8

Chapter Eight
GENERATING TRACE STREAMS WITH A MODEL OR PROGRAM

In the process of learning about the create, time, destroy, place on, and move commands in Chapters 1-7, we hand-edited some animation trace files in order to try out the commands. We also relied on predefined layout files.

We will continue to rely on predefined layout files for now, because our focus remains on trace stream commands. However, in this chapter we will learn how to generate trace streams automatically, using simulation models or other programs.

Adding Animation to a Model

Proof Interface Ease-of-Use Issues

If you haven't already chosen a simulation tool, there are three things you should consider in evaluating the tool's ability to interface with Proof.

First, a wide range of approaches has been taken for generating Proof trace stream commands. On the "high end," requiring the least work on your part, are Proof interfaces that have been *integrated* into simulation software their vendors. Wolverine's SLX™ and Imagine That's Extend™ products are examples of integrated Proof interfaces. On the "low end," requiring the most work on your part, are "do it yourself" approaches based on the use of low-level software features such as "write to file."

Second, simulation tools differ widely in the extent to which you are allowed to "get inside" model building blocks. For example, a very high-level tool for simulation of vehicular transport systems tool might include a feature that makes all collision-avoidance decisions. If you're unable to "get inside" this feature, you may not be able to generate trace stream commands at

the required key points in the logic of a model. Beware of black boxes.

Third, simulation tools that offer extensibility have a decided advantage over those that do not. If you use a simulation tool that has extensibility, you can add reusable shortcuts (verbs?) for using Proof to the tool on your own, if the tool's vendor has not already done so.

The Instrumentation Process

Once you've selected simulation software, the most important step in adding animation to a simulation model is to choose which model features are most important and how best to exploit Proof's visual medium for portraying them. (These issues were discussed in the "Philosophy of Animation" section of Chapter 2. If you haven't already done so, you should read that discussion.)

Once you've decided what you want to show on the screen, you need to carefully *instrument* your model to generate Proof trace stream commands at appropriate points in the model. For example, if you want to use color to indicate the state of a server, at each point where the state of the server changes, you'll need to generate a set color command. This process is usually "surgical" in nature, that is, animation can be added to a large model by making a surprisingly small number of insertions. In this chapter, we will illustrate the addition of animation to a very simple model. Because the model is so simple, the ratio of animation-generation logic to model logic is much higher than it would be in a real-world model.

Communicating with Proof

Time Synchronization

Suppose that an event occurs at simulated time 12.34 in your model, and that as a consequence of this event your model needs to generate several trace stream commands. For proper animation, the Proof commands must occur at animated time 12.34. In other words, animation time and simulation time must be synchronized. To do this, your model must generate time commands that tell Proof the time at which other commands in the trace stream are to be executed. If you haven't already done so, you should read *How Proof Measures Time* (page 3-8), and *Simulation Time, Animation Time, and Real Time* (page 5-3).

The SLX and Extend interfaces to Proof generate time commands automatically, so if you're using either of these products to drive Proof, you won't have to worry about generating time commands. If you're using other simulation software, you'll have to generate your own time commands. The simplest approach is to generate a time command prior to any other command; however, whenever you generate multiple commands at the same instant of simulated time, redundant time commands will add unnecessary overhead.

Redundant time *commands are the number one cause of unnecessarily large trace files.*

The preferred approach is to test whether a time command has already been generated for the current instant in simulated time whenever you're going to generate a trace stream command.

Object IDs

In order to manipulate Proof Objects in trace stream commands, you must specify an Object's name or number. Frequently, Proof Objects have a 1-to-1 relationship with objects moving through your model. Depending on what software you're using, model objects may be called transactions, items, pucks, or something else. In some software, moving model objects are automatically assigned ID numbers. For example, GPSS/H transactions are assigned ID numbers accessible as XID1. If you're using software that provides such ID numbers, and there's a 1-to-1 relationship between model objects and Proof Objects, the easiest approach is to use your model's object IDs as Proof Object IDs. If not, you'll have to create unique ID numbers (or symbolic names) for your Proof Objects, record them in your model objects, and use them consistently.

Other Named Entities

Trace commands that refer to Proof Object Classes, Paths, Plots, Bars, Messages, and Named Views require specification of the names of these entities. Your model must use the case-sensitive names for these entities defined in your animation's layout file.

Software that Works with Proof

Here is a list of selected software that works with Proof Animation:

SLX™ (Wolverine)

GPSS/H™ (Wolverine)

Extend™ (Imagine That, Inc.)

Arena™, AutoMod™, ProModel™, Simscript™, SLAM II™, Taylor ED™, WITNESS™

Procedural programming languages such as BASIC, C/C++, FORTRAN, or Java

Would you ever want to use Proof with software that has built-in animation already (i.e. Arena, AutoMod, Extend, ProModel, SIMUL8, Taylor ED, WITNESS)? Perhaps not, but in some cases Proof offers better, more detailed animation. Proof also can animate a CAD layout better than software that does not support vector-based graphics. In other cases Proof has unique features such as the ability to distribute an animation royalty-free to others such that they can

speed up or slow down, zoom in or out, and so forth – features that are lacking when animations are distributed as movie files.

The Carwash Example

System Description

To illustrate generation of trace stream commands, we will use a simple model of an automated carwash. The carwash operates as follows:

- Cars arrive at the carwash with exponentially distributed interarrival times having a mean of 2.2 minutes.
- The cars wait in a single queue that takes 15 seconds to traverse if it is empty.
- The washing operation takes place as the car moves along a slow, fixed-speed chain drive. This takes a constant 2 minutes.
- When the car being washed has reached the end of the chain drive that leads through the washer, the next car in the queue, if any, gets onto the chain. Also at this time, the car just washed sits for 1.5 minutes while it is dried by hand.
- Finally, the car is driven away, remaining in the system for 15 seconds before disappearing.

