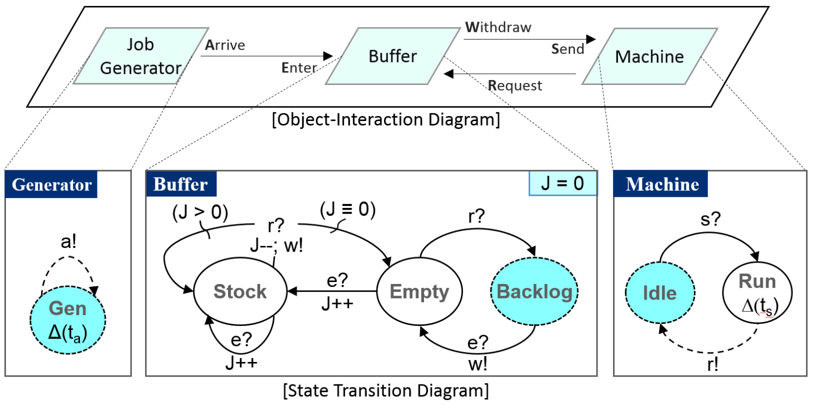
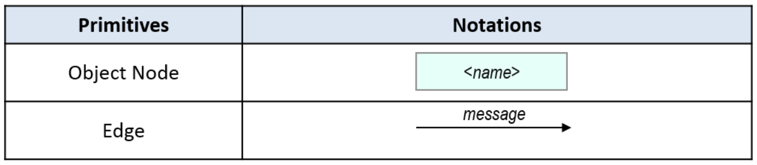


Figure 1: State graph model of a single server system

The resources (or objects) in the above single server system are `Job Generator`, `Buffer`, and `Machine`. In a DES, an *event* represents a system change at a discrete time, which is represented as an *interaction* (or *message*) in the state graph model. The object interaction diagram at the upper part of Figure 1 describes the interactions between the resources in the system. The interactions among the three resources in Figure 1 are identified as (1) job transfer from `Job Generator` to `Buffer`, (2) request for the next job from `Machine` to `Buffer`, and (3) job transfer from `Buffer` to `Machine`. Namely, `Job Generator` sends out *Arrive* message to `Buffer` whenever a new job arrives at the system. The new job *enters* the waiting space of `Buffer`. Once the `Machine` becomes idle, it sends out *Request* message to `Buffer` for the next job. Then, the `Buffer` sends out *Withdraw* message to the `Machine`. Figure 2 represents the graphical conventions for the object interaction diagram of Figure 1.

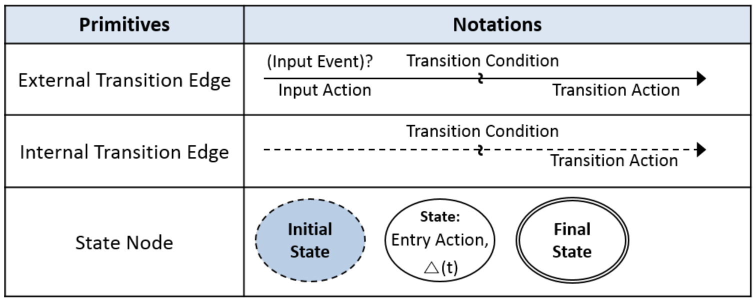


Primitives	Notations
Object Node	*<name>*
Edge	*message* →

Figure 2: Graphical convention of a object interaction diagram

The lower part of Figure 1 presents the state transition diagrams of three objects that comprise of the single server system. The state transition diagram is a graphical specification for an atomic state graph model, which describes the state transition of an object based on the interactions with other objects.

The `Job Generator` sends out an *Arrive* message every t_a time units in a `Gen` state. The `Buffer` has three states of `Backlog`, `Empty`, and `Stock`, and a state variable `J` that represents the number of jobs waiting at the `Buffer`. Initially, the `Buffer` is set to `Backlog` state and its state variable `J` is set to zero. When the `Buffer` receives an *Enter* message (`e?`), it moves onto `Empty` state while sending out a *Withdraw* message (`w!`). Upon receiving another *Enter* message in `Empty` state (`e?`), it increases the number of waiting jobs by one (`J++`) and moves onto `Stock` state. The `Machine` starts with `Idle` state. Upon receiving a *Send* message (`s?`), the `Machine` moves onto `Run` state. After staying at `Run` state for t_s time units, it changes to `Idle` state while sending out a *Request* message (`r!`). Figure 3 represents the graphical conventions for the state transition diagrams of Figure 1.



Primitives	Notations
External Transition Edge	(Input Event)? Transition Condition Input Action Transition Action
Internal Transition Edge	Transition Condition Transition Action
State Node	Initial State; State: Entry Action, △(t); Final State

Figure 3: Graphical convention of a state transition diagram

In modeling formalisms for DESs, such as ACD and event graph, the model specification can be given in three ways: (1) graphical specification, (2) tabular specification, and (3) algebraic specification (Choi and Kang 2014). The state graph also provides all of the three specifications. However, the algebraic specification with a detailed description of the transition function is both tedious and difficult; thus graphical and tabular specification are preferred (Hopcroft et al. 2006).

Tables 1 shows a state transition table of `Buffer` atomic state graph model given in Figure 1. The state transition table contains all the information of the state transition diagram in a clearly-defined structured. Specified for each State are its Name and entry Action, Input Event and Action, Transition Condition and Action, and Next State.

