State-based Modeling and Simulation of Discrete-Event Systems using Graphical and Tabular Specifications

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 require a programming task to define the dynamic behavior of the system so that the modelers who are not experts in the programming have difficulty in the learning and use. This paper presents a state-based modeling and simulation toolkit, which is named state graph simulator, to provide all the functionalities of the model development process with the graphical and tabular modeling, interactive simulation, output analysis, and model verification capabilities. The state graph simulator aims to support the rapid model building of a state graph, one of state-based modeling formalism, which will be illustrated with a signalized urban traffic system.

CCS CONCEPTS

•Computing methodologies →Discrete-event simulation; Simulation tools;

KEYWORDS

State Graph; State Transition Table; Discrete-Event System; Tabular Specification

ACM Reference format:

. 2017. State-based Modeling and Simulation of Discrete-Event Systems using Graphical and Tabular Specifications. In *Proceedings of ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Singapore, May 2017 (PADS'17),* 4 pages.

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A state-based modeling formalism (SBMF) describes the dynamics of a discrete-event system (DES) in terms of the *states* of the resources that reside in the system. The SBMF is originated from the classical *finite state machine* (FSM) that was used for modeling the behavior of sequential circuits [9]. Since then, there have been a number of SBMFs proposed in the literature, as well as a number of simulation packages that follow the SBMFs developed. Among the SBMFs, DEVS [15] and timed automata [1] are regarded as important parts of the theory of modeling, and each forms an independent research community. *Timed automata* is a FSM extended with a finite set of real-valued clocks where two types of clock constraints (*guard* and *invariant*) are placed on a state transition and a state node [4]. Also, *DEVS* (Discrete Event System Specification) is a FSM extended with a *lifepan* for each state in order to accommodate the timing and hierarchical modeling concept [15]. These SBMFs allow to build hierarchical and modular models for the components that compose of a system where the *atomic model* captures the actual behavior of a component and the *coupled model* contains other components.

A number of simulation packages have been introduced to support the state-based modeling and simulation of a DES. Especially, DEVS-based simulation packages have been widely developed, such as ADEVS[11], CD++ [14], PythonPDEVS[13], and so on. These simulation packages require the users to program in order to define the simulation models, which is a difficult task for the modelers who are not programming experts. To mitigate this difficulty, a few simulation packages are introduced with the graphical modeling capability: CoSMos [6] and CD++ Builder[2], and MS4 ME [12].

In the SBMFs, the *state transition diagram* is widely used to specify the atomic models graphically. However, it has a difficulty in having a concise and readable representation when it deals with the dynamic and complex behavior that results in quite a number of states and state transitions. Therefore, another specification, *state transition table*, is more suitable to specify the behavior in a tabular form showing which state will move to, based on the current state and the inputs. No simulation package among the aforementioned simulation packages fully supports the tabular specification.

This paper will present a state-based modeling and simulation environment, which is named *state graph simulator* (SGS) that supports both graphical and tabular modeling so that the user can specify the models in a concise and readable way. In the following sections, Section 2 describes a state-based modeling formalism, *state graph*. Section 3 introduces the SGS briefly. Then, Sections 4 and 5 illustrate the state-based modeling and simulation using the SGS, respectively. At last, Section 6 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 [5]. 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*.

Located in the upper part of Figure 1 is the object interaction diagram that describes the interactions between the resources as message exchanges. Located in the lower part of Figure 1 are the state transition diagrams that describe the state transitions of each resource based on the interactions with other resources using the

PADS'17, Singapore

DOI: 10.1145/nnnnnnn.nnnnnnn

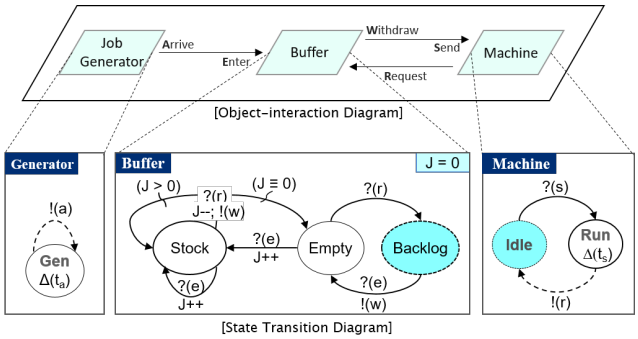

Figure 1: State graph model of a single server system [5]

graphical notations, where each node represents a state that the resource can have and each edge indicates the transition to a state upon the receipt of a message or upon a time constraint.

