Dynamic Time Step Control Algorithm Enhancements

James C. (Jim) Bach
Hardware Analysis & Simulation Group
Delphi Delco Electronics Systems, Kokomo, IN, 46904-9005 USA
Tel: (765)-451-0455 E-mail: James.C.Bach@DelphiAuto.com

Abstract

Analog simulators typically use a dynamic, adaptive time step control strategy in order to minimize the calculations performed during inactive portions of a system's operation, while maintaining adequate time resolution during active portions. Without this dynamic control of time steps, one could generate too few time steps to accurately render fast-edged events, or spend a large percentage of CPU time calculating redundant values.

*These **D**ynamic **T**ime **S**tep (DTS) algorithms are often overzealous in their relaxation of time steps, and thus can render waveforms inadequately or inaccurately. Thus, the user can make poor decisions because of the incorrectly presented waveforms. Some simulators provide user-adjustable control parameters for the DTS. The Saber simulator, with four primary DTS control parameters, allows users to "dial in" just about any results they want. Which one is correct is often debatable.*

The techniques presented here allow the user to override the built-in DTS in order to obtain trustworthy results.

Introduction

The Saber simulator, like most analog time-domain simulators, makes use of an algorithm commonly referred to as **D**ynamic **T**ime **S**teps (DTS) to control the time intervals at which the governing equations are solved. The purpose of this algorithm is to reduce the number of calculations performed during the inactive portions of the simulation, while at the same time maintaining the fidelity of the results during the active portions. Without DTS, that is, if one were to use **F**ixed **T**ime **S**teps (FTS), the simulator could produce either:

- *Results that are too coarse to accurately render important, transient events (edges, glitches, etc.). Fast-edged events would have jagged "bumps".*
- *Results that contain many redundant data points generated at the expense of CPU time. The user would wait a long time for the results, possibly filling the hard drive in the process.*

For the most part, Saber's built-in DTS, using the default settings for the four primary DTS control parameters, produces acceptable results, particularly with simple circuits and systems. However, the built-in algorithm often does not provide an adequate number of time steps to accurately render waveforms. The user can adjust these control parameters, and often obtain better-looking results; however, this requires a lot of experimentation by even an experienced user. Often this "calibrating the simulator" (as Saber's creators refer to this task) is non-productive, as adjusting these parameters can cause convergence problems, preventing the simulation from completing. Novice and infrequent users often don't even know that these control parameters exist, much less how to adjust them properly.

The techniques presented here allow the user to override the built-in DTS in order to provide more believable and trustworthy results. These techniques can be "built-in" to device or sub-system models, so that the end users of the models are able to use them without added thought or distress. Adding in some of these alternative DTS algorithms can make the models a "no brainer" to use. Although these techniques were developed to overcome perceived deficiencies with the Saber simulator, it is hoped that they can be beneficial with other analog HDL-based simulators.

Example of the Problem

Given the simplistic (but real-world) circuit below:

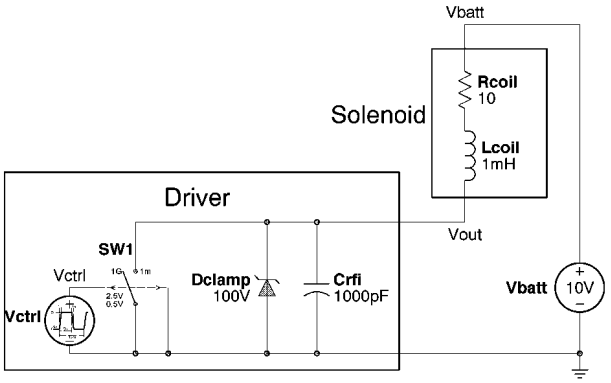

Figure 1 – Solenoid & Driver

The Natural Resonance Frequency (NRF) is:

$$F_{Res} = \frac{1}{2p\sqrt{L_{Coil}(C_{RFI} + C_{Zener})}} \approx 153.2kHz$$

The Decay (or Damping) Time-Constant is:

$$t_{Decay} = 2\frac{L_{Coil}}{R_{Coil}} = 200ms$$

The time to decay to ½of peak amplitude is:

$$T_{\frac{1}{2}Amplitude} = 0.69315 * t_{Decay} = 138.6ms$$

The Apparent Resonance Frequency (ARF, i.e. what you would measure with an oscilloscope) is:

$$F_{Osc} = \sqrt{F_{Res}^{\ 2} - \left(\frac{1}{2p * t_{Decay}}\right)^2}$$

$$\approx \sqrt{153.2kHz^{\ 2} - 795.8Hz^{\ 2}}$$

$$\approx 153.197kHz$$

Key points that a designer would be interested in observing/learning from the simulation are:

- ♦ *Output voltage rise and fall times*
- ♦ *Load current rise and fall times*
- ♦ *Percentage of overshoot (if any)*
- ♦ *Amplitude/frequency of ringing (or oscillation)*

What kind of results do we expect from this kind of circuit? This is a "textbook example" of an under-damped system, which exhibits severe ringing just after the end of the "Avalanche" interval. PSPICE easily gives us the classical result shown in Figure 2.