The graphical and tabular specifications contains all the information for the atomic state graph model. Therefore, they are interchangeable. However, as the target DES becomes large and complex, the number of graphical primitives that comprise of the state transition diagram also increases, which results in difficulties

Table 1: State transition table of a `Buffer` atomic state graph model in Figure 1

State		**Input**		**Transition**		**Next**
Name	**Action**	**Event**	**Action**	**Condition**	**Action**	**State**
Backlog	-	(e)?	-	True	(w)!	Empty
Empty	-	(e)?	J++	True	-	Stock
		(r)?	-	True	-	Backlog
Stock	-	(e)?	J++	True	-	Stock
		(r)?	(w)!; J--	$J \equiv 0$	-	Empty
				$J > 0$	-	Stock

in building and understanding. Especially, the state transition diagram that supports different types of actions, such as entry action, input action, transition action, and so on, may have a difficulty in specifying the state transitions including these actions and conditions in the graph. On the other hand, the tabular specification provides a concise and readable representation of the atomic state graph model.

Table 2 presents an object interaction table of the composite state graph model given in Figure 1. The object interaction table is specified for each interaction (or message) between two objects, which consists of Source Object and its Output Message, and Receiving Object and its Input Message.

Table 2: Object interaction table of the composite state graph model in Figure 1

Source Object	**Output Message**	**Receiving Object**	**Input Message**
Job Generator	Arrive	Buffer	Enter
Buffer	Withdraw	Machine	Send
Machine	Request	Buffer	Request

3 STATE GRAPH SIMULATOR

The state graph simulator supports a state-based modeling and simulation of DESs with state graph described in Section 2. Figure 4 presents the system architecture of state graph simulator that consists of several graphical user interface (GUI) components and two libraries for the modeling and simulation of a state graph model. In state graph simulator, a composite state graph model is specified using the object interaction diagram editor and an atomic state graph model is specified using the state transition table editor. The adoption of the state transition table for specifying an atomic state graph model allows the users to have a concise and readable specification for a large and complex system.

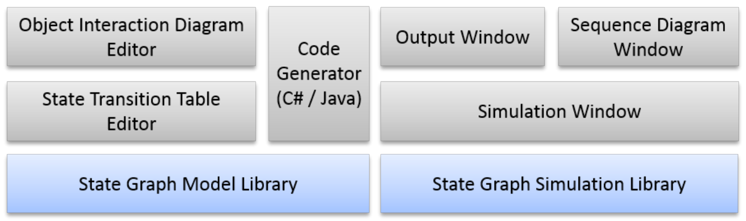
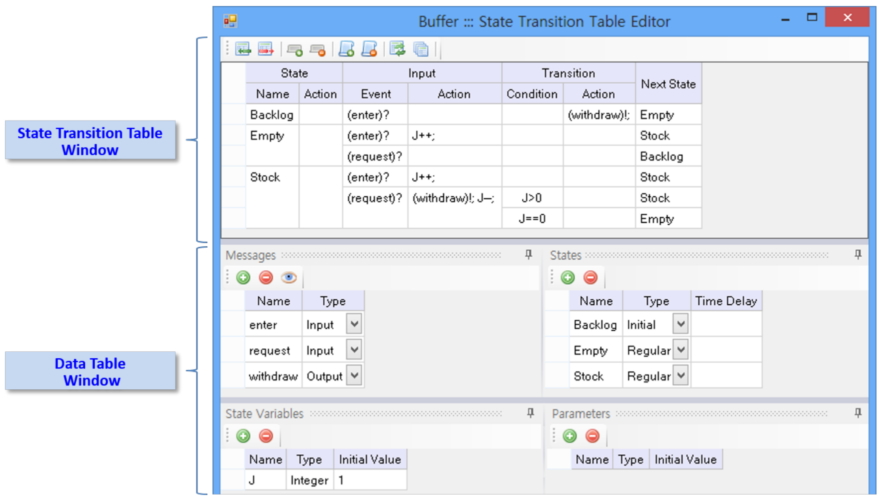

Object Interaction Diagram Editor
Code Generator (C# / Java)
Output Window
Sequence Diagram Window
State Transition Table Editor
Simulation Window
State Graph Model Library
State Graph Simulation Library

Figure 4: System architecture of state graph simulator

As presented in Figure 4, the state graph simulator is not limited only to state graph modeling, but also supports the simulation and model verification. In addition, the state graph simulator provides two components (model and simulation libraries) that can be used for constructing a dedicated simulator. More details on the construction of a dedicated or domain-specific simulator will be introduced in Section 4.

3.1 Modeling of State Graph Model

Figure 5 presents the state transition table editor with `Buffer` atomic state graph model of Figure 1. The state transition table editor consists of a *state transition table window* for specifying the state transitions as same in Table 1 and the *data table window* to define the data tables for state variables, parameters, messages, and states.


