Proceedings of the 2023 Winter Simulation Conference
C. G. Corlu, S. R. Hunter, H. Lam, B. S. Onggo, J. Shortle, and B. Biller, eds.

NEW FUNCTIONS AND STATEMENTS TO SUPPORT PREEMPTION IN THE STROBOSCOPE SIMULATION SYSTEM

Photios G. Ioannou

Dept. of Civil and Environmental Engineering
University of Michigan
2350 Hayward Street
Ann Arbor, MI 48109-2125, USA

Veerasak Likhitruangsilp

Dept. of Civil Engineering
Faculty of Engineering
Chulalongkorn University
Bangkok, THAILAND

ABSTRACT

The new preemption capabilities added to the STROBOSCOPE simulation system are described and illustrated by two examples. The first example involves moving soil using two wheelbarrows and two laborers. It investigates the conditions for preemption to improve production by allowing the return of an empty wheelbarrow to interrupt loading and to start hauling a partially loaded wheelbarrow immediately. In the second example, two cranes unload barges bringing fill material for undersea land reclamation. When only one barge is available, it can unload using both cranes. When two or more barges become available, each barge unloads using one crane. Unloading a barge can switch between using one and two cranes multiple times, with the remaining unload time either cut in half or doubled each time. Modeling the multiple reallocations of cranes and the required time adjustments illustrates the new STROBOSCOPE preemption capabilities.

1 INTRODUCTION

1.1 Overview of the STROBOSCOPE Simulation System

STROBOSCOPE (an acronym for State and Resource Based Simulation of Construction Processes) is a general-purpose discrete-event simulation system based on the activity-scanning paradigm that is particularly suited for modeling complex construction operations. A STROBOSCOPE simulation model is represented by a graphical network of nodes and links (similar to an activity cycle diagram) and is described in detail by statements written in the STROBOSCOPE simulation programming language.

STROBOSCOPE is a Windows application written in C++ that consists of an Integrated Development Environment (IDE) that includes an editor for entering the text-based statements for a simulation model using the STROBOSCOPE language, and a simulation engine (a DLL) that parses and interprets the input, performs the simulation, and sends the simulation output back to the IDE.

STROBOSCOPE also provides a Graphical User Interface (GUI) implemented in Microsoft Visio using smart drag-and-drop graphics and a custom Visio add-on (a separate DLL written in C++) that provides tabbed dialog-boxes for data input, extensive error-checking, and the ability to compile and communicate to STROBOSCOPE all the information entered. The GUI makes it possible to develop and save complete STROBOSCOPE simulation models, including all statements, entirely within Visio files. The networks for the simulation models shown in this paper are live models developed with the STROBOSCOPE GUI.

Resources, such as material, labor, and equipment, can be modeled as generic, characterized, or compound resources in STROBOSCOPE. Generic resources have distinct types. Characterized resources have types and subtypes with static properties. Compound resources can contain other resources (generic, characterized, or even other compound resources) to any level. Both characterized and compound resources are distinct objects that can also have dynamic properties (SaveProps) and methods (VarProps).

The STROBOSCOPE language is quite rich and provides many statement types, predefined object types, built-in functions, predefined variables, events, actions, etc. These facilities provide access to dynamic variables, the properties of resources, and the state of the simulation, and allow, for instance, modeling stochastic resource production, utilization, and consumption, smart resource allocation, characterization of resources created at runtime by combining other resources, and dynamic decisions regarding the flow of resources and the sequence of activities.

The STROBOSCOPE program, its documentation, and several complete examples are available from (Martinez and Ioannou 2023). More than 20 educational videos about learning and using STROBOSCOPE are available on YouTube (https://www.youtube.com/@PhotiosIoannou).

1.2 Preemption in Simulation

Simulation models often need to interrupt or preempt an activity in progress while the simulation is running and reallocate the preempted activity's resources to another activity that is started at that time (Law 2014). The concept of preemption is not new in the general area of simulation and is supported by several commercial systems (often used in industries such as manufacturing and healthcare) such as AnyLogic, Arena, eM Plant, Extend, FlexSim, GPSS, Quest, SimCAD, WorkBench, and others. The ease and completeness by which commercial software have implemented preemption, however, varies widely with scores ranging from 0 to 4 (out of a maximum of 5) as graded by a study that evaluated several commercial discrete event simulation systems (Zapata et al. 2008).

In construction simulation there have been no attempts to implement preemption at the simulation system level. The notable exception is an implementation of a form of preemption by Rekapalli (2008) for the purpose of enabling concurrent visualization simulations through the Novoscope DLL. At the level of a construction simulation model, it has been shown that it is possible to implement preemption without the new preemption facilities added to STROBOSCOPE (Ioannou and Kamat 2005). However, such attempts have also shown that implementing preemption *correctly* without direct system support is considerably more challenging and can easily lead to modeling errors that are not easy to detect. As illustrated by the models in this paper, the direct support of preemption in STROBOSCOPE, a widely used activity-based construction simulation system, provides new capabilities to all users and allows the development of novel simulation models for construction operations in a clear and straightforward manner.

2 NEW STROBOSCOPE FUNCTIONS AND EVENTS TO SUPPORT PREEMPTION

2.1 Preemption in STROBOSCOPE

In the STROBOSCOPE simulation system (Martinez and Ioannou 2023), preemption has been implemented as the immediate termination of an activity instance that currently resides in the Future Event List (FEL) and the start of a new instance of another activity that takes over all the resources of the preempted instance. The new functions and statements that have been added to STROBOSCOPE to support preemption directly make it significantly easier to develop simulation models that implement preemption *correctly*. These new preemption capabilities are illustrated through their application to two examples. In the first, laborers move soil using wheelbarrows, and in the second, cranes unload barges carrying fill material for undersea land reclamation. These examples also serve as case-studies that provide insights as to how to improve construction operations through the prudent use of preemption.