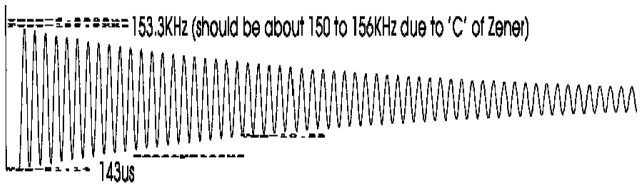

Figure 2 – Driver Output Voltage (PSPICE)

However, with Saber's user-adjustable DTS, we can observe any number of different results. Unfortunately, the result we get with the default settings is nowhere close to "correct". Figure 3 illustrates only a few of the possible "answers" the engineer might observe.

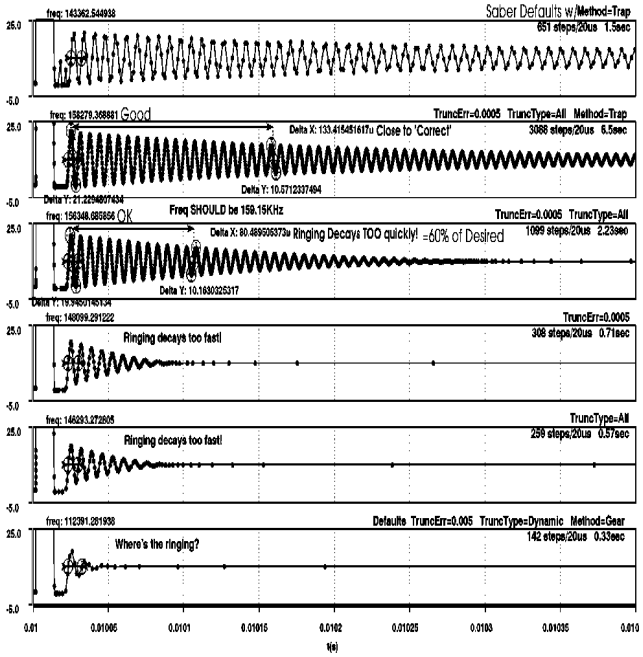

Figure 3 – Driver Output Voltage (Saber)

The bottom waveform was the result of the DTS control parameters being set to their defaults. Notice that there is hardly any ringing/oscillation present. The next two waveforms illustrate what happens when the "Truncation Type" and "Truncation Error" control parameters were changed individually; some ringing is present, but nowhere near "correct". The fourth waveform from the bottom results from changing the previous two parameters together whereas the fifth waveform also changes the "Integration Method" from "Gear" to "Trapezoid". This setting provides the most correct result. The top-most waveform resets the "Truncation Type" and "Truncation Error" controls back to their default settings, but leaves the "Integration Method" set to "Trapezoid".

When you consider the four major and two minor control parameters that preside over Saber's DTS algorithms, you soon realize that there are an overwhelming number of combinations that an engineer might use to achieve useful results.

Table 1 – Saber DTS Control Settings

Parameter	# of Settings	Settings
Truncation Error	3	Default ± 1 decade (0.05, 0.005, 0.0005)
Truncation Type	3	Method to use (Dynamic, Static, All)
Truncation Normalization	6	Formula to use (1, 2, 3, 4, 5, 6)
Integration Method	2	Method to use (Gear, Trapezoid)
Number of Same Points	3	Default + 2 decades (1, 10, 100)
Sample Point Density	3	Default + 2 decades (1, 10, 100)
	972	

As you can see from Table 1, "Calibrating the simulator" can be a very daunting task. It would be nice if there was an easier way to get more time steps when you need them.

The "Limit Step-Out" Template

The first enhancement to the DTS we can make is to override Saber's desire to overzealously relax the time steps whenever it thinks the waveforms are simple and smooth. Saber automatically relaxes the time steps by 10X whenever it decides that solving the system of equations was "easy". For instance, if the current time step was 1µs away from the last time step, then the next time step will be 10µs away and then 100µs, etc. A simple, rounded voltage rise, as typically occurs with RC-filtered pulses, gets coarser and coarser as the voltage rises. On the other hand, PSPICE only relaxes the time steps by a factor of 2X.

