Processing Time Bounds of Schedule-Preserving DEVS *

Moon Ho Hwang (mhhwang@ece.arizona.edu)
Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ 85721, USA

Su Kyeong Cho (sc211@daimlerchrysler.com)
Digital Product Development, DaimlerChrysler, Auburn Hills, MI 48326, USA

Bernard P. Zeigler (zeigler@ece.arizona.edu)
Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ 85721, USA

Feng Lin (flin@ece.eng.wayne.edu)
Dept. of Electrical & Computer Engineering, Wayne State University, Detroit, MI 48202, USA

Jul. 12, 2007

Abstract. This paper proposes a class of discrete event system specification (DEVS), called schedule-preserving DEVS (SP-DEVS), in which the external transition cannot change its next event schedule. This restriction to DEVS is designed for making treatment of timed behavior decidable when dealing with SP-DEVS networks. As a result, it is possible for us to compute processing time bounds of event sequences in the behavior SP-DEVS networks.

Keywords: DEVS, Timed Behavior Verification, Time-Line Abstraction, Reachability Graph, Processing Time Bounds

1. Introduction

This paper shows how to compute the minimum and maximum time bounds of event sequences that are in the behavior set of a discrete event dynamic system (DEDS). The formalism describing DEDS is the discrete event system specification (DEVS) which was invented by Zeigler (Zeigler, 1976), and has evolved to support hierarchical and modular modeling (Zeigler, 1984).

The technique we are using here to identify the performance bounds is not a sampling-based method such as *simulation* which is one of most popular methods of analyzing DEVS models (Ho, 1993; Zeigler et al., 2000), but is instead an exhaustive search method, usually called *verification* (Batt et al., 2006). To verify the performance bounds of event

* Correspondence: Moon Ho Hwang, E-Mail: mhhwang@ece.arizona.edu, Tel: +1+520-626-4737, Fax: +1+520-621-3862, Mailing Address: Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ 85721, USA

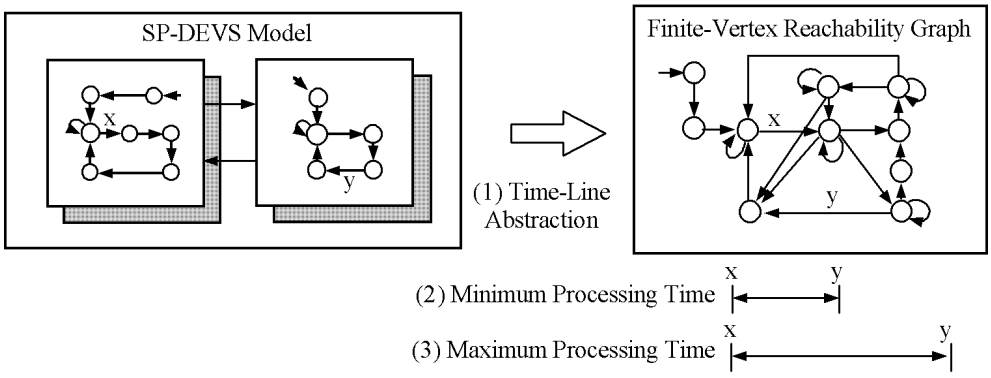

Figure 1. Main Procedure

sequences associated with a given DEVS model, we need to construct a finite-state reachability graph representing the behavior of the given DEVS model. For this reason, we use a sub-class of DEVS, rather than the classic DEVS (Zeigler et al., 2000) because the classic DEVS needs an infinite-state reachability graph to describe its behavior (Hwang and Zeigler, 2006a). The sub-class of DEVS used here is called a schedule-preserving DEVS (SP-DEVS), which restricts DEVS' expressiveness so that an input event cannot change the internal schedule. This allows us to obtain a finite-state reachability graph.

Figure 1 illustrates the main procedure we will go through for the performance bound verification. The procedure consists of three-steps: (1) constructing a finite-state reachability graph for a given SP-DEVS; (2) evaluating the minimum processing time bound, and (3) evaluating the maximum processing time bound. To be successful in implementing such a procedure, the following questions arise.

Q1. Can we construct a finite-state reachability graph that represents the timed-behavior of a given SP-DEVS model? How complex is this construction process?

Q2. Can we determine the minimum time span between two events? How complex is the evaluation process

Q3. Can we determine the maximum time span between two events? How complex is the evaluation process?

To answer these questions, this paper is organized as follows. Section 2 introduces the SP-DEVS formalism that will provide the foundation for of the whole procedure. The terms atomic class and coupled class are introduced along with their behaviors as sets of executable traces. Section 3 proposes a time abstraction technique, called the time-line

abstraction, of the SP-DEVS class. In addition, the completeness and the complexity of the proposed time abstraction are addressed in this section. Section 4 presents a time evaluation theory by which we will compute the minimum and maximum processing time bounds using a reachability graph of SP-DEVS. To do this, certain cost evaluation techniques along a given path of the reachability graph will be introduced in this section. Section 5 shows experimental results in two cases: a crosswalk signal controller and a monorail system. Section 6 gives a summary, notes contributions, and suggests future research directions.

2. SP-DEVS

The DEVS formalism provides a hierarchical and modular modeling structure (Zeigler et al., 2000). There are two building blocks in the formalism: the Atomic DEVS class and the Coupled DEVS class. Like DEVS, SP-DEVS has two such classes. This chapter will define the two classes of SP-DEVS, in terms of both structure and behavior.

2.1. Atomic SP-DEVS

2.1.1. *Structure of Atomic SP-DEVS*

An *atomic SP-DEVS* is defined by

$$A =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$$

where

- X is a finite set of *input events*.
- Y is a finite set of *output events*.
- S is a finite set of *states*.
- s_0 is the initial state.
- $\tau : S \rightarrow \mathbb{Q}_{[0,\infty]}$ is the *time advance function* where $\mathbb{Q}_{[0,\infty]}$ is the set of non-negative rational numbers plus infinity. This function is used to determine the lifespan of a state.
- $\delta_x : S \times X \rightarrow S$ is the *external state transition function* that defines how an input event changes a state.
- $\delta_y : S \rightarrow Y^{\phi} \times S$ is the *output and internal state transition function* where $Y^{\phi} = Y \cup \{\phi\}$ and $\phi \notin Y$ denotes the *silent event*. This output and internal state transition function defines how a state

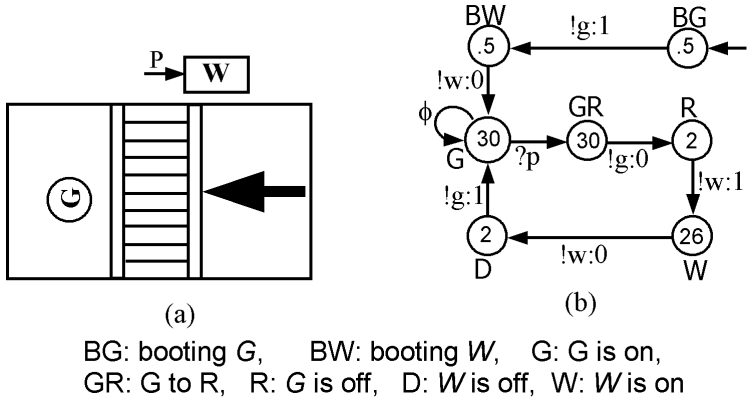

Figure 2. Crosswalk System (a) Configuration (b) Crosswalk Controller (CC)

generates an output event and, at the same time, how it changes the state internally. This function can be invoked when a state's lifespan expires (as determined by τ), and the system thereupon transitions into a new state. [1] ■

EXAMPLE 1 (Atomic SP-DEVS Model for Crosswalk Controller).

Let's consider a controller for a crosswalk light system shown in Figure 2(a). In the system, there are two traffic lights, green(G) and walk(W), facing 90 degrees from each other, with G aligned along a road, and W aligned perpendicular to the road. Initially, G is on and W is off, (for safety, they should not be on at the same time, but may both be off at the same time). Time for booting the system will be 1 second. As long as there is no waiting pedestrian, G and W stay on and off, respectively.

If a pedestrian pushes a button to cross the road, G turns off within 30 seconds and then, W turns on 2 seconds later. Similarly, 26 seconds after W turns on, W returns to off, and G goes back to on 2 seconds after W turns off. This cycle repeats.

We can model the push button event as an input event `?p` and the changing of lights as output events such that `!g:1` for turn-on of G, `!w:0` for turn-off of W, etc [2]. A SP-DEVS model, `CC`, for the crosswalk controller is as follows. $A_{\texttt{CC}} =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$ such that X={`?p`}; Y={`!g:0`, `!g:1`,`!w:0`,`!w:1`}; S={`BG`, `BW`, `G`, `GR`, `R`, `W`, `D`}, where the state definitions are defined as shown in Figure 2; s_0=`BG`;

[1] δ_y can be split into two functions: the output function $\lambda : S \rightarrow Y$ and the internal state transition function $\delta_{int} : S \rightarrow S$ as in (Zeigler et al., 2000).

[2] To distinguish event types, this paper uses '?' before an input event, and '!' before an output event

$$\tau(s) = \begin{cases} 0.5 \text{ if } s = \texttt{BG};\ 0.5 \text{ if } s = \texttt{BW};\ 30 \text{ if } s = \texttt{G};\ 30 \text{ if } s = \texttt{GR}; \\ 2 \quad \text{if } s = \texttt{R};\quad 26 \quad \text{if } s = \texttt{W};\quad 2 \quad \text{if } s = \texttt{D}. \end{cases}$$

$$\delta_x(s, x) = \begin{cases} \texttt{GR} & \text{if } s = \texttt{G} \wedge x = \texttt{?p} \\ s & \text{otherwise.} \end{cases}$$

$$\delta_y(s) = \begin{cases} (\texttt{!g:1}, \texttt{BW}) \text{ if } s = \texttt{BG};\ (\texttt{!w:0}, \texttt{G}) \text{ if } s = \texttt{BW}; \\ (\phi, \texttt{G}) \quad \text{if } s = \texttt{G};\ (\texttt{!g:0}, \texttt{R}) \text{ if } s = \texttt{GR}; \\ (\texttt{!w:1}, \texttt{W}) \quad \text{if } s = \texttt{R};\ (\texttt{!w:0}, \texttt{D}) \quad \text{if } s = \texttt{W}; \\ (\texttt{!g:1}, \texttt{G}) \quad \text{if } s = \texttt{D}. \end{cases}$$

□

Figure 2(b) illustrates an atomic SP-DEVS model for CC in which a circle denotes the time-span of a state s given by $\tau(s)$, and a directed arc indicates a state transition either caused by an input event or accompanied by an output event.

For simplicity, an input event that doesn't make any change of state is omitted in the state transition diagram. For this reason, we don't have to draw arcs of $\delta_x(s, \texttt{?p}) = s$ for any $s \neq \texttt{G}$ in Figure 2(b).

2.1.2. *Behavior of Atomic SP-DEVS*

Given an atomic SP-DEVS $A =< X, Y, S, \tau, \delta_x, \delta_y >$, the *total event set* is defined

$$Z = X \cup Y \cup \{\phi\}$$

as all input events of X and all output events of Y as well as the silent event. For example, the total event set of the CC model in Figure 2 is Z={?p, !g:0, !g:1, !w:0, !w:1, ϕ }.

The *time base* of SP-DEVS, denoted by $\mathbb{T} = [0, \infty)$, is the set of non-negative real numbers. The *elapsed time* of a state s, denoted by $t_e \in \mathbb{T}$ is continuously increasing from 0 to a lifespan of a state s. Let $\mathbb{T}^\infty = \mathbb{T} \cup \{\infty\}$. Then, a function $tr : \mathbb{T}^\infty \to 2^{\mathbb{T}}$ returns a time range to which the elapsed time t_e can belong depending on $t \in \mathbb{T}^\infty$ such that

$$tr(t) = \begin{cases} [0, t] & \text{if } t < \infty \\ [0, \infty) & \text{otherwise} \end{cases}$$

where '[' and ']' are closed boundaries, while ')' is an open boundary. Thus, if $t = \infty$, t_e cannot reach t.