Due to space limitations, only the new statements and functions related to preemption are described in detail. The notation used for the STROBOSCOPE statements in the examples that follow is to show identifiers chosen by the modeler in italics, such as "*Load*" in "*Load*.TotInst.-1", while regular text indicates reserved STROBOSCOPE identifiers that must be entered as shown. It should also be noted that the arguments of STROBOSCOPE functions (such as Preempt) are enclosed in square brackets and not in parentheses. More information about the STROBOSCOPE language can be found in (Martinez 1996).

2.2 Interrupting an Activity Instance in Progress — The "Preempt" Function

Preemption has been implemented in STROBOSCOPE through the new function Preempt that can be called at appropriate trigger events while a simulation model is running to interrupt an activity instance that is currently in progress. The syntax of the function Preempt and the actions it takes are summarized below.

Syntax: Preempt[*CurAct*, *InstNum*, *NewAct*]
Example: Preempt[*Load*, *Load*.TotInst-1, *StopLoad*]
Returns: Either the value 1 (success) or the value 0 (failure)

When the function Preempt is called, it takes the following actions:

- It scans the Future Event List (FEL) to find the instance of activity *CurAct* (e.g., *Load*) whose instance number (i.e., 0, 1, 2, …) is given by the number *InstNum*. For example, *Load*.TotInst-1 is the number of the last instance of *Load* that has been created. It should be noted that the instances of an activity are given consecutive integer numbers with the first instance having the number 0. The global variable *CurAct*.TotInst returns the total number of instances of activity *CurAct* up to now. Thus, the last instance of *CurAct* has the number *CurAct*.TotInst-1 and can be preempted by Preempt[*CurAct*, *CurAct*.TotInst-1, *NewAct*].
- If the instance of activity *CurAct* with number *InstNum* is not found in the FEL, then the function Preempt stops and returns the value 0 (false) to indicate failure.
- If the instance of activity *CurAct* with number *InstNum* is found in the FEL, then the function Preempt creates a new instance of activity *NewAct* (whose instance number is *NewAct*.TotInst-1) that takes over all the resources of the preempted instance of activity *CurAct*.
- The duration of the new instance of activity *NewAct* is set equal to the remaining duration of the preempted instance of *CurAct*.
- The statistics for activity *CurAct* are corrected by un-collecting the original duration of the preempted instance and collecting its truncated duration.
- Instance number *InstNum* of activity *CurAct* is terminated and the function Preempt stops and returns the value 1 (true) to indicate success.

Because the function Preempt is a logical function that returns either 1 (success) or 0 (failure), it can be used in the logical expressions that control the actions of other statements. In the models that follow, for example, the function Preempt is used in the logical preconditions that determine whether other actions will indeed take place at certain events during simulation.

2.3 Resetting the Duration of an Existing Activity Instance — The "SetTmLeftInst" Function

By default, when the function Preempt creates a new instance of activity *NewAct* (with instance number *NewAct*.TotInst-1) it sets its duration equal to the remaining duration of the preempted instance of *CurAct* (with number *InstNum*). Often, however, this is not the appropriate duration required by the model.

The new function SetTmLeftInst can be called at appropriate events during simulation to reset the remaining duration of any activity instance (that is currently in progress) to any desired value. The syntax of the function SetTmLeftInst and the actions it takes are summarized below.

Syntax: SetTmLeftInst[*Activity*, *InstNum*, *RemDur*]
Example: SetTmLeftInst[*StopLoad*, *StopLoad*.TotInst-1, 0]
Returns: Either the value 1 (success) or the value 0 (failure)

When the function SetTmLeftInst is called, it takes the following actions:

- It scans the Future Event List (FEL) to find the instance of *Activity* (e.g., *StopLoad*) whose instance number is *InstNum*. For example, *StopLoad*.TotInst-1 will find the last instance of *StopLoad*.
- If an instance of *Activity* with number *InstNum* is not found in the FEL (because it was never created or because it has already finished), then the function SetTmLeftInst stops and returns 0 (failure).
- If the instance of *Activity* with number *InstNum* is found in the FEL, then the function SetTmLeftInst changes the instance's duration to *RemDur* and corrects its end-event time in the FEL so that the instance will now finish at SimTime + *RemDur*.
- The duration-related statistics for *Activity* are corrected by un-collecting the instance's original duration and collecting its new duration *RemDur*.
- The function SetTmLeftInst ends and returns the value 1 (success).

Like the function Preempt, the function SetTmLeftInst is a logical function that returns the value 1 or the value 0 and can be used in logical expressions to control other statements in the model.

2.4 New Action Events Related to the "Preempt" Function

When an activity instance is preempted successfully, it is often necessary to take additional actions—for example, to store new values in global SaveValues or to store new data in the private SaveProps of specific characterized resources. The following two new action events occur only if the function Preempt is successful. These events occur after the function Preempt has found the instance to preempt in the FEL, and before the function Preempt finishes and returns the value 1 (success).

Syntax: BEFOREPREEMPTED *CurAct* [*Action*] *ActionTarget* *TargetArgs*;

Timing: Occurs after the function Preempt has corrected the duration-related statistics of *CurAct* and before the current instance count of *CurAct* is reduced by one.