`Job Generator` sends out an *Arrive* message {`!(a)`} every t_a time units in a `Gen` state. `Buffer` has three states of `Backlog`, `Empty`, and `Stock`, and a state variable `J` that represents the number of jobs waiting at `Buffer`. Initially, `Buffer` is set to `Backlog` state and its state variable `J` is set to zero. When `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. `Machine` starts with `Idle` state. Upon receiving a *Send* message {`?(s)`}, `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)`}.

As a modeling formalism for a DES, state graph supports not only the graphical specification as presented in Figure 1, but also tabular and algebraic 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 [7]. Tables 1 presents a *state transition table* of `Buffer` atomic state graph model given in Figure 1.

Table 1: State transition table: `Buffer` atomic model

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)	J--; !(w)	J ≡ 0	-	Empty
				J >0	-	Stock

The graphical and tabular specifications contain all the information for the atomic state graph models. Therefore, they are interchangeable. However, as a target system becomes larger and more complex, the number of graphical elements in the state transition diagram also increases, which runs into difficulties in building and understanding the models. Furthermore, different types of actions (e.g. entry action, input action, transition action) supported in the atomic state graph model may have difficulty in specifying the state transitions including the actions and conditions with the state transition diagram. On the other hand, the state transition table is more suitable for capturing the complex behvaiors by providing a concise and readable representation. Table 2 presents an *object interaction table* of the composite state graph model given in Figure 1 where each row represents the interaction from a source object to a receiving object.

Table 2: Object interaction table

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

State graph simulator (SGS) is a state-based modeling and simulation toolkit that supports the state graph modeling formalism. As depicted in Figure 2, SGS provides a modeling capability of state graph models in both graphical and tabular specifications with *object interaction diagram editor* for specifying the composite state graph model in both graphical and tabular specifications and *state transition table editor* for specifying the atomic state graph model in the tabular specification. SGS provides a simulation capability of state graph models: *simulation window* for scaled real-time and as-fast-as simulation executions and simple model verification with graphical elements, *output window* for viewing the simulation outputs with state-time charts for each atomic model, *sequence diagram window* for the model verification regarding the state transitions along with the message exchanges.

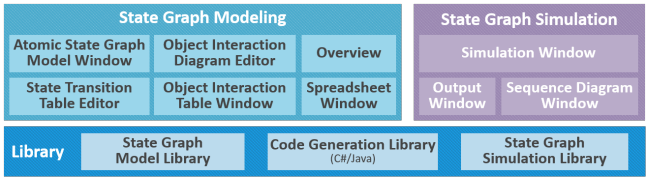

Figure 2: System architecture of state graph simulator

SGS supports the rapid development of a domain-specific simulator by providing the *code generator* and the *state graph simulation library*. Once the state graph model is constructed and verified in SGS, the code generator will produce the source code of the state graph model in Java or C#. Then, the software developer can build a domain-specific simulator with customized input editor and output viewer. The state graph simulation library consists of a simulation engine and supportive classes for the data collection.

4 STATE GRAPH MODELING USING SGS

In this section, state graph modeling in the SGS will be illustrated with the signalized urban traffic system. The basic modeling concept of the signalized urban traffic system using state graph is first introduced in [8], and then it is elaborated in [10]. The signalized urban traffic system is a physical road network with the traffic