Based on the tr function, let the *legal state set* $Q_{\texttt{p}}$ be

$$Q_{\texttt{p}} = \{(s, t_s, t_e) | s \in S, t_s \in \mathbb{Q}_{[0,\infty]}, t_e \in tr(t_s)\}.$$

For example, if CC of Figure 2(b) has a legal state q=(W, 26, 4), it means CC has been staying at the state W for 4 seconds, and the remaining time

to change to the next state internally is 22 (=26-4). The *illegal state set* is denoted by Q_{imp} and is defined by

$$Q_{\mathsf{imp}} = \{(\mathsf{imp}, \infty, t_e) | t_e \in tr(\infty)\}$$

where $\mathsf{imp} \notin S$ such that $Q_{\mathsf{p}} \cap Q_{\mathsf{imp}} = \varnothing$. Then, the *total state set* Q is defined by

$$Q = Q_{\mathsf{p}} \cup Q_{\mathsf{imp}}.$$

Based on the total state, Q, and the total event set, Z, we can define the complete state transition dynamics. The *total state function* $\delta : Q \times Z \to Q$ defines state transitions over all combinations of states and events such that for $q = (s, t_s, t_e) \in Q$ and $z \in Z$

$$\delta(q, z) = \begin{cases} (s', t_s, t_e) & \text{if } q \in Q_{\mathsf{p}}, z \in X, \delta_x(s, z) = s' \\ (s', \tau(s'), 0) & \text{if } q \in Q_{\mathsf{p}}, z \in Y^{\phi}, t_e = t_s, \delta_y(s) = (z, s') \\ (\mathsf{imp}, \infty, t_e) & \text{otherwise} \end{cases} \quad (1)$$

The first statement of Equation (1) is the state transition caused by an input event. In this case, the lifespan and the elapsed time are preserved. The second statement of Equation (1) shows the state transition with a resulting output in which the lifespan and the elapsed time are reset. Any other conditions make the transition reach or stay at the imp state. Observe that once an atomic SP-DEVS model reaches the imp state, it cannot transit out of it. That property gives us an implication of the behavior of the atomic DEVS that will be defined in Equation (3).

Let's try to understand Equation (1) by using `CC` in Example 1. Suppose that there is an push button event `?p` when the total state is $q = (\mathtt{G}, 30, t_e)$ where $t_e \in [0, 30]$. Then the next total state is determined by $\delta((\mathtt{G}, 30, t_e), \mathtt{?p}) = (\mathtt{GR}, 30, t_e)$ as the first condition of Equation (1). On the other hand, when the state is `W` and its elapsed time reaches the lifespan 26, then the output and the next state can be described by $\delta((\mathtt{W}, 26, 26), \mathtt{!w:0}) = (\mathtt{D}, 2, 0)$ as the second condition of Equation (1). Since it is impossible to execute an output and internal state transition when $t_e \neq t_s$ $\delta((\mathtt{W}, 26, 24), \mathtt{!w:0}) = (\mathsf{imp}, \infty, 24)$ as the last condition of Equation (1).

Based on Equation (1), a state trajectory can be defined over a sequence of timed events. A *timed event* is a pair of an event $z \in Z$ and its occurrence time $t \in \mathbb{T}$ thus it is denoted as (z, t). The *concatenation* of two events (z_1, t_1) and (z_2, t_2) is denoted by $(z_1, t_1)(z_2, t_2)$ and can be defined when $t_1 \leq t_2$. The *identity* of the concatenation operation is the *null-event*, denoted by ϵ. The null-event with a time interval $[t_l, t_u] \in T$ is denoted by $\epsilon_{[t_l, t_u]}$ indicating that there is no

event between t_l and t_u. Given an event set Z and a time interval $[t_l, t_u] \subseteq \mathbb{T}$, the set of *total event sequences* is denoted by $\Omega_{[t_l,t_u]}$ and is defined by $\Omega_{[t_l,t_u]} = (Z \times [t_l, t_u])^*$ which is the set of all possible concatenations of timed events (plus $\epsilon_{[t_l,t_u]}$) over Z in $[t_l, t_u]$. Given $\Omega_{[t_l,t_u]}$ and $t_l < t_1 < t_2 < t_u$, $\omega = (z_1, t_1)(z_2, t_2) \in \Omega_{[t_l,t_u]}$ is equivalent to $\omega = \epsilon_{[t_l,t_1]}(z_1, t_1)\epsilon_{[t_1,t_2]}(z_2, t_2)\epsilon_{[t_2,t_u]}$. For simplicity, this paper will use the former rather than the latter.

A *state trajectory* is defined by the function $\Delta : Q \times \Omega_{[t_l,t_u]} \to Q$ such that for $q = (s, t_s, t_e) \in Q$ at time t_l, $\omega, \omega' \in \Omega_{[t_l,t_u]}$ and $z \in Z$,

$$\Delta(q, \omega) = \begin{cases} (s, t_s, t_e + t_u - t_l) & \text{for } \omega = \epsilon_{[t_l,t_u]} \\ \delta(\Delta(q, \omega'), z) & \text{for } \omega = \omega'(z, t_u) \end{cases} \tag{2}$$

Equation (2) says that in the time interval $[t_l, t_u]$ of the null-event sequence, the only thing that can change is the elapsed time $t'_e = t_e + t_u - t_l$. At the time t_u of an event z, the next total state is determined by the total state transition function δ which is defined in Equation (1). We can apply these steps recursively.

Based on the state trajectory function, the behavior of a given atomic SP-DEVS $A =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$ is defined as the all possible event sequences which don't enter to a illegal state. Formally, let $q_0 = (s_0, \tau(s_0), 0)$. The *behavior* of A, or the *language* of A, denoted by $L(A)$, is

$$L(A) = \{\omega \in \Omega_{[0,\infty)} | \Delta(q_0, \omega) \in Q_{\mathtt{p}}\}. \tag{3}$$

Figure 3 illustrates a state trajectory associated with an event sequence of the atomic SP-DEVS `CC` introduced in Example 1 where the associated event sequence is $\omega_{[0,\infty)}$=(`!g:1`,0.5)(`!w:0`,1.0) (ϕ, 31)(`?p`, 47)(`!g:0`,61)(`!w:1`,63)(`?p`, 70) (`!w:0`,89)(`!g:1`,91)(ϕ,121)(ϕ,151) ... which is an element of $L(A_{\mathtt{CC}})$ because $\Delta((\mathtt{BG}, 0.5, 0), \omega_{[0,\infty)})$=($\mathtt{G}$,30,$t_e$) $\in Q_{\mathtt{p}}$.

In the meantime, if $\omega_{[0,\infty)} = \omega'_{[0,2]}\omega''_{[2,\infty)}$ where $\omega'_{[0,2]}$ = (`!g:1`, 0.5) (`!w:0`, 2.0) and $\omega''_{[2,\infty)}$ = ($!\phi$, 31) ..., ω is not an element of $L(A_{\mathtt{CC}})$ because $\Delta((\mathtt{BG}, 0.5, 0), \omega_{[0,\infty)})$=$\Delta(\Delta((\mathtt{BG}, 0.5, 0), \omega'_{[0,2]}), \omega''_{[2,\infty)})$ = $\Delta((\mathsf{imp}, 1.0, 0.5), \omega''_{[2,\infty)}) = (\mathsf{imp}, \infty, t_e) \in Q_{\mathsf{imp}}$ where $t_e \in tr(\infty)$.

2.2. Coupled SP-DEVS

2.2.1. *Structure of Coupled SP-DEVS*

The coupled SP-DEVS provides the hierarchical and modular structure necessary to describe system networks. Formally, a coupled SP-DEVS is defined by

$$N =< X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select >$$

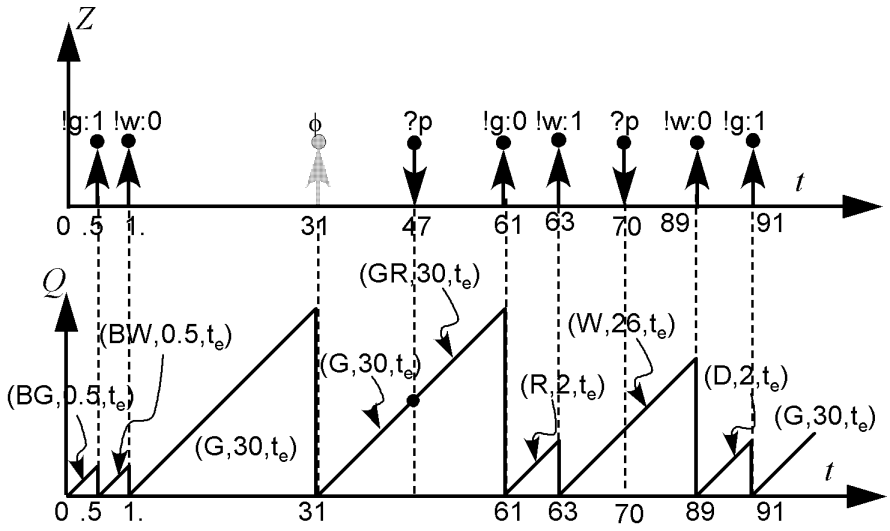

Figure 3. Trajectory of Crosswalk Controller

where

- X is a finite set of *input events*.
- Y is a finite set of *output events*.
- D is a finite set of *names of sub-components*
- $\{M_i\}$ is an index set of *SP-DEVS models* where $i \in D$. M_i can be either an atomic SP-DEVS model or a coupled SP-DEVS model.
- $C_{xx} : X \rightarrow \bigcup_{i \in D} X_i$ is an *external input coupling function*, where X_i is the set of input events of sub-component $i \in D$.
- $C_{yx} : \bigcup_{i \in D} Y_i \rightarrow \bigcup_{i \in D} X_i$ is an *internal coupling function*, where Y_i is the set of output events of sub-component $i \in D$.
- $C_{yy} : \bigcup_{i \in D} Y_i \rightarrow Y^{\phi}$ is an *external output coupling function*.
- $Select : 2^D \rightarrow D$ is a *tie-breaking function* to arbitrate the occurrence of simultaneous events. ■

EXAMPLE 2 (Coupled SP-DEVS Model for Crosswalk Controller).

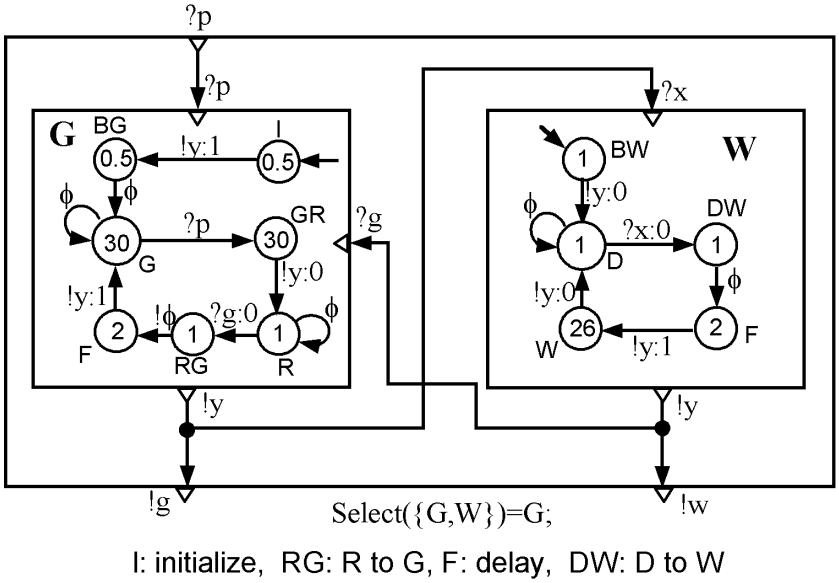

Figure 4. Crosswalk Controller built as an SP-DEVS Network

Figure 4 shows another crosswalk controller, CC, which is built as a coupled SP-DEVS model such that $N_{\mathtt{CC}} =< X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select >$ where X={?p}; Y={!g:0, !g:1, !w:0, !w:1}; D={G, W}; $M_{\mathtt{G}}$ and $M_{\mathtt{W}}$ are atomic SP-DEVS whose transition diagrams are shown in Figure 4; $C_{xx}(\mathtt{?p_{CC}})=\mathtt{?p_G}$ where $\mathtt{?p_{CC}}$ indicates ?p of the coupled model CC, while $\mathtt{?p_G}$ stands for ?p of the sub-component G; $C_{yx}(\mathtt{!y:0_G})=\mathtt{?x:0_W}$, $C_{yx}(\mathtt{!y:1_G})=\mathtt{?x:1_W}$, $C_{yx}(\mathtt{!y:0_W})=\mathtt{?g:0_G}$, $C_{yx}(\mathtt{!y:1_W})=\mathtt{?g:1_G}$; $C_{yy}(\mathtt{!y:0_G})=\mathtt{!g:0_{CC}}$, $C_{yy}(\mathtt{!y:1_G})=\mathtt{!g:1_{CC}}$, $C_{yy}(\mathtt{!y:0_W})=\mathtt{!w:0_{CC}}$, $C_{yy}(\mathtt{!y:1_W})=\mathtt{!w:1_{CC}}$; $Select(\{\mathtt{G,W}\})=\mathtt{G}$.

□

Practically, we can see an event as a pair of (*port*, *value*) and the coupling as a pair of ($port_{source}$, $port_{destination}$) (Zeigler, 1990; Zeigler et al., 2000). The basic assumption of such *port coupling* is that the value of $port_{source}$ is casted to that of $port_{destination}$. We can find that realistic examples of the port coupling in the VHDL language (Skahill, 1996) and the language of programmable logic controller (PLC) (Lewis, 1998). For convenience, we will use the port coupling view as shown in Figure 4 throughout this paper.

2.2.2. *Behavior of Coupled SP-DEVS*
Given an SP-DEVS network, $N =< X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select >$, we can associate with it a basic DEVS, called the *resultant*, given by A_N and defined by

$$A_N =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$$

where

$$S = \underset{i \in D}{\times} Q_{\mathsf{p},i} \tag{4}$$

where $Q_{\mathsf{p},i}$ denotes the legal state set of component i. The initial state $s_0 = (\ldots, (s_{0i}, \tau_i(s_{0i}), 0), \ldots)$ for each $i \in D$.