Syntax: AFTERPREEMPTING *NewAct* [*Action*] *ActionTarget* *TargetArgs*;

Timing: Occurs after the preemption of CurAct is complete and after the new instance of NewAct has been created and has taken over the preempted resources and the remaining duration of CurAct.

The action events BEFOREPREEMPTED or AFTERPREEMPTING can be triggered by the action event for another activity that calls the function Preempt (such as, BEFOREEND *TriggerAct*). In this case, both the instance of *TriggerAct* and the instance of *CurAct* (or *NewAct*) are in context. This is important because in this situation the attributes and resources of both the instance of *TriggerAct* and the instance of *CurAct* (or *NewAct*) are accessible, such as through the new activity instance variable *Activity*.StartTm described next.

2.5 Accessing the Start Time of an Activity Instance — "Activity.StartTm" Instance Variable

It is often necessary to access the start time of a particular instance of an activity that is currently in context in order to calculate, for example, the duration of that instance up to the current simulation time, or to use this start time to set the attributes (such as the duration) of another activity instance later.

The new instance variable *Activity*.StartTm provides the start time of any activity instance (such as *CurAct* or *NewAct*) that is currently in context. The requirement for an activity instance to be in context is important because it allows the system to determine unambiguously which of the several instances of *Activity* that may be in progress is being referenced.

The global variable *Activity*.LastStart, though similar, has a different purpose. It can be used at any time to return the start time of the last instance of *Activity* that was ever created, even if that last instance is not

in context or has already finished. In contrast, the instance variable *Activity*.StartTm can return the start time for any instance of *Activity* that is in progress and in context (and not necessarily just the last one).

Appropriate uses of the instance variable *Activity*.StartTm, as well as applications of the new preemption functions, Preempt and SetTmLeftInst, and the new action events BEFOREPREEMPTED and AFTERPREEMPTING are illustrated by the following two example models.

3 SIMULATION OF MOVING SOIL BY WHEELBARROW

In this apparently simple example, two laborers use two 6-cf wheelbarrows to move soil. The *Load* activity uses the first laborer (the *Loader*) to load one of the wheelbarrows with soil. The *Haul* activity uses the other laborer (the *Hauler*) to push a loaded wheelbarrow, dump the soil, and return the empty wheelbarrow to load again. The durations of the *Load* and *Haul* activities L and H (in minutes) are balanced and follow normal distributions with the same mean value of 2 min. However, they have different standard deviations $SD[L]$ and $SD[H]$. This example investigates whether the deliberate preemption of the *Load* activity by the *Haul* activity can improve long-term production (in cf/min) by comparing the following two policies A and B while varying the standard deviations $SD[L]$ and $SD[H]$.

Policy A: A wheelbarrow can be hauled only when fully loaded with 6 cf of soil. When activity *Haul* finishes first, it waits for activity *Load* to finish before both can start again.

Policy B: If a wheelbarrow is still being loaded when activity *Haul* finishes, then the loading of that wheelbarrow stops, and *Load* starts again with the empty wheelbarrow that has just arrived. At the same time, the *Haul* activity starts again with the partially loaded wheelbarrow whose load is 6 cf times its truncated load time over the total time that would have given a complete load of 6 cf.

3.1 Wheelbarrow Cycle Analysis

It is interesting to note that under both policies A and B, the activities *Load* and *Haul* start at the same time in each cycle but have different durations L and H (in min.). The two policies also have different cycle times. The cycle time for policy A is $\max(L, H)$ whereas the cycle time for policy B is H.

For each policy, the production in each cycle $r(\mathrm{A})$ and $r(\mathrm{B})$ (in cf/min) depends on whether $L < H$ or $L > H$.

$$\text{Policy A:} \quad \begin{cases} r(\mathrm{A}) = 6/H & \text{when } L < H \\ r(\mathrm{A}) = 6/L & \text{when } L > H \\ r(\mathrm{A}) = 6/\max(L,H) & \text{in any cycle} \end{cases}$$

$$\text{Policy B:} \quad \begin{cases} r(\mathrm{B}) = 6/H & \text{when } L < H \\ r(\mathrm{B}) = 6(H/L)/H & \text{when } L > H \text{ (the load } 6(H/L) \text{ is incomplete and less than 6 cf)} \\ r(\mathrm{B}) = 6/\max(L,H) & \text{in any cycle} \end{cases}$$

This analysis shows that the productions in each cycle $r(\mathrm{A})$ and $r(\mathrm{B})$ (in cf/min) for the two policies are identical and equal to $6/\max(L, H) = \min(6/L, 6/H)$. The difference is that under policy B some cycles carry less than 6 cf of soil, but also take less time. Based on this, the two policies would appear to be equivalent.

However, the long-term productions for the two policies, $R(\mathrm{A})$ and $R(\mathrm{B})$ (in cf/min), that are given by the ratio of the total volume of soil moved in n cycles over the total time of n cycles are quite different.

$$\text{Policy A:} \quad R(\mathrm{A}) = 6n \Big/ \sum_{i=1}^{n} \max(H_i, L_i)$$

$$\text{Policy B:} \quad R(\mathrm{B}) = 6 \sum_{i=1}^{n} \min(1, H_i / L_i) \Big/ \sum_{i=1}^{n} H_i$$

Clearly, the long-term productions $R(A)$ and $R(B)$ over n cycles are random variables given by different functions of the random variables L and H and cannot be compared analytically. A meaningful comparison of $R(A)$ and $R(B)$ can be made by simulation that uses both preemption and the *common random numbers* variance reduction technique (Ioannou and Martinez 1996).