Buffer ::: State Transition Table Editor
State Transition Table Window
Data Table Window

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
Backlog		(enter)?			(withdraw)!;	Empty
Empty		(enter)?	J++;			Stock
		(request)?				Backlog
Stock		(enter)?	J++;			Stock
		(request)?	(withdraw)!; J--;	J>0		Stock
				J==0		Empty

Messages

Name	Type
enter	Input
request	Input
withdraw	Output

States

Name	Type	Time Delay
Backlog	Initial	
Empty	Regular	
Stock	Regular	

State Variables

Name	Type	Initial Value
J	Integer	1

Parameters

Name	Type	Initial Value

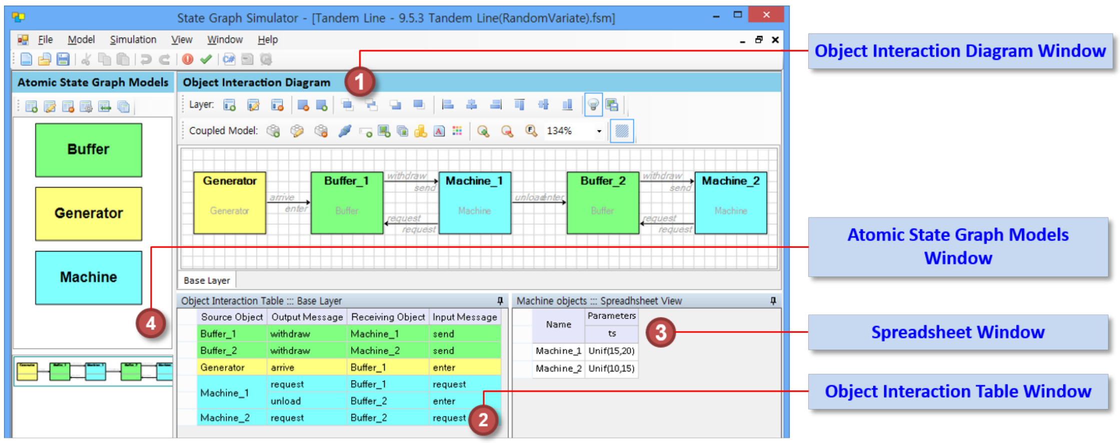
Figure 5: State transition table editor with `Buffer` atomic state graph model in Figure 1

Prior to specifying the state transitions, data tables have to be defined in the data table window: (1) *State variable data table* for state variables with name, type, and initial value, (2) *Parameters data table* for the parameters of the atomic state graph model, such as service time (t_s) and inter-arrival time (t_a), (3) *Messages data table* for input and output messages with name and type, and (4) *States data table* for specifying the attributes of each state, such as state name, state type {Initial, Regular, Final}, and time delay. Specified in Figure 5 are a state variable `J` of integer type with an initial value, messages of *Enter*, *Request*, and *Withdraw*, and states of `Backlog` with the initial type, and `Empty` and `Stock` with the regular type. In `Buffer` atomic state graph model, no parameter is defined.

The toolbar located in the top of the state transition table window provides a few buttons to manipulate the table structure, such as add/remove a state, an input event, or a state transition for each edge. After the table structure is constructed, the state transitions can be entered as same as the state transitions in Table 1.

All the state transition tables of the atomic state graph models can be accessed in the *atomic state graph models window* of the *object interaction diagram editor* as presented in Figure 6. Once all the atomic state graph models are specified, the object interaction diagram can be modeled in the *object interaction diagram window*: (1) drag-and-drop a rectangle of an atomic state graph model in the atomic state graph models window into the object interaction diagram window, and then (2) an *atomic object node* of the designated atomic state graph model will be instantiated. Once all the atomic object nodes are placed, the message couplings that represent the interactions among the atomic objects can be connected: (1) click `Connect` button in the toolbar of the object interaction diagram editor to enable the connection, (2) click a source atomic object node and drag-and-drop a port placed on the selected atomic object node to the destination atomic object node, (3) an Message Coupling dialog box will show up and specify the output message of a source atomic object and the input message of a destination atomic object.

The *object interaction table window* allows the users to check the message couplings among the atomic objects in a table. The parameter values of the atomic objects can be set or modified in the *spreadsheet window*. For example, `Machine_1` and `Machine_2` atomic objects are instances of `Machine` atomic state graph model having a parameter of service time (t_s).


State Graph Simulator - [Tandem Line - 9.5.3 Tandem Line(RandomVariate).fsm]
File
Model
Simulation
View
Window
Help
Atomic State Graph Models
Object Interaction Diagram
Layer:
Coupled Model:
134%
Buffer
Generator
Machine
Generator
Buffer_1
Machine_1
Buffer_2
Machine_2
arrive
enter
withdraw
send
request
unload
Base Layer
Object Interaction Diagram Window
Atomic State Graph Models Window
Spreadsheet Window
Object Interaction Table Window
1
2
3
4

Object Interaction Table ::: Base Layer

Source Object	Output Message	Receiving Object	Input Message
Buffer_1	withdraw	Machine_1	send
Buffer_2	withdraw	Machine_2	send
Generator	arrive	Buffer_1	enter
Machine_1	request	Buffer_1	request
	unload	Buffer_2	enter
Machine_2	request	Buffer_2	request