Let s be a state of A_N such that $s = (\ldots, (s_i, t_{si}, t_{ei}), \ldots) \in S$.

Then the time advance function $\tau : S \to \mathbb{T}^\infty$ is defined by

$$\tau(s) = \min\{t_{ri} | t_{ri} = t_{si} - t_{ei}, i \in D\} \tag{5}$$

where t_{ri} is the *time remaining until the internal state transitions in component i*.

The external state transition $\delta_x : S \times X \to S$ is defined such that for $x \in X$,

$$\delta_x(s, x) = s' = (\ldots, (s'_i, t'_{si}, t'_{ei}), \ldots)$$

where any $i \in D$

$$(s'_i, t'_{si}, t'_{ei}) = \begin{cases} (\delta_{x,i}(s_i, x_i), t_{si}, t_{ei}) & \text{if } C_{xx}(x) = x_i \\ (s_i, t_{si}, t_{ei}) & \text{otherwise.} \end{cases} \tag{6}$$

For any state, s, let us define the set of *imminents* of s, $IMM(s) = \{i | i \in D, t_{ri} = \tau(s)\}$. Imminents is the subset of names of components that have minimum remaining time t_{ri}, i.e., they are the names of candidate components which are in states where the next output and internal state transition may occur. Let $i^* = Select(IMM(s))$. This i^* is the name of the component whose output and internal state transition function will be executed because it was selected using the tie-breaking function *Select*.

Based on the function *Select*, the output and internal state transition function $\delta_y : S \to Y^\phi \times S$ is defined by

$$\delta_y(s) = (y, s') = (C_{yy}(y_{i^*}), (\ldots, (s'_i, t'_{si}, t'_{ei}), \ldots))$$

where

$$(s'_i, t'_{si}, t'_{ei}) = \begin{cases} (s'_i, \tau(s'_i), 0) & \text{if } i = i^*, \delta_{y,i^*}(s_{i^*}) = (y_{i^*}, s'_{i^*}) \\ (\delta_{x,i}(s_i, x_i), t_{si}, t_{ei}) & \text{if } C_{yx}(y_{i^*}) = x_i \\ (s_i, t_{si}, t_{ei}) & \text{otherwise} \end{cases} \tag{7}$$

By using the crosswalk controller CC in Example 2, let's try to understand Equations (4) to (7). At the beginning, CC's initial state is $s_0 = ((\mathtt{I}, 0.5, 0)_{\mathtt{G}}, (\mathtt{BW}, 1, 0)_{\mathtt{W}})$. Now suppose that the current state of CC is $s = ((\mathtt{G}, 30, 5.65)_{\mathtt{G}}, (\mathtt{D}, 1, 0.65)_{\mathtt{W}})$. Then, its time advance value $\tau(s) = \min\{24.35 = 30 - 5.65, 0.35 = 1 - 0.65\} = 0.35$, and the imminent set is $IMM(s) = \{\mathtt{W}\}$.

If there is input event ?p when the state of CC is $((\mathtt{G}, 30, 5.65)_{\mathtt{G}}, (\mathtt{D}, 1, 0.65)_{\mathtt{W}})$, the next state of CC is $\delta_x(s, \mathtt{?p}) = (\delta_{x_{\mathtt{G}}}(\mathtt{G}, \mathtt{?p}), (\mathtt{D}, 1, 0.65)_{\mathtt{W}}) = ((\mathtt{GR}, 30, 5.65)_{\mathtt{G}}, (\mathtt{D}, 1, 0.65)_{\mathtt{W}})$.

If the current state of CC is $s = ((\mathtt{G}, 30, 6)_{\mathtt{G}}, (\mathtt{D}, 1, 1)_{\mathtt{W}})$, then the W component of S has timed out which generates an output and internal state transition of $\delta_y(s) = (\phi, (\mathtt{G}, 30, 6)_{\mathtt{G}}, (\mathtt{D}, 1, 0)_{\mathtt{W}})$.

Let's assume that both G and W are ready to execute their own output and internal state transitions such as when the current state of CC is $s = ((\mathtt{GR}, 30, 30)_{\mathtt{G}}, (\mathtt{D}, 1, 1)_{\mathtt{W}})$. Then $IMM(s)$={G,W} and, since $Select(\{\mathtt{G}, \mathtt{W}\}) = \mathtt{G}$, the output and internal state transition of G is selected. Thus $\delta_y(s) = (\mathtt{!g:0}, (\mathtt{R}, 1, 0)_{\mathtt{G}}, (\mathtt{DW}, 1, 1)_{\mathtt{W}})$ where $\delta_{y_{\mathtt{G}}}(\mathtt{GR}) = (\mathtt{!y:0}, \mathtt{R})$ and $C_{yy}(\mathtt{!y:0}_{\mathtt{G}}) = \mathtt{!g:0}$, $C_{yx}(\mathtt{!y:0}_{\mathtt{G}}) = \mathtt{!x:0}_{\mathtt{W}}$ and $\delta_{x_{\mathtt{W}}}(\mathtt{D}, \mathtt{!x:0}) = \mathtt{DW}$.

In the resultant A_N, we can define the total state transition from which we can then define the state trajectory, as we did previously for atomic SP-DEVS. The total event set is $Z = X \cup Y \cup \{\phi\}$. The legal state set is $Q_{\mathsf{p}} = \{(s, t_s, t_e) | s \in S, t_s = \tau(s), t_e \in tr(t_s)\}$ where S and $\tau(s)$ are the state set and the time advance function that are defined in Equations (4) and (5), respectively. The illegal state set is $Q_{\mathsf{imp}} = \{(s, \infty, t_e) | s = (\ldots, \mathsf{imp}_i, \ldots) \forall i \in D, t_e \in tr(\infty)\}$ such that $Q_{\mathsf{p}} \cap Q_{\mathsf{imp}} = \varnothing$. The total state set $Q = Q_{\mathsf{p}} \cup Q_{\mathsf{imp}}$.

Let us now return to N and suppose that $q = ((\ldots, q_i, \ldots), t_s, t_e) \in Q$ and $z \in Z$. Then the total state transition of N is defined by the function $\delta : Q \times Z \to Q$ such that

$\delta(q, z) =$

$$\begin{cases} (s', t_s, t_e) & \text{if } (\forall i, q_i \in Q_{\mathsf{p}_i}), z \in X, \delta_x((\ldots, q_i, \ldots), z) = s' \\ (s', \tau(s'), 0) & \text{if } (\forall i, q_i \in Q_{\mathsf{p}_i}), z \in Y^{\phi}, t_e = t_s, \delta_y((\ldots, q_i, \ldots)) = (z, s') \\ (\mathsf{imp}, \infty, t_e) & \text{otherwise} \end{cases} \tag{8}$$

where δ_x and δ_y are the functions defined in Equations (6) and (7), respectively. Equation (8) says that if there is no sub-component i reach to a illegal state, δ_x and δ_y can be applied. Otherwise, A_N stays in the illegal state. The state trajectory function $\Delta : Q \times \Omega_{[t_l, t_u]} \to Q$ is defined such that for $q = ((\ldots, (s_i, t_{si}, t_{ei}), \ldots), t_s, t_e) \in Q$ at time t_l, $\omega, \omega' \in \Omega_{[t_l, t_u]}$ and $z \in Z$,

$$\Delta(q,\omega) = \begin{cases} q + t_u - t_l & \text{for } \omega = \epsilon_{[t_l,t_u]} \\ \delta(\Delta(q,\omega'),z) & \text{for } \omega = \omega'(z,t_u) \end{cases} \quad (9)$$

where $q + t_u - t_l = ((\ldots,(s_i,t_{si},t_{ei}+t_u-t_l),\ldots),t_s,t_e+t_u-t_l)$ for all $i \in D$. Equation (9) says that during time interval $[t_l,t_u]$ of ϵ, the only things that can change are the elapsed times of all sub-components. In the meanwhile, the total state function δ is used for processing an event $z \in Z$. This procedure can be applied recursively.

Finally, the behavior of N, or the language of N, denoted by $L(N)$, is

$$L(N) = L(A_N) = \{\omega \in \Omega_{[0,\infty)} | \Delta(q_0,\omega) \in Q_{\mathtt{p}}\} \quad (10)$$

where $q_0 = (s_0, \tau(s_0), 0)$.

3. Time-Line Abstraction of SP-DEVS

In spite of the fact that the set of events and the set of states of the atomic SP-DEVS class each have a finite number of members and the fact that set of sub-components of the coupled SP-DEVS class is also finite, the number of states that we need to describe the behavior of a SP-DEVS model is uncountablely infinite because the continuous variable of elapsed time t_e can take on any non-negative real value as we saw in Equations (1) and (8). Fortunately, we can still construct the finite-state reachability graph of a given SP-DEVS model. To do this, we use a technique, called *time line abstraction*.

Even though this paper will focus on how to generate the finite-state reachability graph of the coupled SP-DEVS class later on, this technique can be easily applied to the atomic SP-DEVS class as follows. An atomic SP-DEVS model $A =< X_A, Y_A, S, s_0, \tau, \delta_x, \delta_y >$ can be seen as a coupled SP-DEVS model with a single sub-component which is the original atomic SP-DEVS model A such that $N =< X_N, Y_N, D, C_{xx}, C_{yx}, C_{yy}, Select >$ such that $X_N = X_A$; $Y_N = Y_A$; $D = \{A\}$; $C_{xx}(x) = x_A$ for all $x \in X_N$ and $x_A \in X_A$ s.t. $x = x_A$; C_{yx} is undefined for $y \in Y_A$; $C_{yx}(y) = y_N$ for all $y \in Y_A$ and $y_N \in Y_N$ s.t. $y = y_N$; and $Select(\{A\}) = A$.

3.1. Time-Line Equivalence and Time-Line Abstraction

Suppose that N is a coupled SP-DEVS such that $N =< X, Y, D, C_{xx}, C_{yx}, C_{yy}, Select >$, and its resultant is $A_N =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$.

DEFINITION 3.1. *Let* $s = (\ldots,(s_i,t_{si},t_{ei}),\ldots)$ *and* $s' = (\ldots,(s'_i,t'_{si},t'_{ei}),\ldots) \in S$. *Then* s *is a time-line equivalent (TLE) of* s', *denoted by*

$s \cong^t s'$, if (1)$\forall i \in D$, $(s_i = s'_i) \wedge (t_{si} = t'_{si})$, and (2)$\exists t \in \mathbb{T}$ s.t. $t'_{ei} = t_{ei} + t$ for each $i \in D$ s.t. $t_{si} < \infty$.

Let's try to understand the TLE concept using CC in Example 2. Suppose that we consider the following two states of CC $s = ((\mathtt{G}, 30, 28)_{\mathtt{G}}, (\mathtt{D}, 1, 0)_{\mathtt{W}})$ and $s' = ((\mathtt{G}, 30, 28.5)_{\mathtt{G}}, (\mathtt{D}, 1, 0.5)_{\mathtt{W}})$. Since for each $i \in D$, the state and the lifespan are identical in s and s' [i.e. (G, 30) for G, and (D, 1) for W], and since there exists $t = 0.5 \in tr(\tau(s)) = [0, 1]$ such that 28.5=28+t for G and 0.5=0+t for W, we can say the two state are TLE: $s \cong^t s'$.

Observe that the second condition in Definition 3.1 excludes the component $i \in D$ if $t_{si} = \infty$ because there is no possibility that component i will cause an output and internal state transition when $t_{si} = \infty$. In addition, the external state transition of a SP-DEVS model component i is independent of t_{ei}; we don't have to consider t_{ei} if $t_{si} = \infty$.

To avoid t_{ei} increasing infinitely without any significance in case of $t_{si} = \infty$, we define a *reset operation*, *Reset* over $(\ldots, (s_i, t_{si}, t_{ei}), \ldots) \in S$ such that

$$Reset((\ldots, (s_i, t_{si}, t_{ei}), \ldots)) = (\ldots, (s_i, t_{si}, t'_{ei}), \ldots) \tag{11}$$

where

$$t'_{ei} = \begin{cases} 0 & \text{if } t_{si} = \infty \\ t_{ei} & \text{otherwise.} \end{cases}$$

It is true that $\tau(Reset(s)) = \tau(s)$ and $s \cong^t s' \Leftrightarrow Reset(s) \cong^t Reset(s')$.

DEFINITION 3.2. *Given a coupled SP-DEVS $N =< X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select >$, then the time-line abstraction (TLA) of N is denoted by $TLA(N)$, and is defined by $TLA(N) =< X, Y, S^m, s_0, \tau^m, \delta_x^m, \delta_y^m >$ where, if $A_N =< X, Y, S, s_0, \tau, \delta_x, \delta_y >$ is a resultant of N, then*
$S^m = \{Reset(s) | \forall s, s' \in S \text{ s.t. } s \cong^t s', \tau(s) \geq \tau(s')\}$, $\tau^m := \tau|_{S^m \to \mathbb{T}^\infty}$, $\delta_x^m = \delta_x|_{S^m \times X \to S^m}$, *and* $\delta_y^m = \delta_y|_{S^m \to Y^\phi \times S^m}$.