Proof Features Used to Represent the Carwash

We'll use the following Proof features to represent the carwash:

- Cars flowing through the carwash will be represented by an Object Class named "Car".
- The arrival queue will be represented by an accumulating Path named "ENTRY".
- The chain drive through the wash area will be represented by a non-accumulating Path named CHAIN.
- The drive-away exit area will be represented by a non-accumulating Path named "EXIT".

Let's re-describe a car's activities, using Proof terminology. A car:

- Arrives in system and gets on the ENTRY Path
- Travels up to 15 seconds to get behind previous car in line
- Waits (if necessary) to be first in line
- Waits (if necessary) for previous car to reach end of wash area

- Gets on the CHAIN path and travels 2 minutes to end of wash area
- Gets dried (1.5 minutes, not moving)
- Gets on the EXIT path and travels 15 seconds
- Leaves the system

The Carwash Layout File

The Layout file carwash.lay, found in the exercise folder, contains definitions for the three Paths, ENTRY, CHAIN, and EXIT, as well as the "Car" Object Class called Car. Remember that these names are case sensitive (use "ENTRY" and "Car", not "Entry" or "CAR"). The appearance of the layout file is shown in Figure 8-1.

How Did We Build the Layout?

A complete, blow-by-blow description of how we built the layout would be too lengthy for inclusion at this point. Instead, we'll present a brief outline of the layout-building process. An overview of Proof's Draw Mode is presented in Chapter 9; drawing Lines and Text is described in Chapter 10; defining Object Classes is described in Chapter 11; and defining Paths is described in Chapter 12.

Drawing Layout Geometry

The geometry of the layout was drawn as follows:

- We drew three horizontal lines, end-to-end. The leftmost line is visible in the waiting line area, and the rightmost line is visible in the drive away area. The center line was drawn in the Backdrop color, so although it's visible in Figure 8-1, when an animation runs, this line is invisible.
- We drew two shorter horizontal lines and two short, vertical lines to enclose the washing area.
- We added a title and descriptive text.

Defining the "Car" Object Class

- We drew a simple, small rectangle to represent cars flowing through the system, and we filled the rectangle with a solid color. (Fills are described on page 10-12)

Defining the Paths

- We superimposed the ENTRY, CHAIN, and EXIT Paths over the leftmost, center, and rightmost end-to-end lines described above. The ENTRY Path was defined as an accumulating Path with a travel time of 0.25; the CHAIN Path was defined as a non-accumulating Path with a travel time of 2.0; and the EXIT Path was defined as a non-accumulating Path with a travel time of 0.25. A time unit of minutes was assumed.

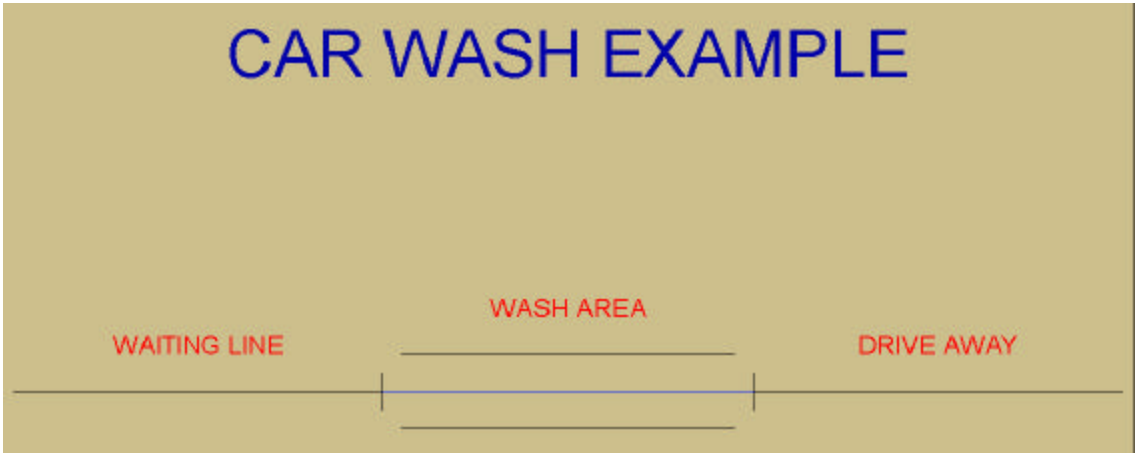

Figure 8-1: Carwash.lay

Exercise 8-1: CARWASH

Using simulation software of your choice, create and test a model of the carwash system described above. Use a base model name of carwash1. Then, add the necessary constructs to your model to cause the carwash.atf trace file to be generated each time the model is run. Call the revised model carwash2.

For SLX, Extend, Arena, and GPSS/H, completed carwash2 examples are presented later in this chapter. If you are using one of these tools, you can check your work against these files. Files containing the carwash2 models in each simulation tool are in the sample folder.

You will also find carwash2.c (for the C programming language) in the sample folder. This file is not reproduced in this book.

Variations on the Carwash Example

In this section, we offer food for thought as to possible enhancements of the carwash2 model. Solutions to these variations are *not* included in the sample folder or in this book.

Adding Color

To add color to the model, assign each car a random color number between F2 and F5. (See the description of the set color command on page 6-8.) Dirty cars should be shown as TAN or F6, and wet soapy cars as gray (L1) (a car gets wet as soon as it gets on the chain drive). A car being dried is GREEN (F7). When a car is finished being dried, its true color can be revealed.

Blockable Resources

For a more complicated example, assume that the lead car in the queue must be prepped before washing. The prep operation takes 30 seconds.

This condition is not as simple as it may first appear to be. A car doesn't simply get prepped "after undergoing the 15 second drive-in time." It has to wait until it is the first in line (according to the condition given in the preceding paragraph).

This implies that the front space in the line must be modeled as a constrained resource. Furthermore, it is a constrained resource that can be blocked by downstream competition for the chain. You can animate this variation, using another color change (e.g. to green) to depict the prep operation.

Upon completion of this variation, you should realize that anything you model can be animated. You are not required to use a particular modeling construct in order to access the animation feature you want.