3.2 STROBOSCOPE Simulation Model for Policies A and B

The STROBOSCOPE network for the simulation model for moving soil by wheelbarrows is shown in Figure 1. This network consists of nodes and links. Queues are shown as circles resembling the letter "Q" and each holds idle resources of a certain type such as *WheelBarrow*. Combi (conditional) activities are shown as clipped rectangles that require resources in order to start and are preceded by queues. Normal (bound) activities are shown as rectangles and can start whenever a direct predecessor activity finishes. Links connect the nodes and indicate the flow of the various types of resources in the model.

Described below are the new STROBOSCOPE statements that are needed to implement the preemption of activity *Load* by activity *Haul*. (The complete models are available from the first author.)

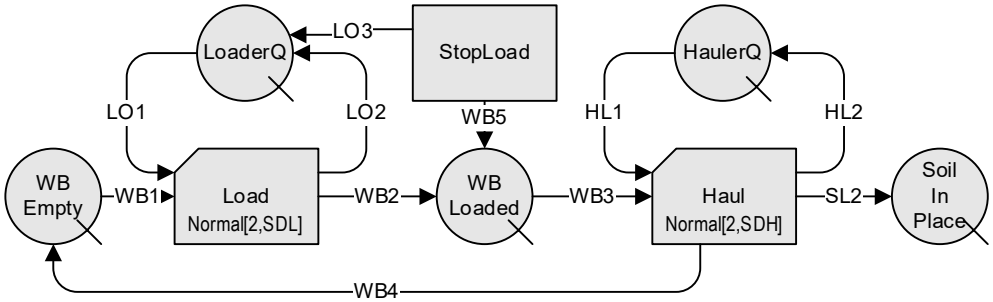

Figure 1: STROBOSCOPE model for earthmoving by wheelbarrow.

The model begins by defining the SaveValue *HaulPreemptsLoad* that determines which policy, A or B, will be simulated by the model.

```
SAVEVALUE HaulPreemptsLoad 1; / Changes policy: 0→A (No Preemption), 1→B (Preemption)
```

Three types of resources are defined. *Laborer* and *Soil* are modeled as generic resources. *WheelBarrow* is modeled as a compound resource. Its SaveProp *AmountLoaded* is used to store the amount of soil currently loaded in the *WheelBarrow*.

```
GENTYPE   Laborer;     GENTYPE   Soil;
COMPTYPE  WheelBarrow; SAVEPROP  WheelBarrow  AmountLoaded;
```

Each time an instance of activity *Load* starts, it stores 6 cf of soil in the SaveProp *AmountLoaded* of the *WheelBarrow* resource that is currently inside that *Load* instance. If this *Load* instance is later preempted, then the *AmountLoaded* in the *WheelBarrow* will be adjusted to less than 6 cf at the time of preemption.

```
ONSTART   Load   ASSIGN    Load.WheelBarrow.AmountLoaded    6;
```

Right before an instance of activity *Haul* ends, it attempts to call the function SetTmLeftInst. This call takes place only if its two logical preconditions (PRECOND) are satisfied. The first precondition is that *HaulPreemptsLoad* equals 1. The second is that the function Preempt can successfully preempt the last instance of activity *Load* and create a new instance of activity *StopLoad* to take over the resources (a *WheelBarrow* and a *Laborer*) and the remaining duration of the preempted instance of *Load*. If both preconditions return the value 1 (true), then the function SetTmLeftInst is called and resets the duration of the new instance of activity *StopLoad* to zero. This returns the resources *Laborer* and *WheelBarrow* to queues *LoaderQ* and *WBLoaded* and allows both activities *Load* and *Haul* to start again immediately.

```
BEFOREEND Haul   CALL
   PRECOND   'HaulPreemptsLoad   &   Preempt[Load, Load.TotInst-1, StopLoad]'
      'SetTmLeftInst[StopLoad, StopLoad.TotInst-1, 0]';
```

The following statement shows an example of the new event BEFOREPREEMPTED which occurs only if the instance of activity *Load* is indeed preempted. In that case, the *WheelBarrow* is loaded with less than 6 cf and the action taken is to adjust the SaveProp *AmountLoaded* in the *WheelBarrow* accordingly.

```
BEFOREPREEMPTED  Load   ASSIGN    Load.WheelBarrow.AmountLoaded
                        '6 * (SimTime - Load.StartTm) / Load.Duration';
```

In the above statement, the new instance variable *Load*.StartTm returns the start time of the instance of *Load* being preempted (and which is currently in context). Subtracting it from the current simulation time SimTime gives the instance's truncated duration. The instance variable *Load*.Duration returns the original duration of the preempted instance of *Load* that would have resulted in 6 cf being loaded into the wheelbarrow. Thus, 6 cf times the ratio of the truncated duration of the preempted instance of *Load* divided by its original duration gives the current amount of soil in the partially loaded wheelbarrow.

3.3 STROBOSCOPE Simulation Results

To obtain meaningful results, the complete STROBOSCOPE model uses the *common random numbers* variance reduction technique to compare the two policies A and B under identical streams of the durations *L* and *H* of activities *Load* and *Haul* (Ioannou and Martinez 1996). The model output includes the long-term productions *R*(A) and *R*(B) as well as their difference *R*(B)-*R*(A) in cf/min as shown in Figure 2.