By Definition 3.2, given a system, N, all states in N which are TLA to a particular state can be represented by a single representative state. For example, in our system CC, an infinite set of states $P = \{((\mathtt{G}, 30, 28+t_e)_{\mathtt{G}}, (\mathtt{D}, 1, t_e)_{\mathtt{W}}) | t_e \in [0, 1]\}$ can be represented by $s = ((\mathtt{G}, 30, 28)_{\mathtt{G}}, (\mathtt{D}, 1, 0)_{\mathtt{W}})$ in TLA(CC) because $\tau(s) = 1 > \tau(s')$ where $s' \in P$ and $s' \neq s$. Figure 5 illustrates the concept of time line abstraction.

LEMMA 1. *Given a coupled SP-DEVS N, if $G = TLA(N)$, then $L(N) = L(G)$.*

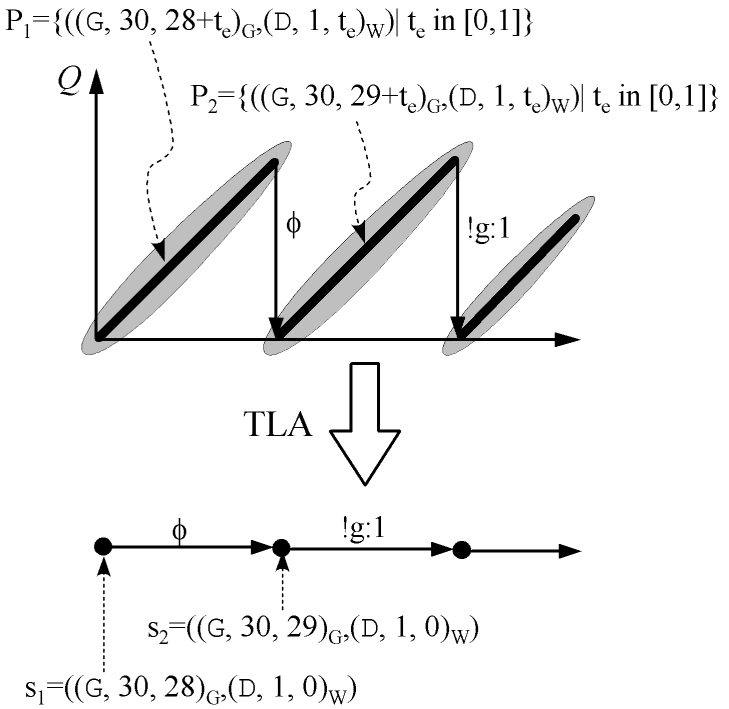

Figure 5. Time-Line Abstraction

Proof of Lemma 1. Since Definition (10) defines $L(N) = L(A_N)$, in order to show $L(N) = L(G)$, we need to show $L(A_N) = L(G)$.

Suppose that $s, s' \in S$ and $t \in tr(\tau(s))$ s.t. $s \cong^t s'$ and $\tau(s) \geq \tau(s')$. Since the event sequence leading s to s' is always the null-event sequence $\epsilon_{[0,t]}$ where , we can reconstruct the continuous state change from s to s' for any time, t. Therefore, there is no information loss in TLA(N), and so there is no behavioral difference between A_N and G. □

3.2. Generating TLA Graph from SP-DEVS Network

In the previous section, we saw that the uncountably-many TLE states can be combined into one state through the TLA operation while preserving the behavior of the system. Thus, there is a possibility that the TLA operation produces an equivalent and finite-state reachability graph.

This section introduces an algorithm for generating a TLA graph from a coupled SP-DEVS model, such that the generated TLA graph whill have a finite number of vertices and edges. The algorithm we introduce here consists of two procedures: `TLA` and `UpdatingS` as shown

in Algorithm 1, below `TLA` is the main procedure in which a newly generated state is registered by calling the procedure `UpdatingS` whose function is explained in lines 20-26 of the algorithm.

In this algorithm, G is the TLA(N) that we want to obtain from N, S^m is the set of states of G, and V_T is a set of states to be tested from which we will generate the next states. `TLA` starts by adding the initial state $s_0 = (\ldots, (s_i, t_{si}, t_{ei}), \ldots)$ (see line 5 in Algorithm 1) to S^m.

`TLA` next picks a state s from the non-empty, but still finite, V_T (line 7), and it resets s by the reset operation *Reset* defined by above Equation (11)(line 8 of the algorithm). Then for each input event $x \in X$, the external state transition function δ_x defined by Equation (6) is used to generate the next state s' from s and x (lines 9-11).

If the lifespan of s is finite, the algorithm lets the elapsed time following each input, t_{ei} exceed the lifespan $\tau(s)$. Then it applies the output and internal state transition function, $\delta_y(s)$ defined by Equation (7) to generate the next state s' as well as the associated output event (lines 12-15).

This procedure continues as long as V_T is not empty.

Algorithm 1 Generating TLA Graph

```
 1 G TLA(↓ N)
 2   //N =< X, Y, C, C_xx, C_yx, C_yy, Select >;
 3   //G =< X, Y, S^m, s_0, τ^m, δ^m_x, δ^m_y >;
 4   V_T: Set<s>; //a set of states to be tested.
 5   UpdatingS(G, V, s_0);
 6   do
 7      s := V_T.pop_front();
 8      s := Reset(s);
 9      for_all x in X
10        δ^m_x(s, x) := s';
11        UpdatingS(G, V_T, s');
12      if τ(s) < ∞ then
13        ∀i, t_ei := t_ei + τ(s);
14        δ^m_y(s) := (y, s');
15        UpdatingS(G, V_T, s');
16   while(V ≠ ∅);
17   return G;
18
19 UpdatingS(↕ G, ↕ V_T, ↓ s = (..., (s_i, t_si, t_ei), ...))
20   if s ∉ S^m then
21     add s to S^m;
22     τ^m(s) := min{t_ri = t_si − t_ei | i ∈ D};
23     V_T.insert(s);
```

3.2.1. *Completeness of Algorithm 1*
In the TLA operation defined by Definition 3.2, the passage of t_e is abstracted so that only *discrete* state transitions such as those generated by the external state transition function and the output and internal state transition function are considered to produce states in the state space.

Algorithm 1 considers every possible event $x \in X$ for each state s when calling $\delta_x(s, x)$ (lines 9-11). In addition, it generates the next state by using the output and internal state transition function if $\tau(s) < \infty$, Algorithm 1 does this by first making t_{ei} jump to $t_{ei} = t_{ei} + \tau(s)$ for each $i \in D$ (line 13). It then, applies $\delta_y(s)$ to identify the next state of the system. Notice that it is impossible to execute $\delta_y(s)$ when $\tau(s) = \infty$.

Thus Algorithm 1 traces completely all possible states in S^m if the number of states, $|S^m|$, is finite. The bound of $|S^m|$ will be examined in the next section.

3.2.2. *Complexity of Algorithm 1*
Since Algorithm 1 continues as long as V_T is not empty, and when a state s can be added to V_T when $s \notin S^m$ (see line 20), the number of states tested in lines 6 through 16 is $|S^m|$. The computing time and storage required for performing Algorithm 1 is proportional to $|S^m|$. Thus, we would use the bound of $|S^m|$ as the complexity index of Algorithm 1 here.

Let $N =< X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}, Select >$ be a coupled SP-DEVS such that M_i is a component of N and is an SP-DEVS for all $i \in D$. Assume also that a state of G is $s = (\ldots, (s_i, t_{si}, t_{ei}), \ldots) \in S^m$ where for all $i \in D$, $(s_i, t_{si}, t_{ei}) \in Q_i$. The possible number of s_i is bounded $|S_i|$ which is a finite number for atomic SP-DEVS. Then the number of possible values of the lifespan, t_{si} can be also bounded by $|S_i|$ because these values can be scheduled by $\tau_i(s_i)$ for all $s_i \in S_i$ (see Equation (1)).

Even though t_{ei} is a continuous variable, the TLA operation of Definition 3.2 captures the value of t_{ei} when $t_e = 0$, which occurs either when the system is initialized or right after the output and internal state transition function, δ_y, is triggered (see Equation (8)).

Let $g \in \mathbb{Q}_{[0,\infty]}$ be the *greatest common divisor* of all $\tau_i(s_i)$ for all $s_i \in \bigcup_{i \in D} S_i$. Given $s = (\ldots, (s_i, t_{si}, t_{ei}), \ldots)$, let u_i be the number of different values of t_{ei} in (s_i, t_{si}, t_{ei}) that is

$$u_i = \begin{cases} 1 & t_{si} = \infty \text{ (by the reset operation)} \\ 0 & t_{si} = 0 \text{ (by } t_{ei} \in tr(0)) \\ t_{si}/g + 1 & \text{otherwise} \end{cases}$$

Let's define the set of times (of the various sub-components) which are less than infinity as follows: $t_{s,<\infty} = \{\tau_i(s_i) | \tau_i(s_i) < \infty, s_i \in S_i, i \in D\}$. If $t_{s,<\infty} = \varnothing$, for all $i \in D$, $s_i \in S_i$, $\tau_i(s_i) = \infty$. Thus $t_{ei} = 0$ in a state of $TLA(N)$ by the reset operation. If $\max(t_{s,<\infty}) = 0$, for all $i \in D$, $s_i \in S_i$, $\tau_i(s_i) = 0$. Thus $t_{ei} = 0$ in a state of $TLA(N)$ because $t_{ei} \in tr(0)$. Let us define the upper bound number of different values of t_{ei} in a state (s_i, t_{si}, t_{ei}) as

$$u_{max} = \begin{cases} 1 & t_{s,<\infty} = \infty \text{ (by the reset operation)} \\ 0 & \max(t_{s,<\infty}) = 0 \text{ (by } t_{ei} \in tr(0)) \\ \max(t_{s,<\infty})/g + 1 & \text{otherwise} \end{cases}$$

Since $u_i \leq u_{max}$, $|S^m| \leq \prod_{i \in D} (|S_i| \cdot |S_i| \cdot u_{max})$.

In addition, by the assumption of finite depth of hierarchy, even if M_i is a coupled SP-DEVS model, $|S^m|$ is bounded as finite.

THEOREM 1. *The behavior of a SP-DEVS network, N, can be described by the behavior of an atomic SP-DEVS model.*

Proof of Theorem 1. We can make $G = TLA(N)$ by Algorithm 1. Notice that the structure of G is the form of atomic SP-DEVS. In addition, $L(N) = L(G)$ by Lemma 1. □

Theorem 1 provides the foundation that allows us to use the TLA graph G that can be seen as a finite-state reachability graph, instead of a SP-DEVS M model that can be either atomic or coupled model, for analysis of M.

Sometimes, a graphical representation is more efficient than a function description for analysing the behavior of a system. For this reason, we will denote a TLA(M) structure as G, where G was previously defined by: $G =< X, Y, S^m, s_0, \tau^m, \delta_x^m, \delta_y^m >$, but can also be described as a labelled graph

$$G =< X, Y, V, v_0^*, E >$$

where

- $V = \{(s^m, \tau^m(s^m)) | s^m \in S^m\}$ is a finite set of vertices.
- $v_0^* = s_0$ is the initial vertex.
- $E = \{e | e = v \xrightarrow{z} v'\}$ is a finite set of active edges such that

$$v \xrightarrow{z} v' \in E \Leftrightarrow (\delta_x^m(v, z) = v' \wedge v \neq v') \vee \delta_y^m(v) = (z, v').$$

In addition, without loss of generality, we will restrict our attention to active event sequences in which every event causes a state change. Given a SP-DEVS model M, $\omega \in L(M)$ is *active* if $\omega = \omega'(z,t)\omega''$ where $\omega' \in \Omega_{[0,t]}, \omega'' \in \Omega_{[t,\infty)}$, $z \in Z$ and $t \in \mathbb{T}$ implies that $\Delta(q_0, \omega') \neq \delta(\Delta(q_0, \omega'), z)$. The language of M can be also restricted so that the *active behavior* or *active language* of M is defined by

$$L_a(M) = \{\omega | \omega \in L(M), \omega \text{ is active.}\}. \tag{12}$$

4. Processing Time Bounds of SP-DEVS

This section introduces definitions of processing time and processing time bounds. Finally, two algorithms computing the processing time bounds of all event sequences which start and end with two specific events of SP-DEVS will be introduced.

4.1. General Framework of Processing Time Bounds

This section introduces a general framework for computing the time bound of a given event sequence. Let Z be an event set and $L \subseteq \Omega$ be an timed language over Z. Given a timed event string $\omega = (z_0, t_0) \cdots (z_n, t_n) \in L$, *the processing time* of ω is denoted as $PT(\omega)$ and is defined by

$$PT(\omega) = t_n - t_0. \tag{13}$$

The *minimum processing time of* L is denoted by $MinPT(L)$ and is defined by

$$MinPT(L) = \begin{cases} \text{undefined} & \text{if } L = \varnothing \\ \min\limits_{\omega \in L} PT(\omega) & \text{otherwise.} \end{cases} \tag{14}$$

Similarly, the *maximum processing time of* L is denoted by $MaxPT(L)$ and is defined by

$$MaxPT(L) = \begin{cases} \text{undefined} & \text{if } L = \varnothing \\ \max\limits_{\omega \in L} PT(\omega) & \text{otherwise.} \end{cases} \tag{15}$$

The processing time bounds in Equations (14) and (15) are defined in the general context of any languages. From now on, we will refine these definitions in the context of a language associated with SP-DEVS.