However, you should also realize that detailed animation is impossible without detailed simulation! You cannot accurately show the timing of the prep operation without precisely modeling it.

At this point in your modeling effort, you have probably added more detail than you originally expected. There is plenty of capacity at the prep station. If you didn't want to animate the prep operation, you would probably omit it from this model. But what if the prep operation is consuming precious resources? Read on.

More Complicated Model Logic

For another twist, consider the staffing requirements for this carwash. We've assumed up to now that there is a dedicated dryer worker and a dedicated prepper. Can one worker handle all drying and prepping? Assume that it takes 20 seconds for the worker to run from the prep area to the dry area or back. You'll need to add a worker resource and a worker location indicator to your model. We've included paths called TODRY and TOPREP if you want to try adding the worker to your animation.

What does the main input queue look like now? Can the system description be altered so that the worker doesn't have to run back and forth constantly?

More observations about this model

In your original model, you may have noticed something unusual when two or more cars were queued up on the ENTRY Path. When the lead car moves slowly onto the CHAIN Path, the second car moves quickly to assume the lead position on ENTRY — "eating into" the former lead car briefly on the screen. This is rather unusual in Proof Animation, but it will occur when fast moving Objects slow down with Objects queued behind them.

How would the real system work? No car is going to move forward until the car ahead has cleared enough space. You can handle this in several ways that involve Proof Animation, your model, or both. These topics are covered in later chapters.

SLX Example

SLX is a highly extensible, language-based simulation platform. Wolverine has developed an interface that makes it extremely easy to generate Proof trace streams in an SLX program. This interface was developed in three stages. First, a prototype interface was developed for generating trace files, using only extensibility features that are readily available to all SLX users. In the process of doing so, we developed SLX equivalents for every Proof trace stream command. Second, we made a number of changes to both SLX and Proof to tighten up the interface, using our internal development tools. Finally, the trace *file* interface was extended to add the capability for concurrent animation.

SLX's extensibility mechanisms include the ability to introduce new statements into the SLX language. Our animated SLX model makes extensive use of SLX statement definitions corresponding to Proof trace stream commands. The definitions of these statements are contained in the file proof4.slx, imported into our SLX carwash model. Although this file was developed by Wolverine, there was nothing "magic" or proprietary about the development process. SLX's extensibility mechanisms were designed for use by ordinary end-users; they are not the personal fiefdom of Wolverine's programming staff.

Each of the statements added to SLX for generating Proof trace stream commands begins with "PA_" (for Proof Animation). For example, the PA_Create statement generates a Proof Create command. The arguments to such statements can be constants or expressions. For example, in the statement

```
PA_Create "Car" ID;
```

"Car" is a constant character string, and ID is an integer variable. The definition of the PA_Create statement specifies how these values are edited into a command line that is written to the active trace file.

An SLX implementation of the carwash model, without animation is shown in Figure 8-2.

An SLX implementation of the carwash model, including animation, is shown in Figure 8-3. The model shown generates a trace file.

```
//*****************************************************************************
//          SLX Carwash Example (Without Animation)
//*****************************************************************************

import <h7>                         // use GPSS/H-style queueing constructs

module CarWash
    {
    static int          NCARS = 100;    // simulate 100 cars
    int                 ncars;          // number of cars created
    control boolean     done;           // set TRUE after NCARS cars
    rn_stream           IAT;            // random number stream
    facility            wash;           // single server

    class car
        {
        int             ID = ++ncars;   // assign car ID number
        actions
            {
            advance     0.25;       // minimum time through the entry queue
            seize       wash;       // get exclusive control of the wash
            advance     2.0;        // wash time
            release wash;           // allow a successor car to be washed
            advance     1.5;        // drying time
            advance     0.25;       // exit time

            if (ID == NCARS)        // last car?
                done = TRUE;        // signal the main program
            terminate;              // exit the system
            }
        };

    procedure main()
        {
        arrivals: car  iat = rv_expo(IAT, 2.2)  count = NCARS;

        wait until (done);
        report system;              // issue summary report
        }
    }
```

Figure 8-2. SLX Carwash Model (Without Animation)

```
//*****************************************************************************
//    SLX Carwash Example (With Animation)
//*****************************************************************************

import <h7>                         // use GPSS/H-style queueing constructs
import <proof4>                     // SLX-Proof interface

module CarWash
    {
    static int          NCARS = 100;        // simulate 100 cars

    int                 ncars;              // number of cars created
    control boolean     done;               // set TRUE after NCARS cars
    rn_stream           IAT;                // random number stream
    facility            wash;               // single server

    class car
        {
        int         ID = ++ncars;           // assign car ID number

        actions
            {
            PA_Create "Car" ID;             // create Proof object
            PA_Place  ID on "ENTRY";        // place on entry path

            advance   0.25;         // minimum time through the entry queue
            seize     wash;         // get exclusive control of the wash

            PA_Place  ID on "CHAIN";        // place on the chain

            advance   2.0;          // wash time
            release wash;           // allow a successor car to be washed

            PA_Place  ID on "EXIT";         // place on the exit path

            advance   1.5;          // drying time
            advance   0.25;         // exit time

            PA_Destroy ID;                  // destroy Proof object
```

Figure 8-3 SLX Carwash Model (With Animation)

```
            if (ID == NCARS)          // last car?
                done = TRUE;          // signal the main program

            terminate;                // exit the system
            }
        };

    procedure main()
        {
        int       ncars;

        PA_ATF     "carwash2.atf";  // animation trace file (atf)

        arrivals: car  iat = rv_expo(IAT, 2.2)  count = NCARS;

        wait until (done);

        PA_End;                   // terminate the trace file
        report system;            // issue summary report
        }
}
```

Figure 8-3 (Continued). SLX Carwash Model (With Animation)

Converting the SLX trace file-generating model to a concurrent animation can be accomplished by replacing the model's PA_ATF statement with a CPA_Layout statement. (The CPA_ prefix stands for Concurrent Proof Animation.) Note that layout file and trace file specifications must be mutually exclusive. A trace file-generating model must contain a PA_ATF statement, and it cannot include a CPA_Layout statement. Conversely, a model that drives a concurrent animation must contain a CPA_Layout statement (in order to tell the Proof library what layout is to be animated), but cannot contain a PA_ATF statement (because no trace file is generated).