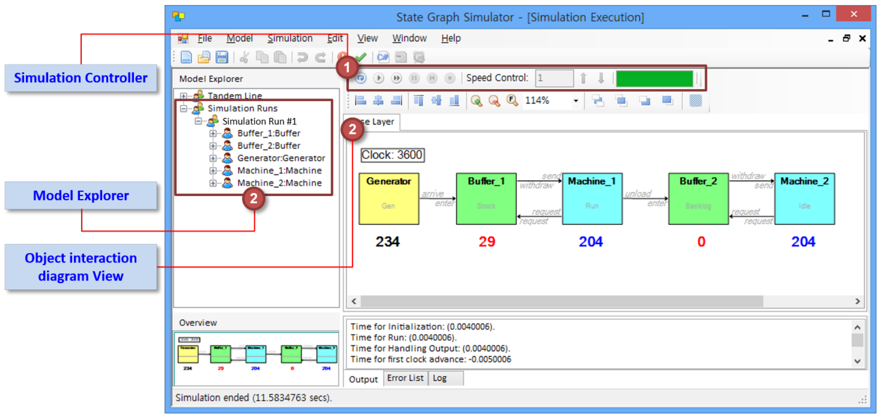
Machine objects ::: Spreadhsheet View

Name	Parameters
	ts
Machine_1	Unif(15,20)
Machine_2	Unif(10,15)

Figure 6: Object interaction diagram editor of state graph simulator

3.2 Simulation of State Graph Model

The *simulation window* provides the simulation execution of a state graph model. Figure 7 presents the simulation window in which the simulation execution is finished. The simulation window consists of the *simulation controller*, *model explorer*, and the object interaction diagram view.



```
Time for Initialization: (0.0040006).
Time for Run: (0.0040006).
Time for Handling Output: (0.0040006).
Time for first clock advance: -0.0050006
```

```
Simulation ended (11.5834763 secs).
```

Figure 7: Simulation window of state graph simulator with a tandem line system

During the simulation execution, the object interaction diagram view updates the current state of each atomic object and the current simulation clock as the simulation progresses. For example, in Figure 7, `Buffer_1` atomic object is in `Stock` state and `Machine_1` atomic object is in `Run` state at the simulation clock of 3600. The simulation window also can present the current value of a state variable using a *label* where the number below each atomic object node in Figure 7 indicates the current value of a state variable that belongs to the atomic object. For example, the number (29) below `Buffer_1` atomic object node represents the current value of the state variable `J` of `Buffer_1` atomic object. The auxiliary nodes just like a label are *Picture Set* for displaying the designated picture for each state of an atomic object, and *Gauge* for displaying a state variable's value in the graphics. As a modeling formalism for a DES, the state graph provides the simulation algorithm, named *synchronization algorithm* that synchronizes the simulation clock and exchanges the messages among the atomic simulators where each atomic simulator is constructed for each atomic object and takes care of simulation of the atomic object (Choi and Kang 2013).

3.3 Output Analysis and Model Verification

After the simulation runs, the simulation output can be accessed in the *model explorer* located at the left of the simulation window. The following simulation output window in Figure 8 will pop up upon double-clicking any atomic object name under the ”Simulation Run #1” of model explorer in Figure 7.

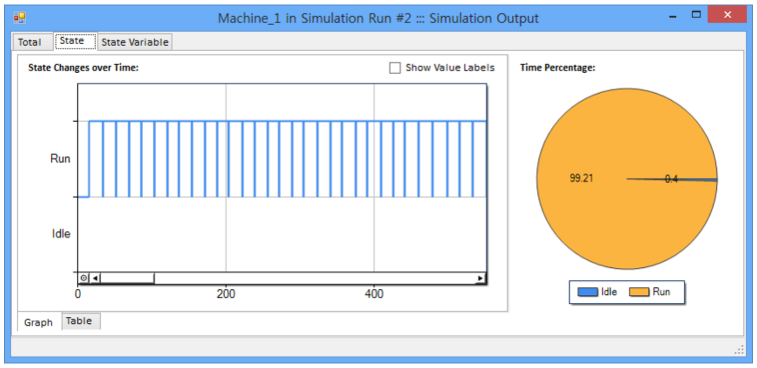


Figure 8: Output window for `Machine_1` atomic object

The simulation output window provides three outputs for each atomic object: (1) *Total output* for displaying the value changes of state and state variables over time, (2) *State output* for displaying the state changes over time and time percentage of each distinct state (e.g. utilization), (3) *State variable output* for displaying the value changes of each state variable over time and its minimum, mean, and maximum values. Figure 8 presents the state output of `Machine_1` atomic object where its utilization (time percentage of staying at Run state) is 99.21 %.

To minimize the errors in the simulation execution, the state graph simulator provides a model checking functionality to find out the syntax errors in the state transition table. However, the logical errors in a state graph model cannot easily be found before the simulation run. Therefore, the *model verification* is required to ensure that the computer program of the computerized model and its implementation are correct (Sargent 2014). Especially, the state graph model consists of several atomic object and their dynamic behaviors are based on the interactions among them. In software engineering, the dynamic behaviors of software components can be specified using sequence diagrams of the Unified Modeling Language (UML). The sequence diagram is a graphical specification language to show the interactions between objects in the sequential order that those interactions occur (Booch et al. 2005). The state graph simulator provides the sequence diagram of a simulation execution so that the user can check the logical flow in the state transitions and interactions among the atomic objects are correctly performed as shown in Figure 9.