4.2. Cost Evaluation of a Path in TLA Graph

As mentioned in the previous section, to analyze the language of a SP-DEVS M, we would prefer to look at the time line abstraction of M, $G = TLA(M)$ rather than at M directly. When determining the processing time bound of M, we will find it easier to evaluate the time cost of traveling through the graph structure of $G = TLA(M)$.

DEFINITION 4.1. *Given a SP-DEVS M, let G be the TLA graph of M s.t. $TLA(M) = G =< X, Y, V, v_0^*, E >$ as defined Theorem 1 in Section 3.2.2, above. An untimed path of G is any vertex and edge sequence (e.g., $v_0 \xrightarrow{z_0} \cdots v_i \xrightarrow{z_i} v_{i+1} \cdots \xrightarrow{z_n} v_{n+1}$) in G.*

Note that v_0 doesn't have to be v_0^*.

DEFINITION 4.2. *The set of timed path of $\pi = v_0 \xrightarrow{z_0} \cdots v_i \xrightarrow{z_i} v_{i+1} \cdots \xrightarrow{z_n} v_{n+1}$ in G is denoted as $\Pi^t(\pi, G)$ and is defined by*

$$\Pi^t(\pi, G) = \{\pi^t | \pi^t = q_0 \xrightarrow[t_0]{z_0} \cdots \xrightarrow[t_n]{z_n} q_{n+1}\}$$

where $\pi^t = q_0 \xrightarrow[t_0]{z_0} \cdots \xrightarrow[t_n]{z_n} q_{n+1}$ is a timed path corresponding to π such that for each $(s_i, t_{s,i}) \xrightarrow{z_i} (s_{i+1}, t_{s,i+1}) \in E$ of G, 1) $q_i = (s_i, t_{s,i}, t_{e,i}) \in Q_{\mathtt{p}}$ where $t_{ei} \in tr(t_{s,i})$, and 2) $q_{i+1} = \delta(q_i, z_i)$.

Let's consider the system CC previously discussed in Example 1 and let π be an untimed path given by π = (G,30) $\xrightarrow{\text{?p}}$(GR,30) $\xrightarrow{\text{!g:0}}$ (R,2) $\xrightarrow{\text{!w:1}}$ (W,26) $\xrightarrow{\text{!w:0}}$ (D,2) $\xrightarrow{\text{!g:1}}$ (G,30). We can then envision two timed paths as follows: 1) π_1^t= (G,30,9) $\xrightarrow[10]{\text{?p}}$(GR,30,9) $\xrightarrow[31]{\text{!g:0}}$(R,2,0) $\xrightarrow[33]{\text{!w:1}}$(W,26,0) $\xrightarrow[59]{\text{!w:0}}$(D,2,0) $\xrightarrow[61]{\text{!g:1}}$(G,30,0), and 2) π_2^t=(G,30,30) $\xrightarrow[31]{\text{?p}}$(GR,30,9) $\xrightarrow[31]{\text{!g:0}}$(R,2,0) $\xrightarrow[33]{\text{!w:1}}$(W,26,0) $\xrightarrow[59]{\text{!w:0}}$(D,2,0) $\xrightarrow[61]{\text{!g:1}}$(G,30,0). Then the processing times of the two timed event strings ω_1 =(?p,10)(!g:0,31)(!w:1,33)(!w:0,59)(!g:1,61) and ω_2 =(?p,31)(!g:0,31)(!w:1,33)(!w:0,59)(!g:1,61) are $PT(\omega_1) = 61 - 10 = 51$ and $PT(\omega_2) = 61 - 31 = 30$, respectively.

DEFINITION 4.3. *The set of timed language of $\pi = v_0 \xrightarrow{z_0} \cdots v_i \xrightarrow{z_i} v_{i+1} \cdots \xrightarrow{z_n} v_{n+1}$ in G is denoted as $L(\pi, G)$ and is defined by*

$$L(\pi, G) = \{(z_0, t_0) \ldots (z_n, t_n) | \pi^t = q_0 \xrightarrow[t_0]{z_0} \cdots \xrightarrow[t_n]{z_n} q_{n+1} \in \Pi^t(\pi, G)\}.$$

4.2.1. *Minimum Processing Time for an Untimed Path*

DEFINITION 4.4. *Given a SP-DEVS M and an untimed path $\pi \in G = TLA(M)$, the minimum processing time of π in G is $MinPT(L(\pi, G))$.*

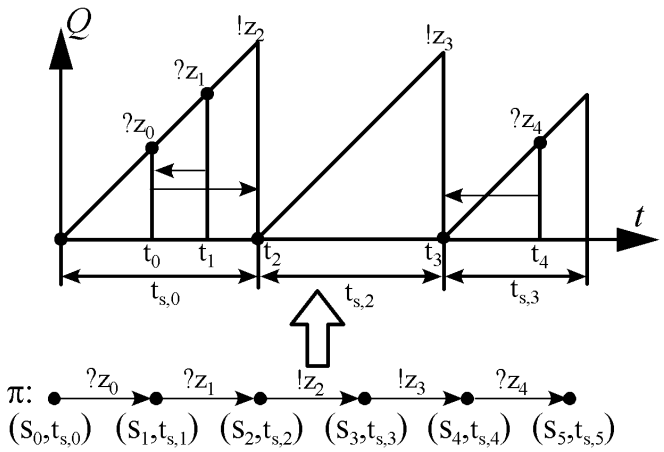

Figure 6. Determining the shortest timed path. An input event $z_i \in X$ should occur as soon as possible except $z_0 \in X$ that should occur as late as possible.

Figure 6 illustrates the condition necessary for creating the shortest timed path, denoted by π^{t*}, given an untimed path $\pi = (s_0, t_{s,0}) \xrightarrow{?z_0} (s_1, t_{s,1}) \xrightarrow{?z_1} (s_2, t_{s,2}) \xrightarrow{!z_2} (s_3, t_{s,3}) \xrightarrow{!z_3} (s_4, t_{s,4}) \xrightarrow{?z_4} (s_5, t_{s,5})$. π^{t*} occurs when $t_0 = t_1 = t_2$, $t_3 = t_2 + t_{s,2}$, and $t_4 = t_3$; and $MinPT(L(\pi, G)) = t_4 - t_0 = t_{s,2}$.

The non-determinism of a processing time of a path π is caused by the various external state transitions in π because an external state transition $v_i \xrightarrow{x} v_{i+1}$ can occur with the many differences in the elapsed time in the time path π^t such that $q_i \xrightarrow[t_i]{x} q_{i+1}$ where $q_i = (s_i, t_{s,i}, t_{e,i})$ can vary over all $t_{e,i} \in tr(t_{s,i})$.

It turns out that there is a simple rule for the shortest timed path π^{t*}: each external state transition for $z_i \in X$ in π^{t*} should occur "as soon as possible" except for the initial input, z_0 which should happen "as late as possible". These conditions are incorporated into the following theorem.

THEOREM 2. *Given a SP-DEVS M, suppose that there is an untimed path $\pi = v_0 \xrightarrow{z_0} \cdots v_i \xrightarrow{z_i} v_{i+1} \cdots \xrightarrow{z_n} v_{n+1}$ in $G = TLA(M)$ where $n > 0$ and $v_i = (s_i, t_{s,i})$. Let $\omega = (z_0, t_0) \cdots (z_n, t_n) \in L(\pi, G)$. Then $MinPT(L(\pi, G)) = t_n^*$ is given by the following procedure: for $i = 0$ to n,*

1. $t_0^* = 0$
2. *if $i > 0$ and $z_i \in X$, then $t_i^* = t_{i-1}^*$*
3. *otherwise*

a) if $z_0 \in X$ and z_i is the first output event since z_0, then $t_i^* = t_{i-1}^*$

b) otherwise, $t_i^* = t_{i-1}^* + t_{s,i}$,

Proof of Theorem 2. Let us use $\pi(j)$ as the prefix-path up to j-th event z_j of π such that $\pi(j) = v_0 \xrightarrow{z_0} \cdots v_j \xrightarrow{z_j} v_{j+1}$ where $0 \leq j \leq n$. Let $MinPT(L(\pi(i-1), G)) = t_{i-1}^*$ be the minimum processing time from z_0 to z_{i-1} where $0 < i \leq n$. Let's now evaluate $MinPT(L(\pi(i), G))$ which is the minimum processing time from z_0 to z_i.

Observe that, in order to make $\pi^t(i)$ shortest, $z_i \in X$ should occur at $t_{e,i} = 0$ for no-delay (as soon as possible if $i > 0$). That means, $t_i^* = t_{i-1}^*$ and it is Condition 2 of Theorem 2.

Let's consider the case of $z_i \in Y^\phi$. First, suppose that $z_0 \in X$ and z_i is the first output event in Y^ϕ since z_0 (Condition 3*a*). To have the shortest length $\pi^t(i)$, $v_0 \xrightarrow{z_0} v_1$ should happens when $t_{e,0} = t_{s,0}$, it is the condition "as late as possible" for z_0. Since z_i is the first output event since z_0, the condition $t_{e,j} = t_{s,0}$ holds for all j=0 to i without resetting $t_{e,j}$. Thus, $t_i^* = t_{i-1}^*$.

Let's consider the other case of $z_i \in Y^\phi$ but z_i is *not* the first output event in Y^ϕ since $z_0 \in X$ (Condition (3*b*))". In both cases of $z_{i-1} \in Y^\phi$ (by Equation (1)) and $z_{i-1} \in X$ (by Condition 2 of this Theorem)), and z_i can occur only when $t_{e,i} = t_{s,i}$ (by Equation (1)), so the difference between t_{i-1}^* and t_i^* is $t_{s,i}$. Thus, we can say $t_i^* = t_{i-1}^* + t_{s,i}$.

Since Conditions 1, 2, 3*a* and 3*b* hold $MinPT(L(\pi(i), G)) = PT(\pi^t(i)) = t_i^*$, $MinPT(L(\pi(n), G)) = MinPT(L(\pi, G)) = t_n^* - t_0^* = t_n^*$ ($\because t_0^* = 0$) by induction. □

For instance, suppose we want to know the minimum processing time of π = (G,30) $\xrightarrow{?p}$(GR,30) $\xrightarrow{!g:0}$ (R,2) $\xrightarrow{!w:1}$ (W,26) $\xrightarrow{!w:0}$ (D,2,) $\xrightarrow{!g:1}$ (G,30) of CC in Example 1. Then we can calculate $MinPT(\pi, \texttt{CC}) = 30$ by observing π^{t*}=(G,30,30) $\xrightarrow[t_0]{?p}$(GR,30,30) $\xrightarrow[t_1]{!g:0}$(R,2,0) $\xrightarrow[t_2]{!w:1}$(W,26,0) $\xrightarrow[t_3]{!w:0}$(D,2,0) $\xrightarrow[t_4]{!g:1}$(G,30,0) and noting $t_1 = t_0 = 0, t_2 = t_1 + 2, t_3 = t_2 + 26, t_4 = t_3 + 2$ and $t_4 = 0 + 2 + 26 + 2 = 30$.

4.2.2. *Maximum Processing Time of an Untimed Path*

DEFINITION 4.5. *Given a SP-DEVS M and an untimed path $\pi \in G = TLA(M)$, its maximum processing time of π in G is $MaxPT(L(\pi, G))$.*

Figure 7 illustrates the methodology for determining the longest timed path, denoted by π^{t*}, from an untimed path $\pi = (s_0, t_{s,0}) \xrightarrow{?z_0} (s_1, t_{s,1}) \xrightarrow{?z_1} (s_2, t_{s,2}) \xrightarrow{!z_2} (s_3, t_{s,3}) \xrightarrow{!z_3} (s_4, t_{s,4}) \xrightarrow{?z_4} (s_5, t_{s,5})$. The

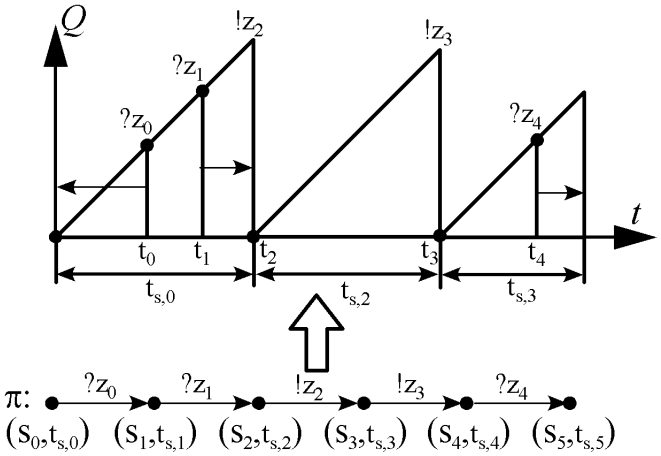

Figure 7. Determining the longest timed path. An input event $z_i \in X$ should occur as late as possible except $z_0 \in X$ that should occur as soon as possible.

maximum processing time $MaxPT(L(\pi, G)) = t_4 - t_0 = t_{s0} + t_{s,2} + t_{s,3}$ if $t_1 = t_0 + t_{s,0}, t_1 = t_2$, $t_3 = t_2 + t_{s,2}$, and $t_4 = t_3 + t_{s,3}$.