Of course, both simulators have this "hard coded", and the user ordinarily cannot adjust it. However, Saber's MAST modeling language allows a model (template) to limit the spacing to the next time step. The **`step_size`** system variable can be set in the **`values`** section of the template to dictate that the next time step can not be any farther out in time than what the template demands. Thus, we can make a very simple symbol/template pair to control the rate at which time steps are relaxed.

The gist of the code is:

```
template Limit_StepOut = \ # Model Name
   StepOut_Factor          # Parameters
Number StepOut_Factor = 2
{
when(time_step_done) {
     Time_Step = time – Last_Time
     Last_Time = time
     Desired_Step_Size =
          Time_Step * StepOut_Factor
     }

values {
     step_size = Desired_Step_Size
     }
}
```

Simply explained:

- *At the end of each time step (event captured by the* **`when(time_step_done)`** *construct), the template calculates the size of the current step (***`Time_Step`***) by subtracting the current time (***`time`***) from the time it was when the last time step was finished (***`Last_Time`***).*
- *The current time step size is multiplied by the user-specified relaxation rate (***`StepOut_Factor`***), which defaults to 2X (to mimic PSPICE), in order to determine how large the next time step can be (***`Desired_Step_Size`***).*
- *The DTS is constrained (via* **`step_size`** *system variable) as to how large it can make the next time step (***`Desired_Step_Size`***).*

The DTS gathers up all of the **`step_size`** constraints given to it by the various templates in the system, and uses the smallest one to constrain the distance to the next time step.

The benefits of this technique are astounding. Figure 4 shows the rising current in the load inductance of the circuit in Figure 1.

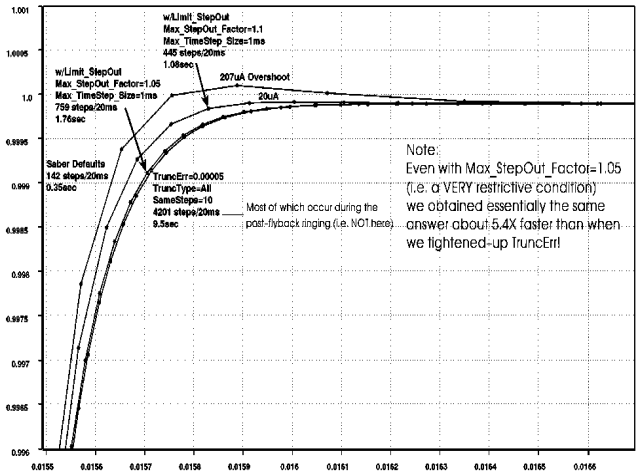

Figure 4 – Limit_StepOut Operation

We all know that the current is supposed to rise smoothly to something just under 1A (10V into 10Ohm coil + switch resistance). However, letting Saber use its default DTS settings shows us a very coarse edge followed by 200µA of overshoot. Yes, Saber kept us down to 142 time steps, but gave us a rough (and wrong) answer. Adding the **`Limit_StepOut`** block, with **`StepOut_Factor`** set to 1.1 gave a slightly smoother curve and reduced the overshoot to 20µA (10X smaller). The number of time steps only increased to 445. Changing the **`StepOut_Factor`** to 1.05 yielded a much smoother curve (with no overshoot) with 759 time steps, which rivaled that of the 4201 time step result obtained by adjusting three of Saber's DTS control parameters. Thus, we were able to obtain in about 1.76 seconds a waveform with the same fidelity and accuracy as what normally would have taken Saber about 9.5 seconds. Not quite as fast as the wrong answer, using the default settings, that we obtained in 0.35 seconds, but well worth the wait. Keep in mind that without the **`Limit_StepOut`** template the accurate answer would be forthcoming IFF we happened to find the 3 or 4 "silver bullets" (parameter settings) to give us the right answer. There was a lot of experimentation needed to get that answer in the first place!

The "Target_Crossing" Template

Another technique to force the DTS to generate time steps often enough is to tell the simulator at which values of a given signal a time step must occur. For instance, it is common for analog circuit designers to talk about rise/fall times in terms of 10%/90% points. Similarly, it is common for digital circuit designers to talk about delay times in terms of 50% points, or even 30%/70% (CMOS logic level) points. It would certainly be handy to have time steps that occur at precisely those levels, so that we can avoid inaccuracies in the linear interpolation between whatever time steps happened to be near those levels.