Figure 9 presents the *sequence diagram window* after the simulation is executed. For each atomic object, its *lifeline* in the vertical line is placed and the state changes within an atomic object are displayed in the time order. Each box placed on the lifeline indicates the time portion of staying at a specific state. The *horizontal arrows* represent the *messages* exchanged between the atomic objects and its text label indicates the input message of receiving atomic object and its receiving time (e.g. enter@9.64). The provided sequence diagram can facilitate the model verification in that the state changes from the interaction and the interaction initiated by the state change can be captured in a graphical way. This is the unique feature of the state graph simulator, which is available in the author’s website (Kang 2015).

4 ILLUSTRATIVE EXAMPLE

The users are allowed to model and simulate a state graph model within the state graph simulator, but they are also able to build their own dedicated simulators using the state graph simulator. For example, the rapid development of an urban traffic simulator can take full advantages of the integrated simulation

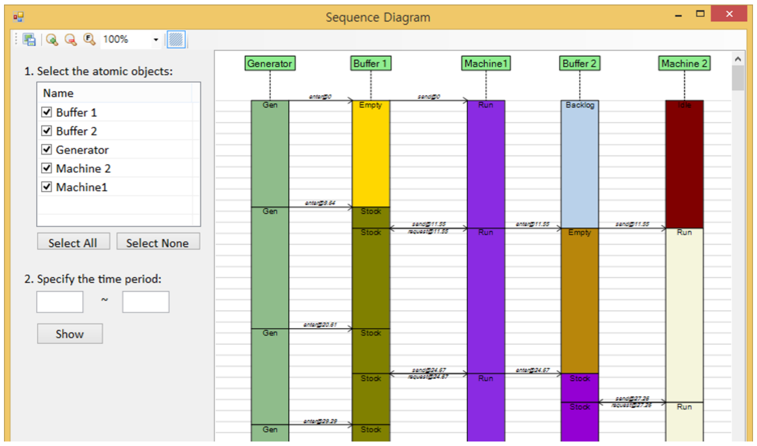


Figure 9: Sequence diagram window of state graph simulator

model development environment of the state graph simulator. Figure 10 presents the simulation model development process for the urban traffic simulator using the state graph simulator.

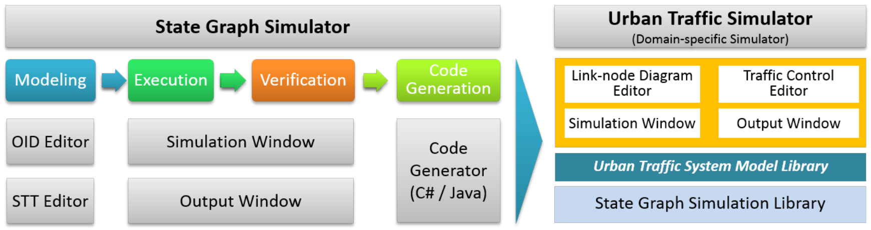


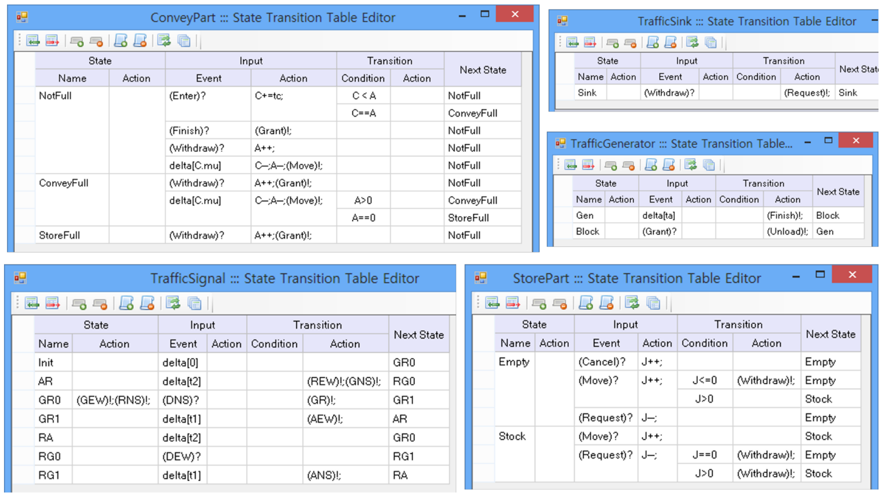
Figure 10: Development process of urban traffic simulator using the state graph simulator

In the state graph simulator, the user constructs a state graph model of an urban traffic system using the object interaction diagram editor and state transition table editor, executes the state graph model, and then performs the model verification using the output window and sequence diagram window. If the state graph model turns out that it works correctly, the state graph simulator can generate the source code of the state graph model in Microsoft C# or Java programming language. Once the source code are generated, these codes are assembled into a *urban traffic system model library*. On the top of the state graph simulation library, the domain-specific modeling tools, such as link-node diagram editor and traffic control editor are constructed and the simulation and analysis tools are also developed.

4.1 State Graph Modeling of Urban Traffic System

The signalized urban traffic system consists of a *road network* and the *traffic signal controllers* for each intersection. The link-node diagram is introduced to represent a physical road network concisely and clearly (Myung et al. 2014), which is a directed graph having a set of links and a set of nodes. A *link* is a directed edge that represent a lane between two intersections where inflow occurs only at the start point of a link and out flow occurs only at the end point of a link (Myung et al. 2014). A *node* is a vertex that connects the links and indicates the stopping point at each incoming lane of an intersection (`Gate` and `End` nodes), splitting and merging of lanes (`Split` and `Merge` nodes), and the generation and disposal of vehicles (`Source` and `Sink` nodes).