In contrast to the case of the shortest timed path, for the case of the longest timed path, we will require external state transitions to occurs "as late as possible" unless the external state transition is the first one in the timed path, in which case, it should happens "as soon as possible". These conditions are reflected in the following theorem.

THEOREM 3. *Given a SP-DEVS M, suppose that there is a loop-free untimed path* $\pi = v_0 \xrightarrow{z_0} \cdots v_i \xrightarrow{z_i} v_{i+1} \cdots \xrightarrow{z_n} v_{n+1}$ *in* $G = TLA(M)$ *where* $n > 0$ *and* $v_i = (s_i, t_{s,i})$. *Let* $\omega = (z_0, t_0) \cdots (z_n, t_n) \in L(\pi, G)$. *Then* $MaxPT(L(\pi, G)) = t_n^*$ *is given by the following procedure: for* $i = 0$ *to* n,

1. $t_i^* = 0$,
2. *if* $i = 1$ *then* $t_1^* = t_{s,1}$,
3. *else if* $z_{i-1} \in X$ *then* $t_i^* = t_{i-1}^*$
4. *otherwise* $t_i^* = t_{i-1}^* + t_{s,i}$.

Proof of Theorem 3. Let $MaxPT(L(\pi(i-1), G)) = t_{i-1}^*$ be the maximum processing time from z_0 to z_{i-1} where $0 < i \leq n$. Let's now evaluate $MaxPT(L(\pi(i), G))$ which is the maximum processing time from z_0 to z_i.

Let's assume that $i = 1$. This is the case of Condition 2, so the longest length of $\pi^t = q_0 \xrightarrow[t_0^*]{z_0} q_1 \xrightarrow[t_1^*]{z_1} q_2$ is $t_1^* = t_{s,1}$ regardless of z_0's

type because (1) $z_0 \in X$ occurs as soon as possible, that is, $t_{e,0} = 0$, and (2) $z_0 \in Y^{\phi}$ resets $t_{e,1}$ by 0, and z_1 occurs at $t_{e,1} = t_{s,1}$.

If $i > 1$, there are two possible cases: $z_{i-1} \in X$ and $z_{i-1} \in Y^{\phi}$. If $z_{i-1} \in X$, that is Condition 3, since z_{i-1} occurs as late as possible in the longest timed path when $t_{e,i-1} = t_{s,i-1}$ and the rest of $t_{e,i}$, there is no difference between t^*_{i-1} and t^*_i so $t^*_i = t^*_{i-1}$.

Otherwise,($z_{i-1} \in Y^{\phi}$, that is Condition 4), since when z_{i-1} occurs, $t_{e,i}$ is reset to 0, as well as, z_i happens at $t_{e,i} = t_{s,i}$, thus $t^*_i = t^*_{i-1} + t_{s,i}$.

Since above all conditions hold $MaxPT(L(\pi(i), G)) = PT(\pi^t(i)) = t^*_i$, $MaxPT(L(\pi(n), G)) = MaxPT(L(\pi, G)) = t^*_n - t^*_0 = t^*_n$ ($\because t^*_0 = 0$) of Theorem 3 is true by induction. □

For instance, suppose we want to know the maximum processing time of $\pi = (\mathtt{G},30) \xrightarrow{?\mathrm{p}} (\mathtt{GR},30) \xrightarrow{!\mathrm{g}:0} (\mathtt{R},2) \xrightarrow{!\mathrm{w}:1} (\mathtt{W},26) \xrightarrow{!\mathrm{w}:0} (\mathtt{D},2) \xrightarrow{!\mathrm{g}:1} (\mathtt{G},30)$ of CC in Example 1. Then we can calculate $Max(\pi, \mathtt{CC}) = 60$ where π^{t*}=(G,30,0) by observing that $\xrightarrow[t_0]{?\mathrm{p}}$(GR,30,30) $\xrightarrow[t_1]{!\mathrm{g}:0}$(R,2,0) $\xrightarrow[t_2]{!\mathrm{w}:1}$(W,26,0) $\xrightarrow[t_3]{!\mathrm{w}:0}$(D,2,0) $\xrightarrow[t_4]{!\mathrm{g}:1}$(G,30,0) and noting $t_1 = 30, t_2 = t_1 + 2, t_3 = t_2 + 26, t_4 = t_3 + 2$ and so $t_4 = 30 + 2 + 26 + 2 = 60$.

4.3. Processing Time Bounds between Two Events of SP-DEVS

Recall that both the minimum processing time $MinPT(L)$ defined by Equation (14) and the maximum processing time $MaxPT(L)$ defined by Equation (15) need the language L, to be specified.

Given a language $L \subseteq \Omega$, the *sub-language of* L is denoted by $sub(L)$ and is defined by

$$sub(L) := \{\omega' | \omega \in L, \omega = \omega_1 \omega' \omega_2\}.$$

Given a timed event sequence $\omega = (z_0, t_0) \dots (z_0, t_n)$ where $n > 0$, let $i(\omega)$ denote the initial event of ω s.t. $i(\omega) = z_0$, and let $f(\omega)$ denote the final event of ω s.t. $f(\omega) = z_n$.

Given a SP-DEVS model M, and two events z_s and z_e in Z of M, this section considers an active sub-languages of M, denoted by $L(M, z_s, z_e)$ in which every timed event sequence starts with z_s and ends with z_e. Formally, $L(M, z_s, z_e)$ is defined by

$$L(M, z_s, z_e) := \{\omega | \omega \in sub(L_a(M)), i(\omega) = z_s, f(\omega) = z_e\}.$$

Recall that $L_a(M)$ is an active language of M which was defined in Equation (12).

The following two subsections show how to compute $MinPT(L(M, z_s, z_e))$ and $MaxPT(L(M, z_s, z_e))$.

4.3.1. *Computing* $MinPT(L(M, z_s, z_e))$
Since the main procedure for computing $MinPT(L(M, z_s, z_e))$ uses "One-source Shortest Paths (OSP)" algorithm, we will examine the OSP algorithm first and then introduce the main procedure.

4.3.1.1. *One-source Shortest Paths* From the ending vertex v' of an initial transition edge $e_0 = v \xrightarrow{z} v'$, the OSP algorithm computes a shortest time vector `wt` for each vertex as well as an incoming edge vector `spt` for each vertex on each of the shortest paths in G. In order to determine the time costs of the external state transitions, we will use Theorem 2 that obtains for us the shortest event sequence to each vertex. Observe that the OSP algorithms 2 is identical to the Dijkstra algorithm for the one-source shortest paths (Sedgewick, 2002) except when evaluating the transition costs.

Let $|E|$ and $|V|$ be the number of edges and the number of vertices of G, respectively. Since the complexity of the Dijkstra's Algorithm is $O(|E| \log |V|)$ when we use the priority queue `Q` (Sedgewick, 2002), the complexity of Algorithm 2 is also $O(|E| \log |V|)$ because checking conditions in lines 14 to 16 need a constant time, .

Algorithm 2 OSP: One-source Shortest Path

```
 1 OSP(↓ G, ↓ e0, ↑ wt, ↑spt)
 2   //G =< X, Y, V, v0, E >;
 3   //e0 = (v -z-> v'); edge structure
 4   bool Con3a := (e0.z ∈ X)? true: false;
 5   for each v ∈ G.V // Initializations
 6     wt[v] := ∞;
 7     spt[v] := undefined;
 8   wt[e0.v'] := 0; // Condition 1.
 9   Q.push(e0.v');
10   while (Q ≠ ∅)
11     v:=Extract_Min(Q) // Remove the best vertex from queue
12     for each edge e = (v -z-> v') outgoing from v = (s, ts, te)
13       double p;
14       if (e.z ∈ X) p := 0; // Condition 2.
15       else if (Con3a = true) { p := 0; Con3a := false; }
16       else p = v.ts; // Condition 3.
17       if (wt[e.v] + p < wt[e.v']) // Update wt[e.v']
18         wt[e.v'] := wt[e.v] + p;
19         spt[e.v'] := e
20         Q.push(e.v');
```

4.3.1.2. *Main Procedure for Computing $MinPT(L(M, z_s, z_e))$* Recall that the `OSP` algorithm computes the one-source shortest time vector `wt` for each vertex, and an incoming edge vector `spt` for each vertex on each of shortest paths. Based on `OSP`, the minimum processing time defined in Equation (19) can be calculated using Algorithm 3 (shown below) in which a set of edges $\{e|e = (v \xrightarrow{z} v')\}$ is computed by `Get_Edges`(G, z) whose computational complexity is bounded by $O(|E|)$. Since there are two consecutive calls of `Get_Edges`(G, z), and `OSP` is called for each $e \in E_s$, the computational complexity is bounded by $O(|E| + |E| + |E| \cdot |E| \log |V|)) = O(|E|^2 \log |V|)$.

Algorithm 3 Minimum Processing Time

```
1 MinPT(↓ G, ↓ z_s, ↓ z_e, ↑min, ↑shortest_path)
2   //G =< X, Y, V, v_0, E >;
3   //z_0, z_e ∈ X ∪ Y^φ starting and ending events;
4   Es = Get_Edges(G, z_s);
5   Ee = Get_Edges(G, z_e);
5   min := ∞;
6   for each e_s ∈ Es
7     OSP(G, e_s, wt, spt);
8     for each e_e = (v --z--> v') ∈ Ee
9       if(wt[e_e.v'] < min)
10        min := wt[e_e.v'];
11        shortest_path := spt;
```

4.3.2. *Computing $MaxPT(L(M, z_s, z_e))$*
Unlike the shortest path algorithm, the search for the longest path cannot terminate when there is a loop in the path (Sedgewick, 2002). We need to preprocess a path $\pi \in G$ from z_s to z_e to see if it has a loop. If a path π has a loop, the maximum processing time of π is infinite because of the possibility of repeating in the loop an infinite number of times.

4.3.2.1. *One-source Longest Paths* The basic assumption we will make in seeking the one-source longest path of a graph is that the graph is a directed acyclic graph (DAG)(Sedgewick, 2002). If we have a DAG, we can apply the topological sorting to it and then compute the longest path from the sorted DAG (Sedgewick, 2002). While searching for the longest path using the topologically sorted DAG, we can evaluate the cost of each transition by Theorem 3.

The "One-source Longest Path (OLP)" algorithm (Algorithm 4) is based on the above ideas, so it needs a DAG, G_{DAG}, and a starting transition e_s as its initial input information. The OLP algorithm

then calculates two output results: a longest time vector `wt`, and the incoming path vector `lpt` for each state on the longest path.

To obtain the topologically sorted graph from G_{DAG}, the OLP algorithm calls `TopologicalSorting`(G_{DAG}, e_s)and gets G_{TS} which sorts G_{DAG} from the starting vertex of e_s in the reverse order (Sedgewick, 2002). As a result, the order of the starting vertex of e_s is the biggest number in G_{TS}, while the topologically farthest vertex from e_s has order 0. Thus, from the index $G_{TS}.|V|$ to 0 where $G_{TS}.|V|$ is the number of vertices in G_{TS}, the algorithm tries to update `wt` and `lpt` by evaluating the time-delay cost for each transition in the case of the longest path (which was proven to exists in Theorem 3).

Algorithm 4 OLP: One-source Longest Path

```
1 OLP(↓ G_DAG, ↓ e_s, ↑ wt, ↑ lpt)
2   //G_DAG =< X,Y,V,v_0,E >: DAG;
3   //e_s = (v --z_s--> v'): starting edge;
4   A_TS:=TopologicalSorting(G_DAG, e_s);
5   for (int i:=G_TS.|V|-1; i >= 0; i--)
6     if(i=G_TS.|V|-1) // Condition 1 of Theorem 2
7       wt[e_s.v'] := 0;
8       lpt[e_s.v'] := e_s;
9     else
10      v := G_TS.V(i);
11      for each edge e = (v --z--> v') outgoing from v = (s,t_s,t_e)
12        double p;
13        if (e_s = lpt[e.v]) p := e.v.t_s; // Condition 2.
14        else if (lpt[e.v].z ∈ X ) p := 0; // Condition 3.
15        else p = e.v.t_s; // Condition 4.
16        if (wt[e.v'] < wt[e.v] + p)
17          wt[e.v'] := wt[e.v] + p;
18          lpt[e.v'] := e;
```

The complexity of Algorithm 4 is $O(|V||E|)$ where V and E are the vertex set and the edge set of G_{DAG}, because the topological sorting is bounded by $O(|E|)$ (Sedgewick, 2002), and the update of `wt` and `lpt` in lines 5 to 18 is bounded by $O(|V||E|)$.