Extend Example

Extend has an integrated interface to Proof Animation, and many Extend blocks include implicit Proof commands that can be accessed from the block's dialog. Thus the Proof Control block is the only block that was added explicitly to animate this Extend model. The remaining animation commands were implemented by making selections from popup lists in the dialogs of the blocks already in the model. Additionally, Extend automatically issues the time animation command to Proof whenever the simulation clock advances, so you do not need to issue those commands separately.

The following example assumes the appropriate blocks have been placed and connected on the model worksheet and model parameters have been entered. It also assumes that a Proof layout containing all the necessary elements—paths and car objects—has already been created, but Proof animation commands have not yet been entered in block dialogs.

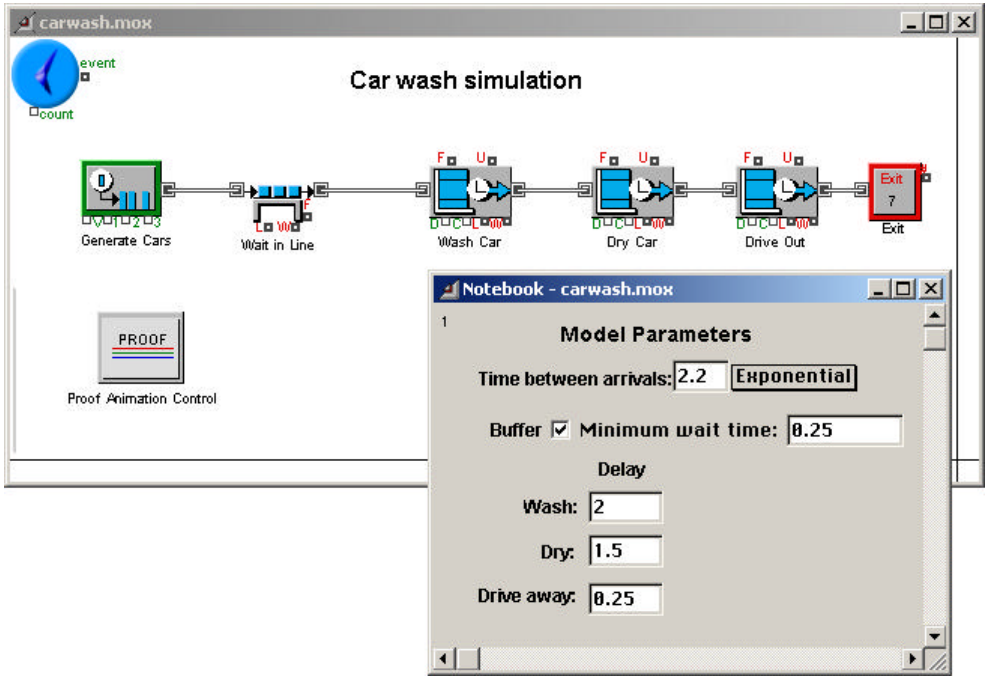

Figure 8-4. Carwash system representation using Extend software

The carwash model shown in Figure 8-4 consists of the following 8 blocks:

- Executive: controls the timing of discrete event models
- Proof Animation Control: the interface between Extend and Proof
- Generate Cars: creates items (cars) with an exponential interarrival time of 2.2 minutes
- Wait in Line: holds the cars in a FIFO queue for a minimum .25 minutes (15 seconds)
- Wash Car: delays the cars for a fixed 2 minutes of wash time
- Dry Car: delays the cars for a fixed 1.5 minutes of drying time
- Drive Out: delays the cars for the .25 minutes (15 seconds) it takes to drive away
- Exit: removes the cars from the system and counts them as they leave

For visual clarity, input parameters (such as the distribution of arrivals and the wash time) have been "cloned" from the dialogs of the blocks to the model's Notebook.

Double-clicking a block opens a dialog box for entering or changing model parameters. For example, double-clicking the Proof Animation Control block to opens its dialog, shown in Figure 8-5. The Controls tab of this block allows you to specify the Proof layout file and whether the animation is generated concurrently or as a post-process trace file. As you will see later in this section, the Animate block tab displays the appropriate commands, objects and paths for the other blocks in the model. In this example, you only need to check "Concurrent animation" in the Controls tab, then click the **Layout file name:** button and locate the carwash layout file:

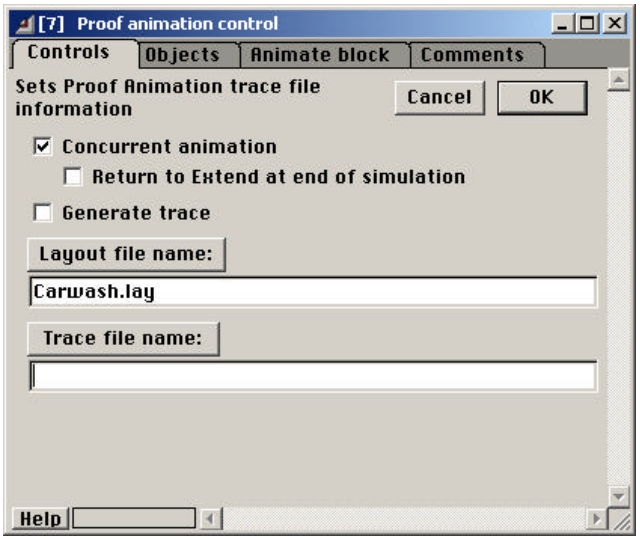

Figure 8-5. Proof Animation Control dialog

Note that Extend is almost always used with Proof to create concurrent animations; however, by specifying a trace file name, you can use Extend to create a trace file to be used for post-processed animation.