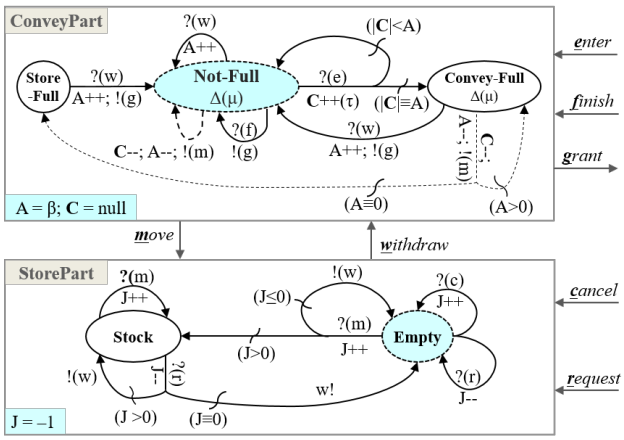

Figure 3: `ConveyPart` **and** `StorePart` **atomic models**

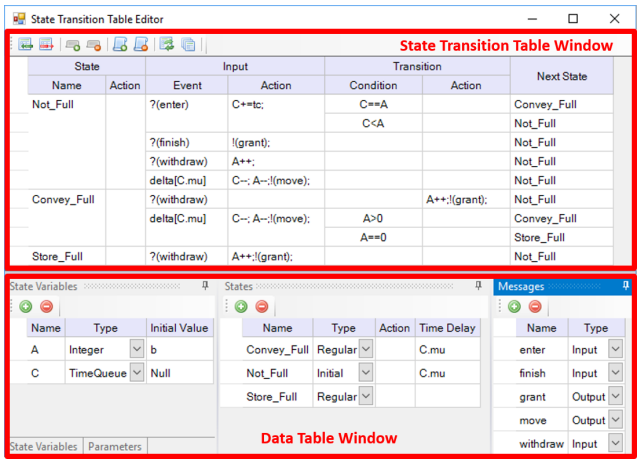

Figure 4: State transition table editor

controllers at each intersection, which can be represented as a directed graph. A *link* in the directed graph represents a road lane where inflow occurs only at the start point of a link and outflow occurs only at the end point of a link. A *node* in the directed graph connects the incoming and outgoing 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), or the vehicle generation and disposal (`Source` and `Sink` nodes).

In [10], each link is represented by a pair of two atomic state graph models: `ConveyPart` for moving the vehicles from the link's start point to its end point and `StorePart` for storing the vehicles arriving at the end point and passing them to the out-flow links according to the traffic signal. Each type of nodes is represented by a respective atomic state graph model, e.g. `Source` and `Sink` atomic state graph models for `Source` and `Sink` nodes and `Diverter` and `Merge Controller` for `Split` and `Merge` nodes.

Figure 3 depicts the state trandition diagrams of `ConveyPart` and `StorePart` atomic state graph models that compose a link. Presented in Figure 4 is the state transition table (STT) editor with `ConveyPart` atomic state graph model in SGS. STT Editor consists of windows: *STT window* for editing the state transition table and *data table window* for defining state variables, parameters, input/ouptut messages, and states in a spreadsheet style. In STT window, the user can interactively manipulate the state transition table using the tools located at the top of the window, such as adding/removing a state transition, or adding/removing an input.

Presented in Figure 5 is the composite state graph model of a traffic network with four intersections where each road consists of one lane in the object interaction diagram editor. Located at the left of Figure 5 is the *atomic graph model window* where the atomic models are stored for the model reuse. Each atomic model can be instantiated as an *atomic object* into the object interaction diagram by a drag-and-drop way. The top-right of Figure 5 is the *object interaction diagram editor* having 514 atomic objects and 1464 messages connecting the atomic objects. The *object interaction table window* located at the bottom-left of Figure 5 presents the object interaction table of the composite state graph model. Located at the bottom-right of Figure 5 is the *spreadsheet window* where the user can change the values of parameters and state variables of the atomic objects having the same type of the atomic model. The object interaction diagram shown in Figure 5 only represents the road network system where the traffic controls at four intersections are distributed in other four layers located next to the layer, named *Road Network System*. The layers are useful in modeling a large and complex system. More details on the state graph model of the signalized urban traffic system can be found in [10].

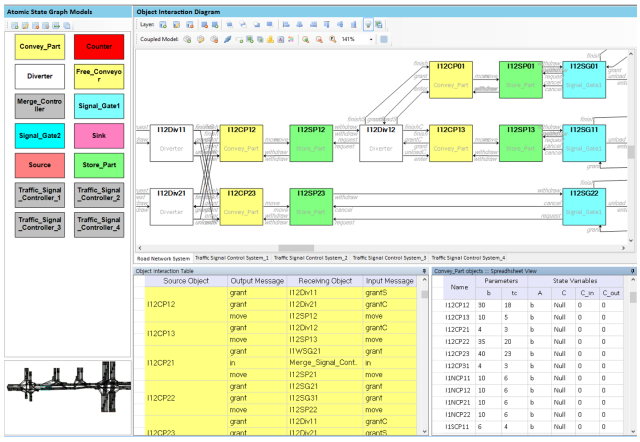

Figure 5: Object interaction diagram editor

5 STATE GRAPH SIMULATION USING SGS

5.1 Simulation

As a modeling formalism for a DES, the state graph provides a simulation algorithm, named *synchronization algorithm*, which 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 [5]. As presented in Figure 6, a *simulation window* takes care of the simulation execution of the state graph model with visual

elements, such as *Label*, *Picture Set*, and *Gauge*. Each visual element represents the value of a state variable or the state of an atomic object that changes as the simulation proceeds. In Figure 6, two labels are placed to track the number of cars entered from the west and exited to the west, which project the state variables `In` and `Out` of `Counter_W` atomic object, respectively.