4.3.2.2. *Main Procedure for Computing* $MaxPT(L(M, z_s, z_e))$ Algorithm 5 is the main procedure used to calculate the maximum processing time, `max`, and the longest path, `longest_path`, if they exist. The procedure, `Get_DAG` called in line 7 (see Appendix), obtains an acyclic subgraph G_{DAG} from G, if the return value of `Get_DAG` is `true`.

If G_{DAG} has a path that terminates with an edge whose triggering event is not z_s, or G_{DAG} is not a directed acyclic graph (which means

G_{DAG} has a loop), the maximum processing time `max` is set to ∞ (see lines 8 to 11). Otherwise, we compute the longest time vector `wt` and the incoming edge vector `lpt` for each vertex by calling `OLP` (Algorithm 4).

After getting `wt` and `lpt` from `OLP`, then for each transition edge whose associated event is z_e, where $e = (v \xrightarrow{z_e} v')$ of G_{DAG}, and `wt`$[e.v]$ is greater than `max`, update `max`:= `wt`$[e.v]$ and `longest_path`:=`lpt`, as shown in lines 15 to 17.

Algorithm 5 Maximum Processing Time

```
 1 MaxPT(↓ G, ↓ z_s, ↓ z_e, ↑max, ↑longest_path)
 2   //G =< X, Y, V, v_0, E >;
 3   //z_s, z_e ∈ Z starting and ending events;
 4   Es := Get_Edges(G, z_s);
 5   max := 0; longest_path := ∅;
 6   for each e_s in Es
 7     all_end_with_z_e = Get_DAG(G, e_s, z_e, G_DAG);
 8     if((all_end_with_z_e = false) or (G_DAG has a loop))
 9       max := ∞;
10       longest_path := ∅;
11       return;
12     else
13       OLP(G_DAG, e_s, wt, lpt);
14       for each e = (v --z_e--> v') in G_DAG.E
15         if (max < wt[e.v'])
16            max := wt[e.v'];
17            longest_path := lpt;
```

Observe that the main loop of `MaxPT` is bounded by $O(|E|)$, and `Get_DAG` is bounded by $O(|V| + |E|)$ (see Appendix). THe check to determine if G_{DAG} contains a loop is performed by $O(|E|_{G_{DAG}}) < O(|E|)$ (Sedgewick, 2002). `OLP` is bounded by $O(|E|_{G_{DAG}}|V|_{G_{DAG}}) < O(|E||V|)$ (see Algorithm 4). Thus $O(|E| \cdot (((|V|+|E|)+|E|+|E||V|)) = O(|E|^2|V|)$.

5. Experimental Results

As illustrated in Figure 1, the procedure for computing the processing time bounds consists of three steps (1) creating time abstraction by using the TLA algorithm (see Algorithm 1); (2) determining the minimum processing time bound (see Algorithm 5); and determining (3) the maximum processing time bound (see Algorithm 5). We wrote this three-step procedure in C# language, and tested two examples using a

Presario X1000^{TM} laptop computer with a 1.5 GHz Intel Centrino ^{TM} CPU and 1 G-byte of RAM. The C# source codes, two examples shown in Sections 5.1 and 5.2, and as the simulation and verification engine are all available through the Open Source Community at sourceforge.net (Hwang, 2007).

5.1. Crosswalk Controller in Example 2

The crosswalk controller, `CC`, shown in Example 2 was analyzed in the first experiment to determine the min/max processing time bounds from the push-button event `?p` to the green light event at `!g:1`.

In this experiment, we increased the scanning frequency of the component `W` by decreasing its lifespan value of τ(`D`). The values used are displayed in the first column of Table I. The values in columns $|V|$, $|E|$, Mem., and C_G of Table I display the number of vertices in the TLA graph G of `CC`, the number of edges G in the TLA graph G, the peak memory required for generating G in KBytes, and the CPU time used for generating G in seconds, respectively.

The two values in the column $[Min, Max]$ show the minimum and the maximum processing time bounds calculated by the `MinPT` and `MaxPT` algorithms, respectively. We can see that `MinPT`(`CC`, `?p`, `!g:1`)=31 and `MaxPT`(`CC`, `?p`, `!g:1`)=61 for all values of $\tau_{\mathtt{W}}(D)$. The columns labeled C_{Min} and C_{Max} of Table I display the CPU times required to complete the `MinPT` and `MaxPT` algorithms in the form of `mm:ss.s` where `ss.s` is in seconds and `mm` is in minutes (if `mm`>0).

Two values in the last column $[Min, Max]_s$ were obtained from a simulation run whose running time was 100,000 time units. For all cases of $\tau_{\mathtt{W}}(D)$, the sampled simulation case fell within the specific time bounds since $[Min, Max]_s \subseteq [Min, Max]$.

As we showed in the `TLA` algorithm in Section 3.2.2, the complexity of the TLA graph, indicated by the number of its vertices and edges, increased when the greatest common divisor of $\tau(s_i)$ for all $s_i \in S_i$ for all $i \in D$ decreased. This phenomenon was demonstrated in this example because when the value of $\tau_{\mathtt{W}}(D)$ decreased, the size of TLA(`CC`) increased. This in turn increased all computations of `MinPT` and `MaxPT` since their computational burdens are proportional to the size of TLA(`CC`).

5.2. Monorail System

We examined the effect of the number of sub-components in a SP-DEVS network by varying the number of stations in a monorail system.

Figure 8(a) shows the coupling diagram of the monorail system we are considering in this example. We varied the number of stations from

Table I. Experiment for Finding Min/Max Processing Time Bounds in Example 2's CC with (z_s, z_e)=(?p, !g:1)

$\tau_{\mathtt{W}}(D)$	$\|V\|$	$\|E\|$	Mem.	C_G	$[Min, Max]$	C_{Min}	C_{Max}	$[Min, Max]_s$
1	101	133	228	0.1	[31, 61]	0.1	0.1	[31.02, 60.97]
0.1	670	973	1,476	1.0	[31, 61]	2.1	0.5	[31.01, 60.99]
0.01	6,340	9,343	3,308	12.5	[31, 61]	5:15.0	45.4	[31.01, 60.99]

$\tau_{\mathtt{W}}(D)$: $\tau(D)$ of W component; $|V|$: The number of vertices in $G = TLA(M)$; $|E|$: The number of vertices in G; Mem: Peak memory required to generate G in KBytes; C_G: CPU time to generate G in seconds; $[Min, Max]$: $Min = MinPT(L(M, z_s, z_e))$ and $Max = MaxPT(L(M, z_s, z_e))$; C_{Min} (resp. C_{Max}): CPU time to compute the minimum (resp. the maximum) processing time bound, in minutes:seconds; $[Min, Max]_s$: the minimum and the maximum processing time bounds sampled by a simulation run which was terminated after 100,000 time-units.

3 to 6 while holding the number of vehicles constant at two. The stations were named $\mathtt{ST}_i$ for $i = 0$ to 5. Each station has input and output events such as ?v and !v for vehicles, and ?p and !p for the pull-signals. In addition, ?a is the input event of "*additional* loading request. Vehicle routing is circular clockwise for all vehicles. In addition, the couplings for the pull-signal are also connected between consecutive stations in order to prevent vehicle collisions.

Figure 8(b) illustrates the dynamics of a station. The status of a station is modeled by three state variables: (1) phase $\in$ { Empty (E), Empty_to_Loading (EL), Loading (L), Sending (S), Pulling (P), Pulling_to_Loading (PL), W (Waiting), Waiting_to_Sending (WS), Waiting_to_Loading (WL), Collided (C) }; (2) vid $\in \{0, 1, 2\}$ which memorizes the vehicle id that occupies the station such that vid=1 for vehicle1, and vid=2 for vehicle2, vid=0 for no vehicle; and (3) nso $\in$ {true(t), false(f) } indicating "next station is NOT occupied" for nso=f or "next station is occupied" for nso=t. Thus if ST1 has the state vector (phase, vid, nso)=(L, 1, t), that means ST1 is in the loading phase with the first vehicle and ST2 is occupied by a vehicle.

In Figure 8(b), the phase of a state, s and the lifespan of the state given by $\tau(s)$ are represented as a *node*, while vid and nso are used to described the transition-edge firing condition in the form of "(*pre-condition*),(*post-condition*)". For example, as an external state transition, the arc from W to WS augmented by (?p)(nso := t;) means that when the phase variable is W and an input event ?p comes into the station, the variable nso is set to t and the phase variable changes into WS. As an output and internal state transition, the arc from L to W with (nso := t), (ϕ) stands for when the phase variable of a state, s,

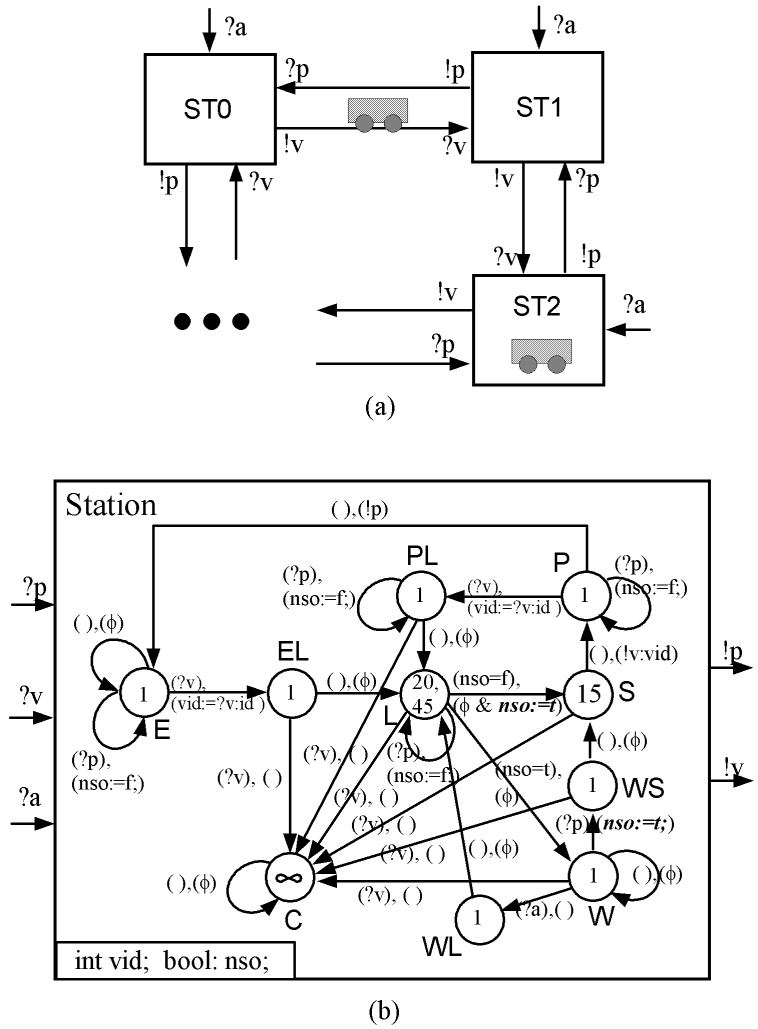

Figure 8. Monorail System

is given by `L` and the elapsed time of s equals the lifespan of the state s, if `nso` is `t` then the phase variable changes into `W` without an output event.

Initially, `ST0` and `ST1` contain vehicle1 and vehicle2, respectively. So the initial state (`phase`, `vid`, `nso`) for each station was set as follows: (`W`, 1, `t`) for `ST0`, (`W`, 2, `t`) for `ST1`, (`P`, 0, `t`) for `ST2`, and (`P`, 0, `t`) for `ST3` in the case that four stations are involved. In addition, to see the waiting behavior explicitly, we used 45 seconds for τ(`L,vid,nso`) (where `vid` $\in \{1, 2\}$ and `nso` $\in \{$`t`, `f`$\}$) for `ST1`, 20 seconds for τ(`L,vid,nso`) for the rest other stations.

Table II. Experiment for Finding Min/Max Processing Time Bounds of a Monorail System with (z_s, z_e)=(!ST0.v:1, ?ST0.v:1) and (z_s, z_e)=(!ST0.v:2, ?ST0.v:2)

n	$\|V\|$	$\|E\|$	Mem.	C_G	$[Min, Max]$	C_{Min}	C_{Max}	$[Min, Max]_s$
3	5,061	5,328	4,568	21.7	[100, 140]	1.8	0.4	[101, 139]
					[100, 140]	1.8	0.4	[101, 139]
4	11,889	12,139	9,620	1:20.6	[130, 175]	5.6	1.1	[130, 166]
					[130, 158]	1.9	0.2	[130, 130]
5	21,458	21,729	15,788	3:23.7	[165, 210]	9.8	2.1	[165, 174]
					[165, 165]	3.1	0.1	[165, 165]
6	34,347	34,681	24,600	7:13.9	[200, 245]	16.7	3.7	[200, 231]
					[200, 200]	4.4	0.2	[200, 200]
3^*	3,453	3,547	3,876	14.3	[95, ∞]	1.8	0.1	$[\infty, \infty]$
					[95, ∞]	1.5	0.1	$[\infty, \infty]$

n= the number of stations used; * the post condition `nso:=t` was ignored; $|V|, |E|$, Mem, C_G, $[Min, Max]$, C_{Min}, C_{Max}, and $[Min, Max]_s$ have the same meanings as in Table I.