The basic modeling concept of an urban traffic system using the state graph is first introduced in Kang et al. (2012), and then it is elaborated in Myung et al. (2014). In the state graph modeling, a link is represented by a pair of `ConveyPart` atomic state graph model for conveying the vehicles at the start point of a link to the end point of the link and `StorePart` atomic state graph model for storing the vehicles awaiting at the end point in order to move onto the next link by passing the connected node. Each node is represented by the respective atomic state graph model: `Traffic Generator` and `Traffic Sink` atomic state graph models for Source and Sink nodes, `Signal Gate` and `End Point` atomic state graph models for Gate and End nodes, `Diverter` and `Merge Controller` atomic state graph models for Split and Merge nodes. Figure 11 depicts the atomic state graph models of only four of the above-mentioned modeling components with the state transition table editor. Presented in Figure 11 is also a `Traffic Signal` atomic state graph model for a traffic signal controller where each state represents a status of traffic lights at each incoming direction.


ConveyPart ::: State Transition Table Editor
TrafficSink ::: State Transition Table Editor
TrafficGenerator ::: State Transition Table...
TrafficSignal ::: State Transition Table Editor
StorePart ::: State Transition Table Editor

ConveyPart ::: State Transition Table Editor

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
NotFull		(Enter)?	C+=tc;	C < A		NotFull
				C==A		ConveyFull
		(Finish)?	(Grant)!;			NotFull
		(Withdraw)?	A++;			NotFull
		delta[C.mu]	C–;A–;(Move)!;			NotFull
ConveyFull		(Withdraw)?	A++;(Grant)!;			NotFull
		delta[C.mu]	C–;A–;(Move)!;	A>0		ConveyFull
				A==0		StoreFull
StoreFull		(Withdraw)?	A++;(Grant)!;			NotFull

TrafficSink ::: State Transition Table Editor

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
Sink		(Withdraw)?			(Request)!;	Sink

TrafficGenerator ::: State Transition Table...

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
Gen		delta[ta]			(Finish)!;	Block
Block		(Grant)?			(Unload)!;	Gen

TrafficSignal ::: State Transition Table Editor

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
Init		delta[0]				GR0
AR		delta[t2]			(REW)!;(GNS)!;	RG0
GR0	(GEW)!;(RNS)!;	(DNS)?			(GR)!;	GR1
GR1		delta[t1]			(AEW)!;	AR
RA		delta[t2]				GR0
RG0		(DEW)?				RG1
RG1		delta[t1]			(ANS)!;	RA

StorePart ::: State Transition Table Editor

State		Input		Transition		Next State
Name	Action	Event	Action	Condition	Action	
Empty		(Cancel)?	J++;			Empty
		(Move)?	J++;	J<=0	(Withdraw)!;	Empty
				J>0		Stock
		(Request)?	J–;			Empty
Stock		(Move)?	J++;			Stock
		(Request)?	J–;	J==0	(Withdraw)!;	Empty
				J>0	(Withdraw)!;	Stock

Figure 11: Atomic state graph models of the modeling components for a link-node diagram in the state transition table editor

Presented in Figure 12 is the composite state graph model of a four-leg intersection where each road consists of one lane with the object interaction diagram editor. The Traffic Signal atomic object is located in the center of the diagram and is connected to all the Signal Gate atomic objects to control the traffic flows. The Traffic Generator and Sink atomic objects are placed in the boundary of the diagram.

4.2 Model Verification

There are two ways to verify the model in the state graph simulator: one is to use the auxiliary atomic state graph model to count the numbers of vehicles that have passed a certain point and the other is to use the sequence diagram window to check whether the logical flow and message exchanges are made correctly or not. Figure 13 presents a portion of simulation window that uses three labels to capture the *number of generated cars*, the *number of exited cars*, and the *number of on-going cars in the system*. If the state graph model in Figures 11 and 12 works correctly, the number of on-going cars should be equal to the number of generated cars subtracted by the number of exited cars. To collect the number of generated cars and exited cars, a new atomic state graph model is introduced, named *Counter*, which simply increases its own counter variable whenever it receives a message from others (e.g. arrival and leave of cars). Similarly, some specialized atomic state graph model may be useful to collect the output data, such as *tally statistics collector* for waiting times and turn-around times, *time-dependent statistics* collector for average queue length and machine utilization.

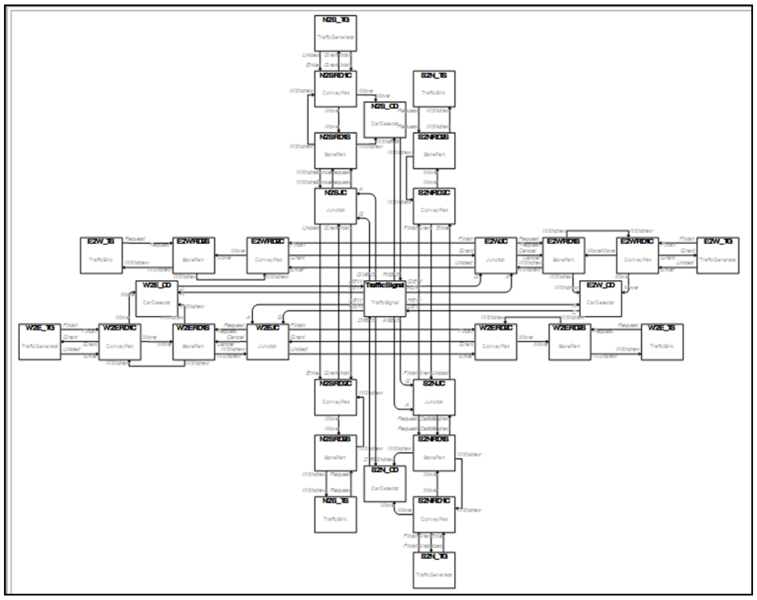


Figure 12: Composite state graph model of a four-leg intersection with the object interaction diagram editor