The steps for enabling Proof and specifying which implicit Proof commands to use is straightforward:

- Double-click an Extend block to open its dialog
- Go to the block's Animate tab
- In the Animate tab, check the "Animate with…" checkbox.
- Then click the "Proof animation" button
- In the dialog that appears, select the appropriate options for Proof

Every time you do this you are sending a message to the Proof Animation Control block, causing its Animate Block dialog to open. That dialog presents the Proof commands, objects, and paths that are appropriate for the Extend block you're animating. These steps are repeated for each block that will effect the animation of the system.

For example, open the Generate Cars block of Figure 8-4 to get its dialog box, shown in Figure 8-6. In the Items tab, you can see that the user has chosen the exponential distribution and entered 2.2 as the mean inter-arrival time. By default, the block causes one car at a time to arrive until the simulation time ends, with the first car arriving at time 0.0.

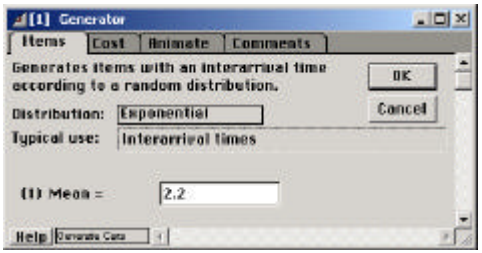

Figure 8-6. Items tab of Generate Cars dialog box

Click on the block's Animate tab, shown in Figure 8-7. Note that this tab allows you to customize the animation for both the Extend model and for the Proof animation of the model. To enable Proof for this block, select the "Animate with…" check box as below:

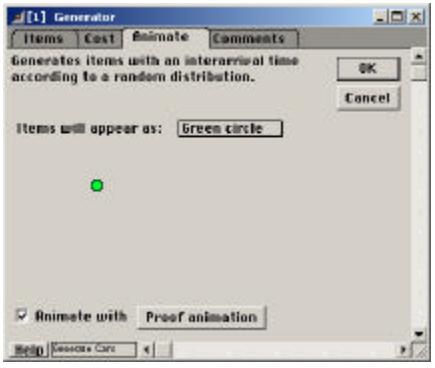

Figure 8-7. Animate tab of Generate Cars dialog box

Clicking the "**Proof** animation" button in this dialog causes the "Animate block" tab of the Proof Animation Control to appear, shown in Figure 8-8. This is where you select the appropriate Proof commands, objects, and paths.

Note that the block being animated is identified by name (Generator) and label (Generate Cars). This dialog doesn't present any options for Proof commands, because **create** is the only command that the Generator block can send to Proof. From the available list of Proof objects, select the "Car". Then every time it creates an item entity, the Generator will instruct Proof to create a Car object. Note: Extend prevents you from referring to non-existent Object Classes, since it allows you to select only Class names it has found in your layout file.

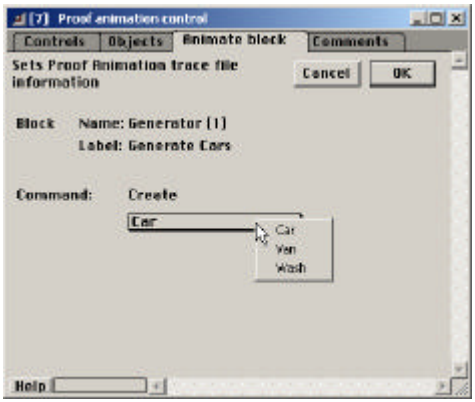

Figure 8-8. Animate (Generator) block tab of the Proof Animation Control block

The next block shown in Figure 8-4 is a Buffer, labeled "Wait in Line", that queues the cars before the chain. When the car arrives at the Buffer block, you want to show it moving on the Proof ENTRY path. In the Animate tab of the Buffer, select the "Animate with …" checkbox, then click the **Proof animation** button, as you did for the Generator block. As before, the Animate block dialog appears (Figure 8-9), but in this instance it allows you to select the command and path for the Buffer (Wait in Line) block. In the dialog, select the Place on Proof command and the ENTRY path:

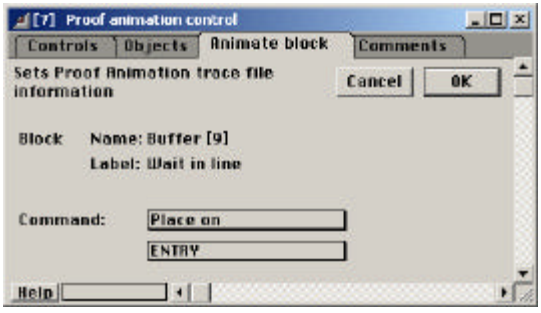

Figure 8-9. Animate (Buffer) block tab of the Proof Animation Control block

This will issue a Place *item* on ENTRY command to Proof whenever an item travels through this Buffer. (*item* is the unique index for each item in Extend; it is used by Proof as the object ID to identify each animation object.)

To summarize, you enable Proof in the Animate tab of a block, then select the appropriate commands from the Animate block dialog that appears. Repeat this process for each block that will affect the animation, as shown in the following table:

Block Label	Block Name	Command	Object	Path
Generate Cars	Generator	create	Car	-
Wait in Line	Buffer	place on	-	ENTRY
Wash Car	Activity Delay	place on	-	CHAIN
Dry Car	Activity Delay	-	-	-
Drive Out	Activity Delay	place on	-	EXIT
Exit	Exit	destroy	-	-

When you run the simulation, you will see Proof animate concurrently with the simulation. You can also pause the Proof animation, switch back to Extend, change model parameters, resume the simulation run, and see the effect of your changes in the animation.

Although this model used only implicit commands, additional Proof commands can be accessed explicitly by using blocks in the Proof library that ships with the Extend Suite package from Imagine That, Inc.