We tested the minimum and maximum processing bounds from departure to arrival at `ST0` for each vehicle. For each test case, the number of stations n, $|V|$, $|E|$, Mem., and C_G are shared for both vehicle1 and vehicle2 in Table II. But in the columns labeled C_{Min}, C_{Max}, $[Min, Max]$ and $[Min, Max]_s$, the top row for each vehicle of n reports results for vehicle1, while in the bottom row reports results for for vehicle2. For example, if n=4, $[Min, Max]$ for vehicle1 is [130, 175], while that for vehicle2 is [130, 158].

We can observe that increasing the number of stations increases the complexity of the TLA graph, as was proven shown in Section 3.2.2. One of the interesting phenomena found in this example is that the processing time bounds for vehicle2 does not change, when the number of stations increase to 5 and 6. This can be explained by noting that initially vehicle2 is at station$_1$ ahead of the first vehicle so once the phase of station$_1$ becomes `W`, it can changes to `WtS` immediately. However, the station$_0$ which has vehicle1 is seem to be `W` for a while during the circulation. In this monorail example, we did not find any counter example such that a simulated processing time was outside the boundary of $[Min, Max]$ for station numbers $n \in \{3, 4, 5, 6\}$.

In addition, the last row marked 3^* shows the case that the post-condition (`nso:=t`) of the two arcs `L` to `S` and `W` to `WS` in Figure 8(b)

was intentionally ignored. In that case, the maximum processing time became ∞ which meant there was a possibility of collision along the vehicle route. This example demonstrates that our approach can be applied to the non-steady state system (such as the collision case) as well as to steady systems (such as the non-collision cases).

6. Conclusion

We conclude this paper with a short summary, a discussion of contributions of this paper, and suggestions for possible future work in the area of SP-DEVS.

6.1. Summary

This paper showed a sub-class of DEVS, called SP-DEVS, whose structure and dynamics were described in Section 2. Section 3 showed that the behavior of SP-DEVS can be represented by a finite-vertex graph using a time-abstracting technique. Finally, in Section 4, quantitative analyses using SP-DEVS were performed to evaluate the shortest time and the longest time possible for event sequences which start and end with two specific events. Section 5 presented two experimental examples which illustrated the process of computing processing time bounds.

In addition, the questions Q1, Q2, and Q3 which arose in the introduction were answered as follows.

Q1. Can we construct a finite-state reachability graph that represents the timed-behavior of a given SP-DEVS model? How complex is this construction process?

A1. In section 3, we saw that the time-line abstraction (TLA) process constructs a finite graph structure whose behavior is the same as that of the original SP-DEVS model. In Section 3.2.2, the complexity of the TLA operation (investigated in Algorithm 1) was shown to be exponentially bounded by the number of involved models, the number of states specified for each model, and the lifetime of each state of each model.

Q2. Can we determine the minimum time span between two events? How complex is the evaluation process

A2. In Section 4.3.1, the existence of a shortest time span for a given event sequence was proven by Theorem 2. Based-on Theorem 2 and Dijkstra's one-source shortest path algorithm, we developed Algorithm 3 to compute the minimum processing time between

two events. We also showed that the complexity of Algorithm 3 is polynomially bounded by $O(|E|^2 \log |V|)$.

Q3. Can we determine the maximum time span between two events? How complex is the evaluation process?

A3. In Section 4.3.2, the existence of a longest time span for a given event sequence was proven by Theorem 3. Based-on Theorem 3 and the topological sorting algorithm (Sedgewick, 2002), we developed Algorithm 5 to compute the maximum processing time between two events. We also showed that the complexity of Algorithm 5 is polynomially bounded by $O(|E|^2|V|)$.

6.2. Contributions

This paper contributes to two communities. One is the DEVS research community, the other is the broader DEDS research community. So far, the DEVS research community seems to have had no clear idea of how to establish the processing time bounds of a given DEVS model with a view towards verification. Even though there have been state-space analysis papers (Hong and Kim, 1996), (Hwang and Zeigler, 2006b), they evaluated the problem with the qualitative analysis techniques such as deadlock and livelock, but not with quantitative analysis designed to answer the question, "how long will it take between two events?". There has been some research in computing a time constraint satisfying certain reachability conditions (Zeigler and Chi, 1992), but that paper assumed a closed system in which there was no arbitrary input event. This paper eliminates the need for such an assumption.

From the view point of the broader DEDS research community, there are two kinds of performance analysis which do not use simulation. One technique uses steady-state performance analyses such as Generalize Petri-net(M. Ajmone Marsan, G. Balbo, G. Conte, 1984), (Wang, 1998) and Queueing Theory (Gross and Harris, 1998). These analyses assume that the system behavior repeats in a certain cyclic pattern. But in practice, we don't know if a system repeats patterns or not, and if it does, we don't know what the patters are. A second category of performance analysis is a reachability graph approach. This is the category to which this paper belongs. This second approach allows the analyst to obtain all processing time bounds without having to make the steady-state assumption. In this category, (Courcoubetis and Yannakakis, 1992) showed a methodology for evaluating the minimum and maximum performance bounds by using Timed Automata. However, this methodology needs a finite-vertex reachability graph, which the

author call a *region graph* whose "exact" computational complexity (not the worst case complexity) is so huge (Alur and Dill, 1994), that the approach needs improvement for practical applications. A further complication arises in that the minimized reachability graph of Timed Automata using the bi-simulation relation (Courcoubetis and Yannakakis, 1992; Tripakis and Yovine, 2001) has not been proved to apply to the method addressed in (Courcoubetis and Yannakakis, 1992).

6.3. Future Works

We can look forward to two main streams of work to be extended from this paper. One is an extension within SP-DEVS, the other is an extension to the higher verifiable classes of DEVS.

In the first category, behavior segments which are more complicated than the two-event segments of this paper need be described and verified in terms of their processing time bounds. To do this, the computational tree temporal logic (CTTL) (Alur et al., 1993) might be a challenging research topic.

In the second category, the three-step approach of this paper can be extended to finite-deterministic DEVS (FD-DEVS) (Hwang and Zeigler, 2006b; Hwang and Zeigler, 2006a). Since the methodology for obtaining the reachability graph of FD-DEVS has been researched (Hwang and Zeigler, 2006c), one topic remaining to be resolved is that of evaluating the event sequences.

Appendix: Getting an Acyclic Subgraph from a TLA graph

`Get_DAG` has the following three input arguments: 1) a TLA graph, G; 2) a starting edge e_s; and 3) an ending event z_e; and it has one output argument, as the sub-graph G_{DAG} of G. The return type of `Get_DAG` is boolean, and `Get_DAG` returns `false` if there is a path terminating with an transition edge whose triggering event is not z_e. Otherwise, it returns `true`.

`Get_DAG` uses the set of all vertices visited, `VV`, and a set of edges to be tested, `ET`. The starting edge e_s and the starting vertex of e_s are initially added to `ET` and G_{DAG}, respectively, as shown in lines 5 to 7.

`Get_DAG` picks an edge e in `ET` and constructs a sub-graph reachable from e_s (see lines 9 to 11 of Algorithm 6). If the ending vertex v' of testing edge $e = v \xrightarrow{z} v'$ was not visited before, it adds v' to `VV` (lines 12 to 13). And then if the ending vertex v' has no outgoing edge (in other words, v' is a terminal node), `Get_DAG` returns `false` because there is a path which terminates with a non-z_s event (lines 14 to 15). Otherwise,

for each outgoing edge $e' = v' \xrightarrow{z} v''$ from v', if the triggering event z of e' is equal to z_s, the algorithm skips further searching from v''. Otherwise, the algorithm continues further searching by adding e' to `ET` (lines 17 to 18). The searching statements in lines 8 to 18 continue as long as `ET` is not empty.

Since $|E|$ of G is finite, the loop through lines 8 to 19 eventually terminates. If the loop doesn't break at line 15 by returning `false`, Algorithm 6 return `true` at line 19. The complexity of the `Get_DAG` algorithm is $O(|V| + |E|)$, which is the complexity of the depth-first search algorithm that we used (Sedgewick, 2002).

Algorithm 6 Get Acyclic Subgraph of G

```
 1 bool Get_DAG(↓ G, ↓ e_s, ↓ z_e, ↑ G_DAG)
 2   //G =< X, Y, V, v_0, E >;
 3   //e_s = (v -z-> v') starting transition edge;
 4   //z_e ending event;
 5   Set VV; // vertices visited
 6   Set ET; add e_s to ET; // edges to be tested
 7   Graph G_DAG;
 8   while ET ≠ ∅
 9     e := ET.pop_front(); // e = (v -z-> v')
10     add v' to G_DAG.V;
11     add e to G_DAG.E;
12     if(e.v' ∉ VV )
13       add e.v' to VV;
14       if (e.v' has no outgoing edge)
15          return false;
16       foreach e' = (v' -z-> v'')
17         if (e'.z != z_e)
18           add e' to ET;
19   return true;
```

Acknowledgements

Many thanks to Dr. Russ Mayers who read this entire document and suggested some constructive ideas.

References

Alur, R., Courcoubetis, C., and Dill, D. (1993). Model-checking in dense real-time. *Information and Computation*, 104(1):2–34.

Alur, R. and Dill, D. (1994). A theory of timed automata. *Theoretical Computer Science*, 126:183–235.

Batt, G., Bradley, J. T., Ewald, R., Fages, F., Hermans, H., Hillston, J., Kemper, P., Martens, A., Mosterman, P., Nielson, F., Sokolsky, O., and Uhrmacher, A. M. (2006). 06161 working groups' report: The challlenge of combining simulation and verification. In Nicol, D. M., Priami, C., Nielson, H. R., and Uhrmacher, A. M., editors, *Simulation and Verification of Dynamic Systems*, number 06161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2006/724> [date of citation: 2006-01-01].

Courcoubetis, C. and Yannakakis, M. (1992). Minimum and maximum delay problems in real-time. *Formal Methods in System Design*, 1(4):385–415.

Gross, D. and Harris, C. M. (1998). *Fundamentals of Queueing Theory.* Wiley-Interscience, 3rd edition.

Ho, Y.-C. (1993). Forward to the Special Issue. *Discrete Event Dynamic Systems:Theory and Applications*, page 111.

Hong, G. P. and Kim, T. G. (1996). A Framework for Verifying Discrete Event Models whithin a DEVS-based System Development Methodology. *Transactions of The Society for Computer Simulation*, 13:19–34.

Hwang, M. H. (2007). DEVS#: C# Open Source Library of DEVS Formalism. http://xsy-csharp.sourceforge.net/DEVSsharp/.

Hwang, M. H. and Zeigler, B. (2006a). Expressiveness of Verifiable Hierarchical Clock Systems. *International Journal of General Systems.* to appear, available at http://acims.arizon.edu/.

Hwang, M. H. and Zeigler, B. P. (2006b). A Modular Verification Framework using Finite & Deterministic DEVS. In *Proceedings of 2006 Spring Simulation Multi-Conference: Proceedings of 2006 DEVS Symposium*, pages 57–65, Huntsville, AL. SCS.

Hwang, M. H. and Zeigler, B. P. (2006c). A Reachable Graph of Finite and Deterministic DEVS Networks. In *Proceedings of 2006 Spring Simulation Multi-Conference: Proceedings of 2006 DEVS Symposium*, pages 48–56, Huntsville, AL. SCS.

Lewis, R. W. (1998). *Programming Industrial Control System Using IEC 1131-3.* The Institution of Electrical Engineers, revised edition.

M. Ajmone Marsan, G. Balbo, G. Conte (1984). A class of generalized stochastic petri nets for the performance analysis of multiprocessor systems. *ACM Transactions on Computer Systems*, 2(1):93–122.

Sedgewick, R. (2002). *Algorithms in C++, Part 5 Graph Algorithm.* Addison Wesley, Boston, third edition.

Skahill, K. (1996). *VHDL for Programmable Logic.* Prentice Hall.

Tripakis, S. and Yovine, S. (2001). Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18:25–68.

Wang, J. (1998). *Timed Petri Nets: Theory and Application.* Kluwer Academic Publishers, Boston.

Zeigler, B. P. (1976). *Theory of Modeling and Simulation.* Wiley Interscience, New York, first edition.

Zeigler, B. P. (1984). *Multifacetted Modeling and Discrete Event Simulation.* Academic Press, London,Orlando, first edition.

Zeigler, B. P. (1990). *Object-oriented Simulation with Hierarchical, Modular Models:Intelligent Agents and Endomorphic Systems*. Academic Press, Boston, Mass. and San Diego, Calif., second edition.

Zeigler, B. P. and Chi, S. (1992). Symbolic Discrete Event System Specification. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6):1428–1443.

Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, London, second edition.