The Saber MAST modeling language provides a construct to generate an interrupt (event) when a signal has crossed through a specified level. MAST also provides a construct that allows the event-driven simulator to force the time-domain simulator to backtrack and recalculate values at any desired point in time. Because of these two constructs, it is easy to create a template (model) which watches the voltage of a node or the voltage between two nodes and force analog time steps to occur at user-specified levels.

The gist of the code is:

```
template Target_Crossing  \  # Model Name
        P                 \
        M                 =  # Pins
        Targets              # Parameters
electrical P, M
number Targets[10] = [ 0.50, # 10% of Vcc
                       1.50, # 30%
                       2.50, # 50%
                       3.50, # 70%
                       4.50, # 90%
                       0.00, # 0%
                       5.00, # 100%
                      -0.50, # Gnd Clamp
                       5.50, # Vcc Clamp
                       inf ] # Unused
{
when(threshold(v(P,M),Targets[1])) {
    schedule_next_time(time)
    }
.
.
.
when(threshold(v(P,M),Targets[10])) {
    schedule_next_time(time)
    }
}
```

Simply explained:

- *The user specifies up to ten voltage levels at which time steps should occur (`Targets`).*
- *The voltage between the inputs (`P` and `M`) are monitored (using `when(threshold())` construct).*
- *When any of the ten thresholds are crossed, an analog time step is forced (using the `schedule_next_time` construct) at the point in time (`time`) which the built-in linear interpolation routine estimated the threshold was crossed.*
- *The analog simulator throws away the data for the just-calculated time step and goes back to the specified point in time.*

This can be illustrated pictorially:

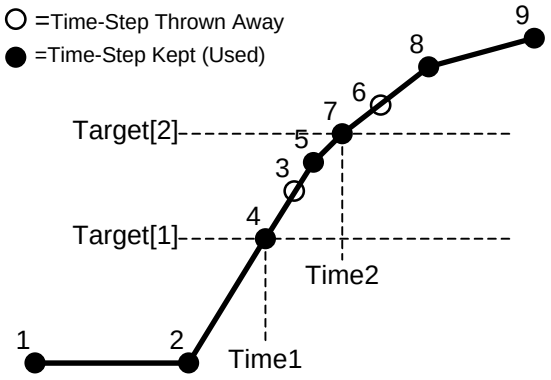

Figure 5 – Target_Crossing Explanation

The simulator is running along solving the system of equations at time steps 1 and 2, both of which are

below the user-specified threshold (**`Target[1]`**). The conditions of the system change such that the signal level starts to rise, and a time step is calculated at position 3. However, this new level is above the threshold, triggering the **`when(threshold())`** construct, passing down the estimated time of that crossing (time). The **`schedule_next_time()`** construct is used to force the analog simulator to go back in time, and generate a time step at position 4. Once completed, the DTS resumes its normal operation, and commands the next time step to occur at position 5. The situation repeats itself when the time step at position 6 occurs, which is above the second threshold. Backtracking is performed, and the time step at position 7 is calculated.

Returning to our test circuit from Figure 1, applying targets of –0.5, 0.0, 0.5, 5.0, 7.5, 10.0, 12.5, 15.0, 19.5, 20.0:

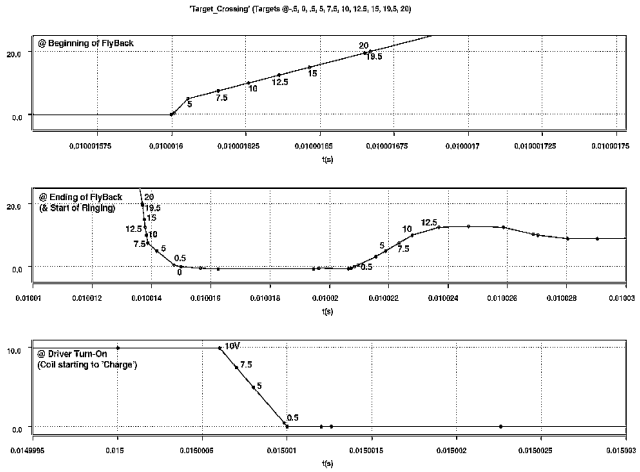

Figure 6 – Target_Crossing Operation

As you can see, time steps occurred precisely at the voltages that were commanded. With Saber's DTS control parameters set to their default values, Saber didn't attempt many more time steps than that which were required to fulfill the needs of the **`Target_Crossing`** template. Without this model, the time steps at the beginning and ending of the avalanche interval were very far apart and produced a very crude representation of the time response of the circuit. In fact, not shown here are the targets that were specified in order to smooth out the curve at the peak of the avalanche interval at approximately 100V. Without this block, the DTS generates so few time steps that the voltage actually overshot the 100V clamp level (for two time steps). Using this block, with targets of 90, 95, 96, 97, 98, 99, 99.5, 100, and 100.5 volts, forced the DTS to slow down as the signal approached the known target, thus avoiding the fictitious overshoot condition.