ARENA Example

This section shows a version of the carwash example using the Arena Basic Edition software. Figure 8-10 shows the main Arena blocks that represent the carwash system, as well as the additional explicit "Proof" blocks that generate a trace (.ATF) file that will drive the Proof Animation. This example assumes that a Proof layout containing all the necessary elements—paths and car objects—has already been created.

The carwash model consists of the following blocks: *Create* (cars enter the system at the specified exponentially distributed interarrival time with a mean of 2.2 minutes), *Car Queue* (time delay of 15 seconds and a car queue while waiting for the washing area to be available; cars seize the washing area before exiting this block), *Washing* (2-minute delay for washing operation; cars release the washing area before exiting the block), *Drying* (1.5-minute delay for drying operation), *Car Exit* (15-second delay for car removal), and *Dispose* (Arena entities are destroyed).

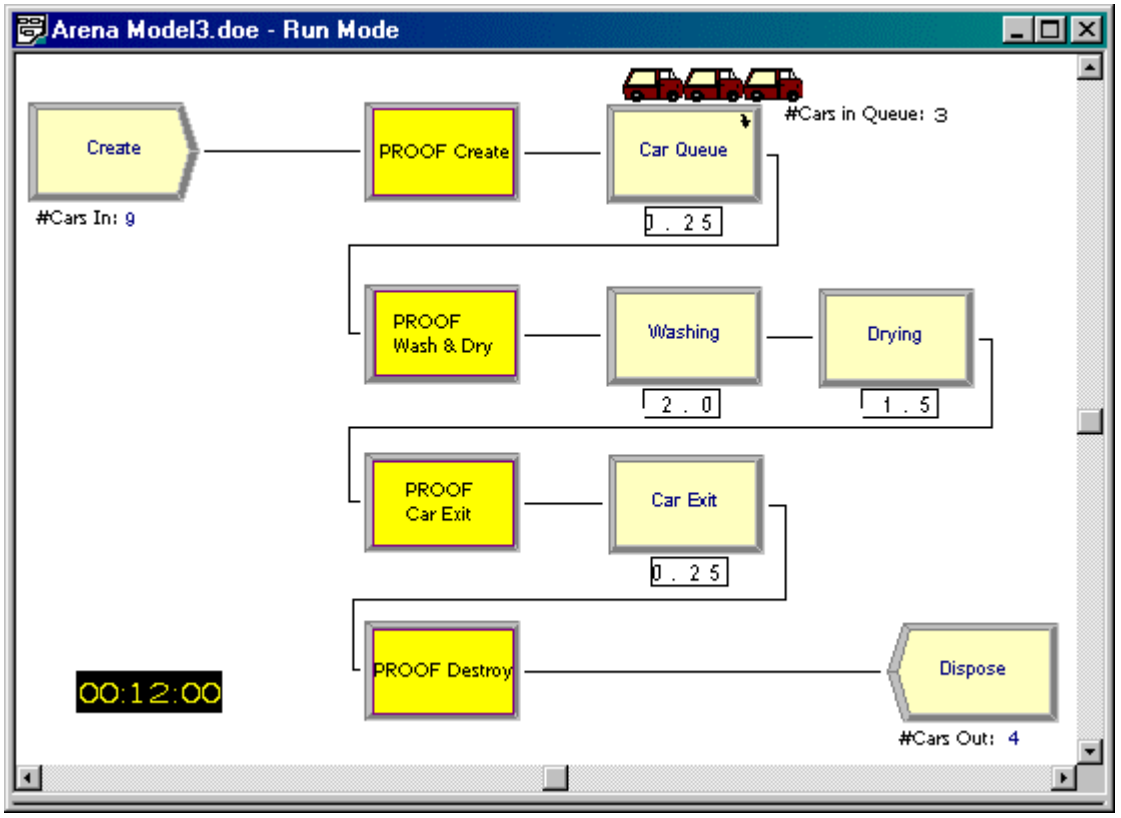

Figure 8-10. Carwash system representation using Arena Basic Edition

Model parameters can be entered or edited by double-clicking on the corresponding Arena blocks. The dialog box of the *Create* block is shown in Figure 8-11 below. Using this dialog box, the user has specified that entities of the type "car" will arrive following an exponentially distributed interarrival time with a mean of 2.2 minutes, that one car will arrive at a time, and that the first car will arrive at time 0.0.

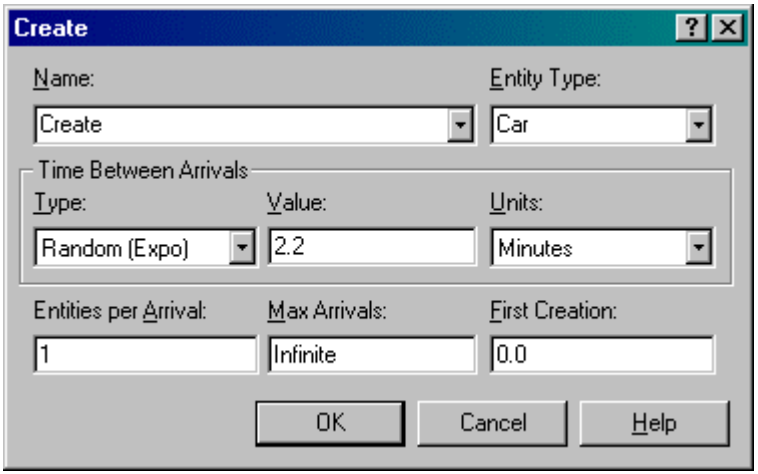

Figure 8-11. Dialog box of the Arena block Create

In this example, "Proof" blocks have been implemented using Arena submodels, which comprise one or more Arena blocks. Submodels can be edited by right-clicking on a submodel block and choosing the **Edit** option. Figure 8-12 shows the blocks that comprise the submodel *PROOF Create*, which creates cars and places them on ENTRY path. Figure 8-10 shows the main Arena blocks that represent the carwash system and a number of inserted "Proof" blocks that write the appropriate trace file commands that will drive the Proof Animation. The animation will show basic car movement through the system, i.e., cars show up at the beginning of ENTRY path, move along ENTRY path and then proceed through CHAIN and EXIT paths, and finally, cars disappear at the end of EXIT path.