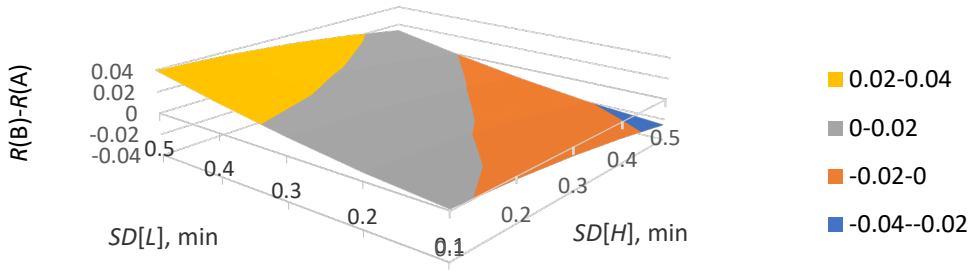

Figure 2: Difference in production *R*(B)-*R*(A) (in cf/min) vs *SD*[*L*] and *SD*[] (in min).

Figure 2 shows that the difference in the long-terms productions *R*(B)-*R*(A) is positive towards the left, negative towards the right, and close to zero along the front-to-back diagonal. Thus, the deliberate preemption of activity *Load* by activity *Haul* does improve long-term production, *R*(B) > *R*(A), when *SD*[*H*] < *SD*[*L*], i.e., when the duration of activity *Haul* has less variability than the duration of activity *Load*. This interesting result suggests that when two balanced activities interact (such as *Load* and *Haul*), it may be beneficial to keep the activity with the less variable duration working continuously (such as *Haul* when *SD*[*H*] < *SD*[*L*]) by interrupting if needed (and if possible) the other activity (e.g., *Load*) that has a more variable duration. Clearly, this is not always feasible as activity *Load*, for example, cannot interrupt activity *Haul*. This is an interesting observation that merits further investigation.

4 UNLOADING FILL MATERIAL FROM BARGES USING TWO CRANES

In this example, barges carrying construction fill material for undersea land reclamation arrive at the unloading facilities of a harbor. The interarrival times between barges are independent and exponentially distributed random variables with a mean of 1.25 days. The harbor has a dock with two berths and two cranes for unloading the fill material from the barges. When both berths are occupied, arriving barges join

a first-in-first-out (FIFO) queue where they wait to be unloaded. The time for one crane to unload a barge is uniformly distributed between 0.5 and 1.5 days. When there are two barges available to unload at the berths, then each barge is unloaded by a single crane. When only one barge is available to unload, then both cranes can work together on the same barge and its remaining unload time is cut in half. If in the meantime another barge arrives at the empty berth, then it takes away one of the cranes and unloading switches back to each barge using one crane. In this case, the remaining unload time for the first barge is doubled. The unloading of a barge can switch several times between using one or two cranes, with corresponding adjustments to its remaining unload time. This makes the unloading operation significantly more difficult to model without the new preemption facilities added to STROBOSCOPE.

4.1 Simulation Model—Unloading may be Preempted Multiple Times

The network for the STROBOSCOPE simulation model is shown in Figure 3. The STROBOSCOPE statements needed to implement preemption are described below.

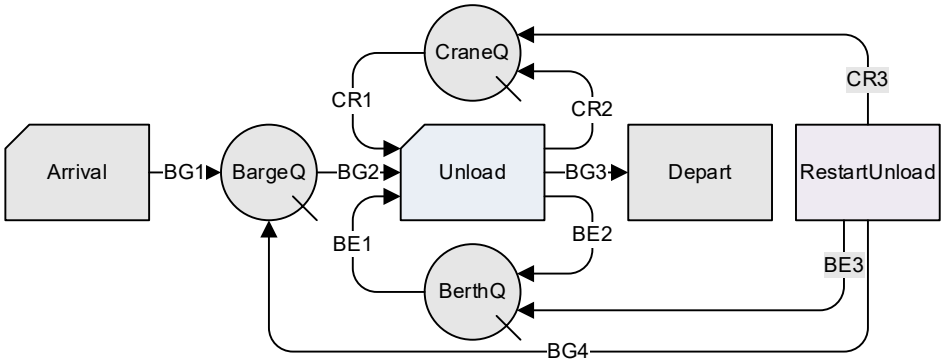

Figure 3: STROBOSCOPE model for unloading fill material from barges using two cranes.

Three types of resources are defined. *Berth* and *Crane* are generic resources. *Barge* is a compound resource with five SaveProps. *RemUnloadDur* stores the remaining unload duration if the work is done by a single *Crane*. *TimesPreempted* stores the number of times that the *Barge* was preempted. *CraneSeq* stores a number made up of the digits 1 and 2 that reflects the sequence of *Cranes* (1 or 2) that unloaded the *Barge*. *UnloadInst* is the number of the current instance of activity *Unload* for this *Barge*. *Docked* is a binary 0/1 value that when equal to 1 indicates that the *Barge* is currently unloading at the *Berths*.

```
GENTYPE     Berth;       GENTYPE   Crane;       COMPTYPE   Barge;
SAVEPROPS   Barge   RemUnloadDur   TimesPreempted   CraneSeq   UnloadInst   Docked;
```

The duration of activity *Arrival* is exponentially distributed. Its Semaphore ensures that activity *Arrival* can only have one instance currently going on during simulation.

```
DURATION     Arrival   'Exponential[1.25]';
SEMAPHORE    Arrival   '!Arrival.CurInst'; /One Arrival at a time
```