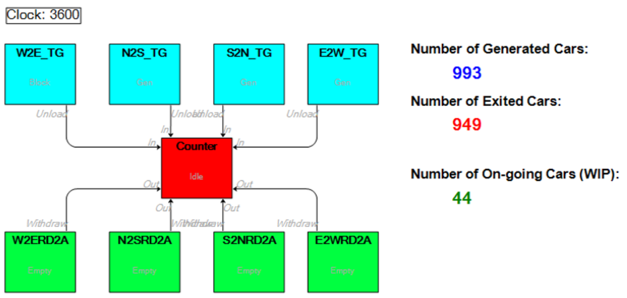


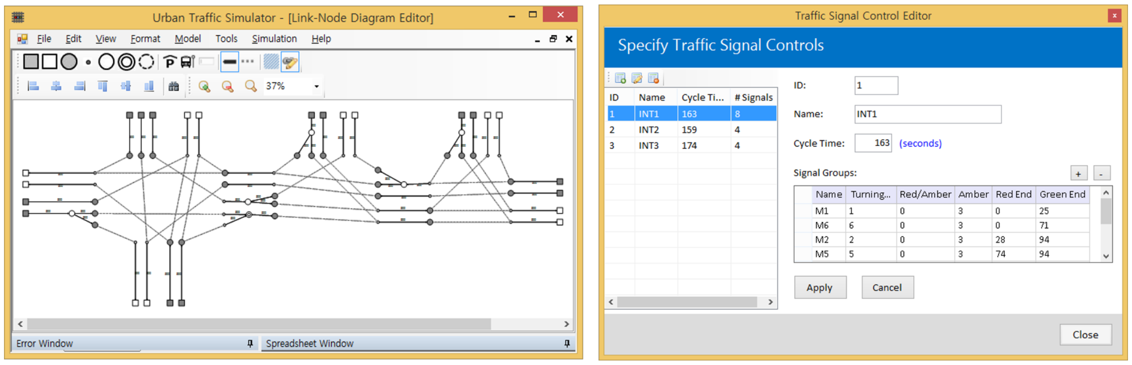
Figure 13: Verification of state graph model for an traffic intersection

4.3 Implementation of Urban Traffic Simulator

The implementation of urban traffic simulator starts with the code generation of the state graph model. The generated source codes are assembled into a *urban traffic system model library* that consists of the atomic simulators for the above-mentioned atomic state graph models.

The urban traffic simulator requires a domain-specific model that includes a link-node diagram and dual-ring-and-barrier diagrams for describing the road network and traffic signal controllers. The *link-node diagram editor* in Figure 14-(a) allows the users to specify the link-node diagram using the graphical elements, such as nodes and links. Also, the *traffic signal controller editor* in Figure 14-(b) let the users specify the signal phases of a traffic signal control in a tabular form.

Before the simulation run, the simulation window in Figure 15 translates a domain-specific model into a composite state graph model by mapping the modeling components of the link-node diagram and traffic signal controllers into the respective atomic state graph simulators provided in the urban traffic system model library. During the simulation run, the simulation window provides two simulation views: (1) link-node diagram view for displaying the congestion level on each link with the color scheme and updating the number of vehicles on each link, and (2) status of a traffic signal controller at each intersection.


Urban Traffic Simulator - [Link-Node Diagram Editor]
File
Edit
View
Format
Model
Tools
Simulation
Help
37%
Error Window
Spreadsheet Window
Traffic Signal Control Editor
Specify Traffic Signal Controls
Apply
Cancel
Close

ID	Name	Cycle Ti...	# Signals
1	INT1	163	8
2	INT2	159	4
3	INT3	174	4

ID: 1

Name: INT1

Cycle Time: 163 (seconds)

Signal Groups:

Name	Turning...	Red/Amber	Amber	Red End	Green End
M1	1	0	3	0	25
M6	6	0	3	0	71
M2	2	0	3	28	94
M5	5	0	3	74	94

Figure 14: Urban traffic simulator: (a) link-node diagram editor and (b) traffic signal controller editor