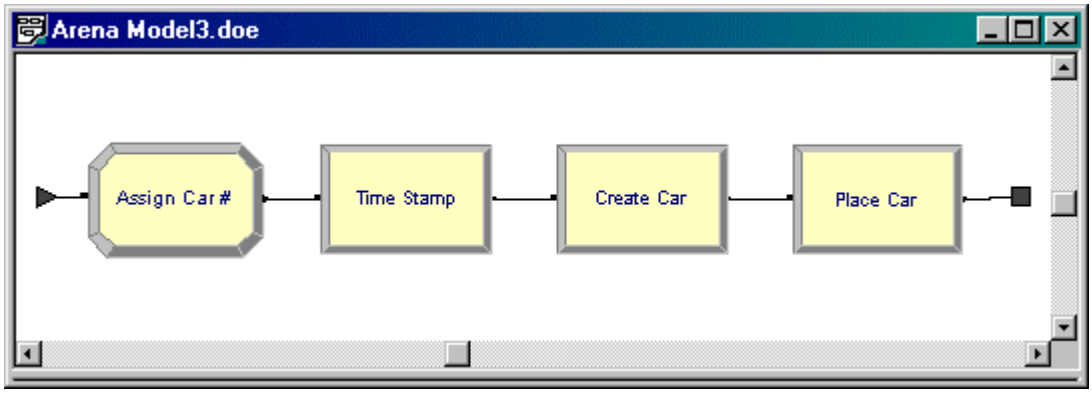

Figure 8-12. Arena submodel PROOF Create

The first block in Figure 8-12, *Assign Car#*, assigns a unique ID number to each Arena entity to allow its identification at any point within the model. The remaining three blocks are ReadWrite blocks that write trace file command lines. The *Time Stamp* block, shown in Figure 8-13, writes a TIME command to the trace file. The *Create Car* block, shown in Figure 8-14, writes the CREATE command that creates an instance of Proof's object "Car." Finally, the *Place Car* block shown in Figure 8-15, writes the PLACE ON command that places cars on ENTRY path.

In summary, following the carwash implementation described above, each "Proof" block will consist of one ReadWrite block to write a time command line followed by one or more ReadWrite blocks to write command lines that will animate any other aspect of the system being represented.