Each arriving resource of type *Barge* is created dynamically right before each instance of activity *Arrival* ends. The time needed to unload the *Barge* by a single *Crane* working alone is sampled from a uniform distribution and is stored in its SaveProp *RemUnloadDur* when the *Barge* flows though link *BG1*.

```
BEFOREEND    Arrival   GENERATE  1  Barge;
ONRELEASE    BG1       ASSIGN    RemUnloadDur 'Uniform[0.5, 1.5]';
```

A new instance of activity *Unload* draws a *Barge* from *BargeQ* and then *Cranes* from *CraneQ*. The number of *Cranes* drawn (1 or 2) through link CR1 depends on the current contents of queue *BargeQ*. If there are still *Barges* in queue *BargeQ* waiting to unload, then link *CR1* draws only 1 *Crane*. Otherwise, link CR1 draws all available *Cranes* (1 or 2) and passes them to the starting instance of activity *Unload*.

```
DRAWAMT      CR1    'BargeQ.CurCount? 1 : CraneQ.CurCount';
```

After a new instance of activity *Unload* draws resources, it determines its duration by dividing the SaveProp *RemUnloadDur* of its *Barge* by the number of *Cranes* (1 or 2) that it has drawn, and which will work together to unload the *Barge*.

```
DURATION     Unload    'Unload.Barge.RemUnloadDur / Unload.Crane.Count';
```

Each time a new instance of activity *Unload* starts, it stores its instance number in the SaveProp *UnloadInst* of its *Barge* so that this *Unload* instance may be identified later and preempted if needed.

```
ONSTART      Unload    ASSIGN    Unload.Barge.UnloadInst    Unload.Instance;
```

There are two cases when an instance of activity *Unload* needs to be preempted, and in both cases only one *Barge* would be currently unloading. These two cases are described below. In both cases, the multiple previous preemptions that activity *Unload* can have make it difficult to predict the correct instance number of *Unload* that should be preempted. Adding to this difficulty is the fact that *Barges* and *Unload* instances do not finish in sequential order because the total times needed for *Barges* to unload are random variables.

A straightforward way to determine the correct number for the instance of activity *Unload* that should be preempted is to define a filter that returns only one *Barge*— the *Barge* that is currently inside the instance of activity *Unload* to be preempted. This is accomplished by the following filter *Unloading* that creates a subset of all the *Barges* currently in the harbor whose SaveProp *Docked* equals 1.

```
FILTER  Unloading   Barge  Docked;
```

The following statements ensure that only one *Barge* (the one that is currently unloading and should be preempted) has a SaveProp *Docked* equal to 1. The global variable *Unloading.UnloadInst*.Value applies the filter *Unloading* to the entire population of *Barges*, finds the one *Barge* that needs to be preempted, and returns the value of its SaveProp *UnloadInst* that stores its current *Unload* instance number. This gives the correct number of the *Unload* instance to preempt (which is needed for the Preempt function).

The following statement uses the global variable *Unloading.UnloadInst*.Value as an argument to the function Preempt to preempt the correct ongoing instance of activity *Unload* (that it is currently using both *Cranes*) when a new *Barge* is released through link BG1 to the queue *BargeQ* and would need a free *Crane* to start unloading.

```
ONRELEASE   BG1   CALL PRECOND 'CraneQ.CurCount < BerthQ.CurCount'
          'Preempt[Unload, Unloading.UnloadInst.Value, RestartUnload]';
```

The precondition for the above statement ensures that the call to the function Preempt takes place only when the number of idle *Cranes* in *CraneQ* is less than the number of idle *Berths* in *BerthQ*. I.e., when the current instance of activity *Unload* is using both *Cranes* and should be preempted. This preemption allows the preempted *Barge* to start a new instance of activity *Unload* that uses one *Crane*, while the other *Crane* starts a second new instance of activity *Unload* that starts unloading the new *Barge* that has just docked.

The SaveProp *Docked* for a *Barge* that starts unloading is changed from 0 to 1 when the *Barge* is drawn through link BG2 by a new instance of activity *Unload*.

```
ONDRAW BG2 ASSIGN Docked 1;
```

When a *Barge* finishes unloading and is released through link BG5 to a new instance of activity *Depart*, its SaveProp *Docked* is changed from 1 back to 0, so that it will no longer pass the filter *Unloading*.

```
ONRELEASE BG3 ASSIGN Docked 0;
```

Activity *Depart* is a dummy activity with zero duration that occurs when a *Barge* finishes unloading and leaves the *Berths*. When an instance of *Depart* starts, it attempts to call the function Preempt if both logical preconditions for the call action return the value 1 (true).

```
ONSTART Depart CALL   PRECOND  'Unload.CurInst & !BargeQ.CurCount'
       'Preempt[Unload, Unloading.UnloadInst.Value, RestartUnload]';
```

The first precondition ensures that activity *Unload* does have a current instance. The second precondition ensures that *BargeQ* is empty, i.e., that the *Barge* still unloading is the only one left in the harbor. If both preconditions return 1 (true), then the function Preempt is called exactly as before to preempt

the ongoing instance of activity *Unload* and start a new instance of *RestartUnload*. Preempting the ongoing remaining instance of activity *Unload* (when the unloading *Barge* is the only one left in the harbor) allows this *Barge* to create a new instance of activity *Unload* that will use both *Cranes*, i.e., the preempted Crane (that will become available immediately) and the Crane that just finished unloading the departing *Barge*.