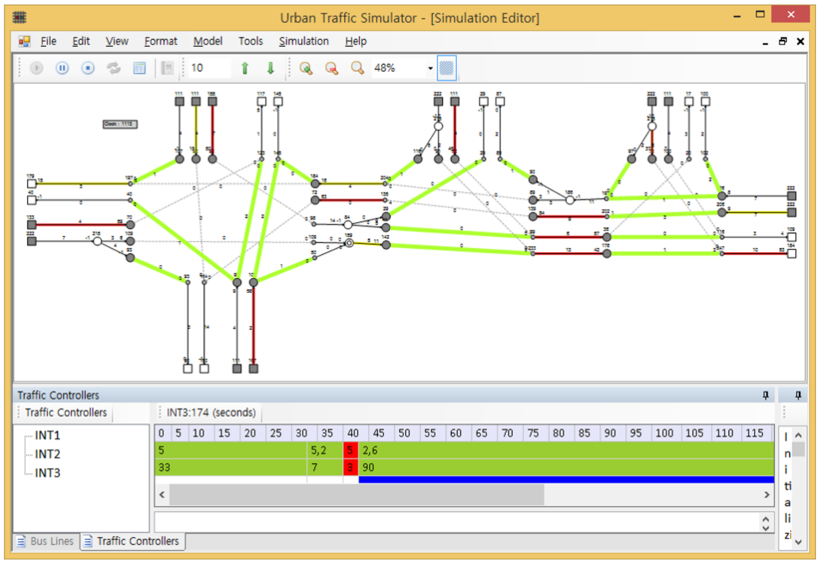


Figure 15: Simulation window of urban traffic simulator

5 CONCLUSION

The state graph simulator provides all the functionality for the state-based modeling and simulation development process in a single integrated development environment with the graphical and tabular modeling, interactive simulation and output analysis, and model verification capabilities. The state graph modeling consists of the composite state graph modeling with the object interaction diagram editor and the atomic state graph modeling with the state transition table editor. The simulation comes with the interactive object interaction view. The output analysis and model verification is based on the automatic data collection to provide the visual system trajectories and the sequence diagram. Also, the state graph simulator provides the components for constructing a domain-specific simulator, such as code generator and simulation library.

Further research should extend the state graph simulator so as to cope with more streamlined modeling development process and modeling and simulation of a large-scale DES. In that sense, the state graph simulator will support the parallel simulation with the automatic load balance and distributed simulation with HLA/RTI in the near future. The state graph modeling formalism can facilitate the federation architecture modeling and automatic generation of federate simulation codes and a federation object model. Also, modeling template can be incorporated into the state graph simulator to help the re-use of atomic state graph models and to provide the primitives for the data collection.

REFERENCES

Alur, R., and D. L. Dill. 1994. "A Theory of Timed Automata". *Theoretical Computer Science* 126 (2): 183–235.

Bonaventura, M., G. A. Wainer, and R. Castro. 2013. "Graphical Modeling and Simulation of Discrete-Event Systems with CD++ Builder". *SIMULATION* 89 (1): 4–27.

Booch, G., J. Rumbaugh, and I. Jacobson. 2005. *Unified Modeling Language User Guide*. Addison-Wesley.

Cassandras, C., and S. Lafortune. 2010. *Introduction to Discrete Event Systems*. 2nd ed. Springer.

Choi, B. K., and D. Kang. 2013. *Modeling and Simulation of Discrete Event Systems*. John Wiley & Sons.

Choi, B. K., and D. Kang. 2014. "How to Develop Your Own Simulators for Discrete-Event Systems". In *Proceedings of the 2014 Winter Simulation Conference*, 147–161.

Hopcroft, J. E., R. Motwani, and J. D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Addison Wesley.

Kang, D. 2015. "State Graph Simulator: State-based Modeling and Simulation Toolkit for Discrete-Event Systems". Accessed April. 1, 2015. http://www.vms-technology.com/sgs/.

Kang, D., J. Kong, and B. K. Choi. 2012. "DEVS Modeling of Urban Traffic System". In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation*, Article No. 16.

Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. 2009. "DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring". In *Proceedings of the 2009 Spring Simulation Multiconference*, Article 161. Society for Computer Simulation International.

Kim, T. G., C. H. Sung, S. Y. Hong, J. H. Hong, C. B. Choi, J. H. Kim, K. M. Seo, and J. W. Bae. 2011. "DEVSim++ toolset for defense modeling and simulation and interoperation". *Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* 8 (3): 129–142.

Mealy, G. H. 1955. "A method to synthesizing sequential circuits". *Bell System Technical Journal* 34 (5): 1045–1079.

Myung, M., D. Kang, and B. K. Choi. 2014. "State-based Modeling and Simulation of Urban Traffic Systems Including Signalized Intersections". In *Proceedings of The 15th Asia Pacific Industrial Engineering and Management Systems Conference*.

Quesnel, G., R. Duboz, E. Ramat, and M. Traor. 2007. "VLE: a multimodeling and simulation environment". In *Proceedings of the 2007 Summer Computer Simulation Conference*, 367–374.

Sargent, R. G. 2014. "Verifying and Validating Simulation Models". In *Proceedings of the 2014 Winter Simulation Conference*, 118–131.

Sarjoughian, H. S., and V. Elamvazhuthi. 2009. "CoSMos: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation". In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Article No. 59.

Sarjoughian, H. S., and B. P. Zeigler. 1998. "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment". In *Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation*, 29–35.

Zeigler, B. P. 1976. *Theory of Modeling and Simulation*. John Wiley & Sons.

AUTHOR BIOGRAPHIES

DONGHUN KANG is a post-doctoral researcher in the Department of Industrial and Systems Engineering at KAIST in Daejeon, South Korea. He received a Ph.D. from KAIST in Industrial Engineering in 2011. His research interests lie in the DES M&S and its applications. His email address is donghun.kang@kaist.ac.kr.