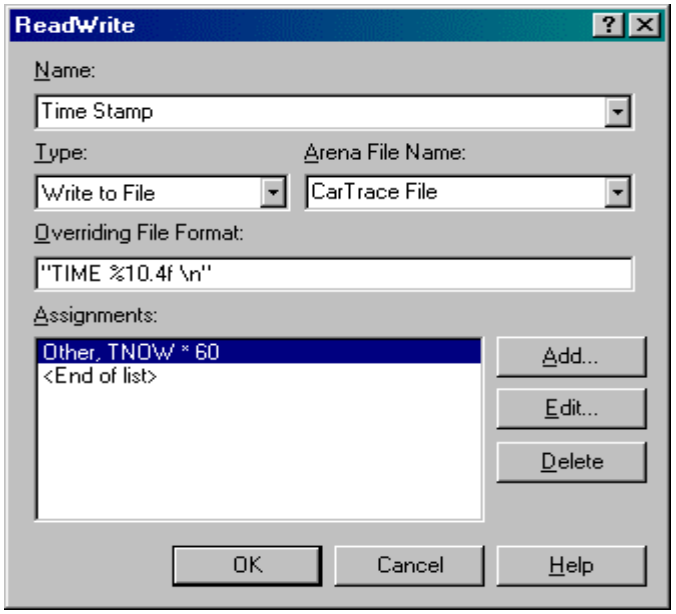

Figure 8-13. Time Stamp *block dialog box*

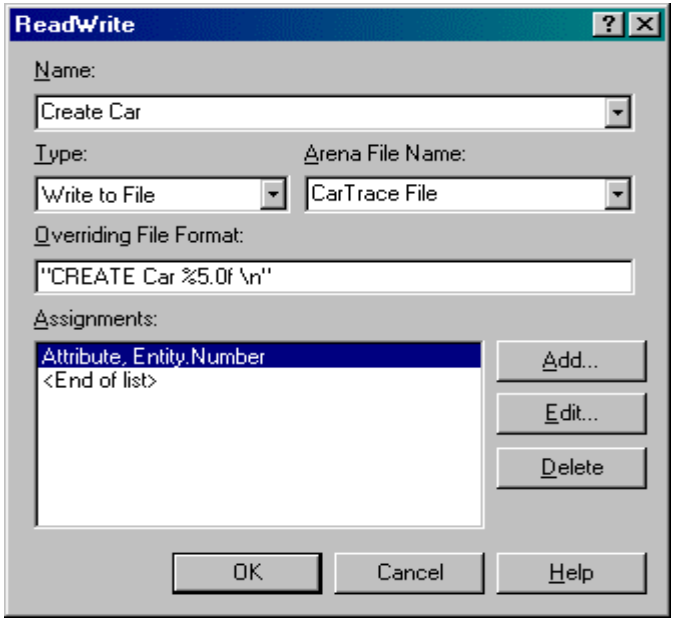

Figure 8-14. Create Car *block dialog box*

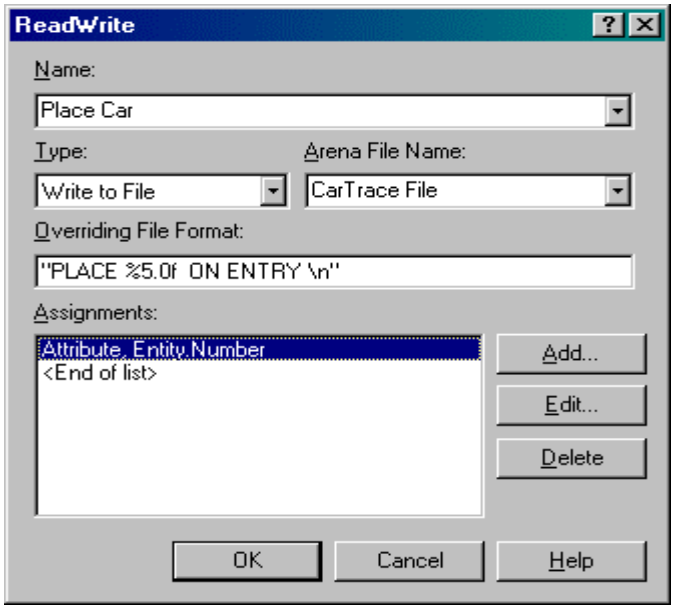

Figure 8-15. Place Car block dialog box

GPSS/H Example

Below is a GPSS/H model of the carwash system, without animation. This model is called carwash1.gps.

```
            SIMULATE
*
* Block Statements
*
            GENERATE    RVEXPO(1,2.2)
            ADVANCE     0.25        Car moves to entry
            SEIZE       CHAIN
            ADVANCE     2           Car moves thru wash
            RELEASE     CHAIN
            ADVANCE     1.5         Car sits for drying
            ADVANCE     0.25        Car moves to exit
            TERMINATE   1
*
* Control Statements
*
            START       100
            END
```

Next we show an "instrumented" version of carwash1.gps. This model is called carwash2.gps.

Notice the importance of including a time command prior to each command sequence. Some of the time commands in the resulting trace file will be redundant (i.e., they will specify identical numbers). Still, it is easier to include them than to code more explicit logic to test whether the time command is really necessary.

```
*
            SIMULATE
*
ATF         FILEDEF     'carwash.atf'
*
* Block Statements
*
            GENERATE    RVEXPO(1,2.2)
            BPUTPIC     FILE=ATF,LINES=3,(AC1,XID1,XID1)
TIME *.**
CREATE Car *
PLACE * ON ENTRY
            ADVANCE     0.25
            SEIZE       CHAIN
            BPUTPIC     FILE=ATF,LINES=2,(AC1,XID1)
TIME *.**
PLACE * ON CHAIN
            ADVANCE     2
            RELEASE     CHAIN
            ADVANCE     1.5
            BPUTPIC     FILE=ATF,LINES=2,(AC1,XID1)
TIME *.**
PLACE * ON EXIT
            ADVANCE     0.25
            BPUTPIC     FILE=ATF,LINES=2,(AC1,XID1)
TIME *.**
DESTROY *
            TERMINATE   1
*
* Control Statements
*
            START       100
            PUTPIC      FILE=ATF
END
            END
```

Using GPSS/H Macros

There is an easier way to drive Proof Animation from GPSS/H than using BPUTPIC statements: the built-in GPSS/H Macro capability. Using Macros, you can create your own higher-level "statements." Wherever you use a Macro "statement," it is expanded at compile time into one or more other statements prior to model execution.

Presented next is a modified version of the same GPSS/H model, with Macro statements to drive the animation. This model is called carwash3.gps.

Notice that we have tucked into the beginning of the CREATE, DESTROY, and PLACEON

Macros a call to another Macro that checks to see whether we really need another time command in the trace file. As a result the time command is inserted only when necessary. It's a win-win situation. Our trace file will be smaller, and our model will be more readable without time calls.

A basic set of Macro definitions for writing the simplest Proof Animation trace file commands from GPSS/H is provided on your diskette in the file proofmac.gps. It includes the four Macro definitions used in carwash2.gps as well as three others (PLACEAT, SETCOLR, and WRITE). (place at and set color were covered in Chapter 6. The write command is covered in Chapter 13.)

If you adopt the Macro approach, you will end up developing your own versions of Macros for various trace file commands. This takes only a few minutes for each Macro you want to use, if you follow the examples in PROOFMAC.GPS. GPSS/H Macros are discussed again in Chapter 13.

As you can see from the listing that follows, with the Macro approach, it's almost as if a set of animation commands were built into the GPSS/H language.

```
            SIMULATE
*
ATF         FILEDEF     'carwash.atf'
*
* ------------------------------------------------------
*
* Proof Animation Macro Definitions Begin Here
*
* ------------------------------------------------------
*
            REAL        &ATFTIME    Declare Ampervariable
                                    used in PUTTIME
*
* In the PUTTIME Macro, remember that "*" used in an
* expression that calculates a Block number, means "the
* current Block."
*
PUTTIME     STARTMACRO
            TEST NE     AC1,&ATFTIME,*+3
            BPUTPIC     FILE=ATF,(AC1)
Time *.*
            BLET        &ATFTIME=AC1
            ENDMACRO
*
* The remaining Macros implement a small set of Trace
* File output commands.
*
CREATE      STARTMACRO
PUTTIME     MACRO
            BPUTPIC     FILE=ATF,(#B)
Create #A *
            ENDMACRO
*
DESTROY     STARTMACRO
PUTTIME     MACRO
            BPUTPIC     FILE=ATF,(#A)
Destroy *
            ENDMACRO
*
PLACEON     STARTMACRO
PUTTIME     MACRO
            BPUTPIC                 FILE=ATF,(#A)
Place * On #B
            ENDMACRO
```

```
*
* ------------------------------------------------------------
*
* CARWASH Model Statements Begin Here
*
* ------------------------------------------------------------
*
* Block Statements
*
            GENERATE    RVEXPO(1,2.2)
CREATE      MACRO       Car,XID1
PLACEON     MACRO       XID1,ENTRY
            ADVANCE     0.25
            SEIZE       CHAIN
PLACEON     MACRO       XID1,CHAIN
            ADVANCE     2
            RELEASE     CHAIN
            ADVANCE     1.5
PLACEON     MACRO       XID1,EXIT
            ADVANCE     0.25
DESTROY     MACRO       XID1
            TERMINATE   1
*
* Control Statements
*
            START       100
            END
```