Whenever an instance of activity *Unload* is preempted, it is also necessary to adjust the time needed for a single *Crane* to finish unloading the *Barge*. This remaining time (which is stored in the SaveProp *RemUnloadDur*) is reduced by the elapsed duration of the preempted instance of activity *Unload*, times the number of *Cranes* (1 or 2) that were unloading the *Barge*.

```
BEFOREPREEMPTED     Unload    ASSIGN    Unload.Barge.RemUnloadDur
   'Unload.Barge.RemUnloadDur - (SimTime - Unload.StartTm) * Unload.Crane.Count';
```

Similarly, whenever an instance of activity *Unload* is preempted, the total number of times that the *Barge* has been preempted (which is stored in the SaveProp *TimesPreempted*) is increased by 1.

```
BEFOREPREEMPTED     Unload    ASSIGN    Unload.Barge.TimesPreempted
   'Unload.Barge.TimesPreempted + 1';
```

The sequence of the number of *Cranes* (1 or 2) that worked alone or together to unload a *Barge* thus far is stored in the SaveProp *CraneSeq* as a number that consists only of the digits 1 and 2. For example, the *CraneSeq* final value 1212 indicates that the *Barge* started unloading with one *Crane*, continued with two *Cranes*, switched back to one *Crane*, and finished unloading with two *Cranes*. The number stored in *CraneSeq* is updated whenever an instance of activity *Unload* is preempted and when it ends normally.

```
BEFOREPREEMPTED     Unload    ASSIGN    Unload.Barge.CraneSeq
   'Unload.Barge.CraneSeq*10 + Unload.Crane.Count';
BEFOREEND           Unload    ASSIGN    Unload.Barge.CraneSeq
   'Unload.Barge.CraneSeq*10 + Unload.Crane.Count;
```

When an instance of activity *Unload* is preempted, it creates a new instance of activity *RestartUnload* whose duration is reset to zero by the following statement that calls the function SetTmLeftInst. This allows the preempted *Cranes* (1 or 2) and the partially unloaded *Barge* to be released back to their queues *CraneQ* and *BargeQ* immediately so that new instances of activity *Unload* can start again.

```
AFTERPREEMPTING RestartUnload
     CALL 'SetTmLeftInst[RestartUnload, RestartUnload.TotInst-1, 0]';
```

4.2 STROBOSCOPE Simulation Results

Table 1 shows simulation results for the first nine *Barges* that unloaded at the harbor. The first column shows that *Barges* do not finish unloading in sequential order and the column "*Unload Last Inst*" shows that the last *Unload* instances for successive *Barges* also skip numbers due to multiple preemptions.

Table 1: Partial results from a simulation run (all times shown in days).

		W2Dock	Unload				Times	Crane
Barge#	Arrived	Delay	Started	Duration	Ended	Last Inst.	Preempted	Sequence
1	0.088	0.000	0.088	0.398	0.486	2	1	[21]
3	0.551	0.000	0.551	0.994	1.545	4	0	[1]
2	0.385	0.000	0.385	1.275	1.660	6	3	[1212]
4	4.922	0.000	4.922	0.491	5.413	9	1	[21]
5	5.319	0.000	5.319	0.930	6.249	8	0	[1]
6	5.358	0.055	5.413	1.032	6.445	10	0	[1]
8	6.332	0.113	6.445	0.711	7.156	12	0	[1]
7	5.583	0.666	6.249	1.260	7.509	15	2	[121]
9	7.234	0.000	7.234	0.601	7.835	16	1	[12]

The histogram in Figure 4 shows the percentage of barges that were unloaded by each possible sequence of cranes working alone or together. The first two columns show that the unloading of 56% of the barges was not preempted, with 37% of the barges using two cranes from start to finish while 19% used one crane. The rest of the columns show that the remaining 44% of the barges were preempted up to three times. For example, about 18% started unloading with one crane, were preempted, and finished unloading with two cranes (1→2). About 4% of barges started unloading with one crane, were preempted and continued with two cranes, and were preempted again and finished with one crane (1→2→1). And about 1% of the barges were preempted three times and followed the sequence (2→1→2→1) or the sequence (1→2→1→2).

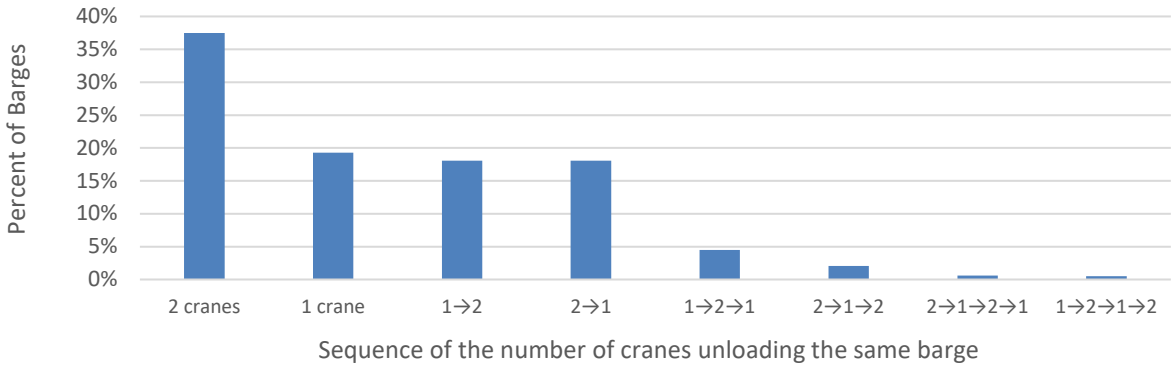

Figure 4: Percent of barges for each possible sequence of the number of cranes unloading the same barge.

As this example shows, modeling all possible sequences of cranes that may unload a barge while keeping track of its remaining unload time can be quite challenging. The new STROBOSCOPE functions and statements that support preemption directly make it much easier to model these possibilities correctly.