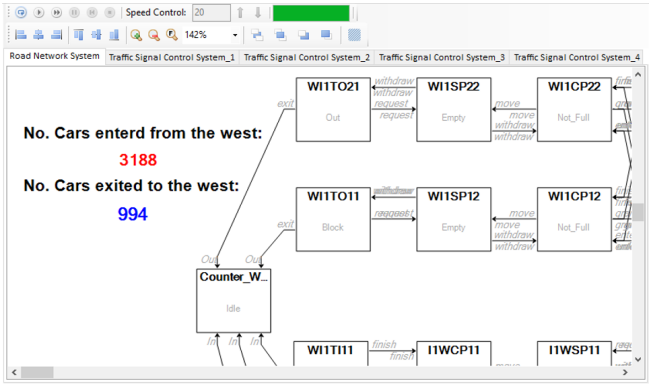

Figure 6: Simulation window

5.2 Output Analysis and Model Verification

The output analysis and model verification are based on the automatic data collection to provide the system trajectories and the sequence diagram. As presented in Figure 7-(a), SGS provides the `Output Windows` for each atomic object, each of which consists of three outputs: (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, (3) *state variable output* for displaying the value changes of each state variable over time with its minimum, mean, and maximum values. The state output and state variable output can be used to collect performance measures, such as machine utilization, average queue length, average waiting time, and so on.

To minimize the errors in the simulation execution, SGS provides the syntactical model validation to find out the syntax errors in the state transition tables and object interaction diagram. However, the logical errors in a state graph model cannot easily be found before the simulation run. Therefore, SGS provides the model verification using the sequence diagram of the Unified Modeling Language (UML), which is suitable to specify the dynamic behaviors of software components. The sequence diagram is a graphical specification language to show the interactions between software components in the sequential order that hose interactions occur [3]. As presented in Figure 7-(b), the *sequence diagram window* of SGS provides the sequence diagram to illustrate the state transitions of selected atomic objects along with the message exchanges among the atomic objects within specific time periods.

6 CONCLUSION

The state graph simulator provides all the functionalities for the modeling and simulation development process of state graph modeling formalism with the support of graphical and tabular modeling,

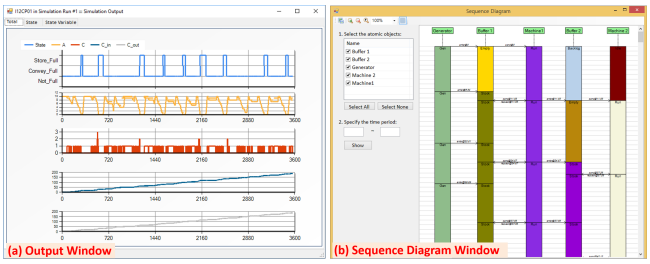

Figure 7: Output window and Sequence diagram window

interactive simulation and output analysis, and model verification. the tabular modeling takes advantage of having a concise and clear representation and faciliates the rapid model development without the expertise in the programming language.

Further research should extend the SGS so as to cope with a large-scale DES, including the dual view of atomic modeling, and enhanced visual and automatic model verification. Also, modeling templates will be introduced to help the reuse of atomic models and to provide some primitives for the advanced data collection.

REFERENCES

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

[2] M. Bonaventura, G. A. Wainer, and R. Castro. 2011. Graphical Modeling and Simulation of Discrete-Event Systems with CD++ Builder. *SIMULATION* 89, 1 (2011), 4–27.

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

[4] C.G. Cassandras and S. Lafortune. 2010. *Introduction to Discrete Event Systems* (2nd ed.). Springer.

[5] Byoung K. Choi and D. Kang. 2013. *Modeling and Simulation of Discrete Event Systems.* John Wiley & Sons.

[6] M. D. Fard and H. S. Sarjoughian. 2015. Visual and Persistence Behavior Modeling for DEVS in CoSMoS. In *Proceedings of the 2015 Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium.*

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

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

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

[10] M. Myung, 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.*

[11] J. Nutaro. 2017. ADEVS (A Discrete EVent System simulator) C++ library. accessed 1 February 2017. (February 2017). http://web.ornl.gov/~1qn/adevs/

[12] C. Seo, B. P. Zeigler, R. Coop, and D. Kim. 2013. DEVS Modeling and Simulation Methodology with MS4 Me Software Tool. In *Proceedings of the 2013 Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium.*

[13] Y. V Tendeloo and H. Vangheluwe. 2017. An evaluation of DEVS simulation tools. *SIMULATION* 93, 2 (2017), 103–121. DOI:http://dx.doi.org/10.1177/0037549716678330

[14] G. A. Wainer. 2002. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience* 32, 13 (2002), 1261–1306.

[15] B. P. Zeigler. 1976. *Theory of Modeling and Simulation.* John Wiley & Sons.