5 ALTERNATIVE MODELS NOT USING THE NEW FUNCTION PREEMPT

To compare and verify the results, the two examples described in this paper have also been modeled without using the new STROBOSCOPE preemption functions and statements described above. For both examples, the final simulation models with and without the new preemption facilities ran equally fast and the simulation results were identical.

However, the development of *correct* simulation models without the new STROBOSCOPE preemption facilities was significantly more difficult. The availability of the correct simulation results (obtained from the models that use the new preemption functions and statements) was particularly helpful to identify the existence of modeling errors that would not have been evident without the correct output. Even when knowing that errors existed, however, finding the causes of these errors and the required corrections was not easy and required significant effort. This was especially true for the second model where the prediction of the future time when a barge would finish unloading so that it may preempt the unloading of another barge (and not be preempted itself in the meantime) proved to be quite challenging to forecast ahead of time. To this end, the correct prediction of when a barge would finish unloading required the modification of the simulation network so that arriving barges first enter the new queue *WaitToDock* (where they wait until a *Berth* becomes free) and then move on to the queue *BargeQ* that now only holds at most two *Barges*, both of which are currently unloading. Moreover, the discipline of the queue *BargeQ* is now changed so that the *Barge* with the minimum *RemUnloadDur* is placed at the front of the queue. This way, when two instances of *Unload* start at the same time (which occurs quite often because of preemption), the first *Unload* instance draws the *Barge* which is predicted to finish unloading first and which may preempt the second instance of *Unload* that is started next at the same simulation time. This is but one example of the

required changes. Several other modifications were also necessary before a correct model that does not use the new STROBOSCOPE preemption facilities could be produced.

The difficulties involved in producing correct alternative models for the above examples made it clear that the new STROBOSCOPE preemption functions and statements are a useful tool for the *easy* and *reliable* development of *correct* simulation models that involve the preemption of activities. This is true for simulation modelers at any level of experience but especially so for novice users (such as students) who are not yet simulation experts.

6 CONCLUSION

Construction simulation models often need to interrupt activities when triggering events occur, such as when differing soil conditions are encountered, when inclement weather occurs, or when equipment arrives or breaks down (Ioannou and Kamat 2005; Ioannou and Martinez 1996). To support these needs directly and to model construction operations correctly, new preemption functions and statements have been added to the STROBOSCOPE simulation system that are described and illustrated through their application to two examples. In the first example, laborers move soil using wheelbarrows, and in the second example, cranes unload barges carrying fill material for undersea land reclamation. These case-studies provide insights as to how the prudent use of preemption can improve the design of construction operations and increase production or reduce waiting and service times. Moreover, the models for these examples serve as templates for the proper use of the new preemption capabilities added to STROBOSCOPE to tackle new situations directly and to model preemption in general. The complete STROBOSCOPE models for the two examples presented are available from the first author.

REFERENCES

Ioannou, P. G., and V. R. Kamat. 2005. "Intelligent Preemption in Construction of a Manmade Island for an Airport". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 1515–1523. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ioannou, P. G., and J. C. Martinez. 1996. "Comparison of Construction Alternatives Using Matched Simulation Experiments". *Journal of Construction Engineering and Management*, 122 (3): 231–241.

Law, A. M. 2014. *Simulation Modeling and Analysis*. 5th ed. New York: McGraw-Hill, Inc.

Martinez, J. C. 1996. *STROBOSCOPE State and Resource Based Simulation of Construction Processes*. Ph.D. thesis, Department of Civil Engineering, University of Michigan, Ann Arbor, Michigan.

Martinez, J. C., and P. G. Ioannou. 2023. "STROBOSCOPE Simulation System Software". University of Michigan. Ann Arbor, Michigan. Retrieved from www.ioannou.org on March 15, 2023.

Rekapalli, P. V. 2008. *Discrete-Event Simulation Based Virtual Reality Environments for Construction Operations*. Ph.D. Thesis, Dept. of Civil Engineering. West Lafayette, Indiana: Purdue University.

Zapata, J. C., P. Suresh, and G. V. Reklaitis. 2008. "Assessment of Discrete Event Simulation Software for Enterprise-wide Stochastic Decision Problems". https://www.researchgate.net/publication/241851771_Assessment_of_Discrete_Event_Simulation_Software_for_Enterprise_wide_Stochastic_Decision_Problems.

AUTHOR BIOGRAPHIES

PHOTIOS G. IOANNOU is Professor in the Department of Civil and Environmental Engineering of the University of Michigan and a Fellow of the ASCE. Together with his former doctoral student J.C. Martinez are the designers and developers of the STROBOSCOPE Simulation System. He has also performed research in the development of other simulation systems, including UM-Cyclone, COOPS, EZStrobe, ProbSched, CPMAddon, and ChaStrobe. His research is in construction engineering and management in the areas of simulation, tunneling, competitive bidding models, project finance, innovative project delivery systems, and project scheduling. His email address is photios@umich.edu and his homepage is https://www.ioannou.org

VEERASAK LIKHITRUANGSILP is Associate Professor in the Department of Civil Engineering of the Faculty of Engineering at Chulalongkorn University in Bangkok, Thailand. His research interests are construction process modeling, optimization, and simulation; Building Information Modeling (BIM); digital twins for construction engineering and management; construction contract and claim management; and circular economy in construction. His email address is veerasak.l@chula.ac.th and his homepage is https://www.research.chula.ac.th/researcher/veerasak-likhitruangsilp/