Output Driver (Source) Control

A technique similar to "Target_Crossing" can be hard coded into a model of an output driver, or signal source, in order to control the time steps during its state changes. The standard Saber "pulse" sources schedule time steps to occur at the beginning and ending points of the linear ramps (rise and fall intervals). They do nothing, however, to provide extra time steps along the way. It is up to the circuitry connected to the source to govern when the other time steps will be scheduled.

When writing our own source (driver) models, we can enhance the performance of the DTS by scheduling time steps at known points during the rise/fall ramps. For instance, if we want to force time steps at 10, 30, 50, 70, and 90% points, we could use the following code:

```
T_Start = time
T_Stop = T_Start + T_Transition
schedule_next_time(T_Start)
schedule_next_time(T_Start +
     (0.1 * T_Transition))
schedule_next_time(T_Start +
     (0.3 * T_Transition))
schedule_next_time(T_Start +
     (0.7 * T_Transition))
schedule_next_time(T_Start +
     (0.7 * T_Transition))
schedule_next_time(T_Start +
     (0.9 * T_Transition))
schedule_next_time(T_Stop)
```

In this example T_Transition is the rise (or fall) time of the transition. The result of these commands is shown pictorially in Figure 7.

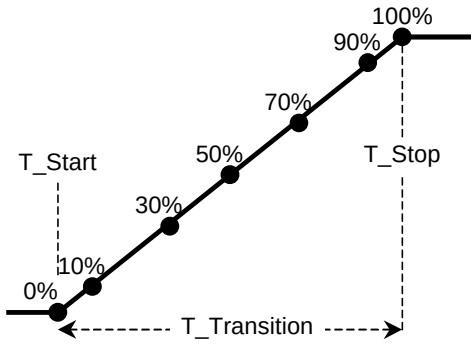

Figure 7 – Output Driver Control Explanation

If the DTS senses, from other portions of the design, that more time steps are needed, it will still generate them. All this technique does is guarantee

that time steps will occur at precise points along this waveform.

The "Angle_DTS" Template

A similar technique can be used in the electromechanical domain as well. A Variable Reluctance (VR) sensor generates an electrical signal (voltage) as a rotating toothed wheel (or gear) spins past a magnetically biased coil of wire. A mixed-domain model of the sensor can be created by monitoring the mechanical "shaft position" input, and using it as the index into a look-up table to obtain the value to be applied across the electrical "voltage" output. Unfortunately, due to the nature of the VR signal, the standard DTS algorithm has a hard time generating enough time steps during the "interesting" parts of the waveform. Left unassisted, the DTS often spreads out the time steps so rapidly that the signal generated looks nothing like what it should. Since we were the ones who generated the look-up table, we know which shaft angles correspond to "interesting" portions of the VR waveform, and which do not. We can thus force the time steps to be closer in the "interesting" portions and relax them elsewhere. The gist of the code is:

```
template Angle_DTS  \ # Model Name
       Shaft_Posn     # Pin
rotational Shaft_Posn
{
values {
    Rev_Num = int(Shaft_Posn/360)
    Shaft_Angle = Shaft_Posn -
        (Rev_Num * 360)
    Angle_Step = 10
    if(abs(Shaft_Angle)<30) {
        Angle_Step = 2
        }
    if(abs(Shaft_Angle)<10) {
        Angle_Step = 0.4
        }
    step_size = Angle_Step / Degs_per_sec
    }
equations {
    Degs_per_sec: Degs_per_sec =
        d_by_dt(ang_deg(Shaft_Posn))
    }
```

Simply explained:

- *The model monitors the rotational shaft angle (`Shaft_Posn`), and calculates the position in degrees in a "modulo" fashion.*
- *If the shaft angle is within 10 degrees of the zero crossing, the desired shaft angle change per time step is 0.4 degrees.*
- *If the shaft angle is between 10 and 30 degrees of the zero crossing, the desired shaft angle change per time step is 2 degrees.*
- *If the shaft angle is greater than 30 degrees of the zero crossing, the desired shaft angle change per time step is 10 degrees.*
- *The `step_size` system variable is used to constrain Saber's DTS algorithm so that the next time step is the desired distance from the current time step.*

The results are shown below:

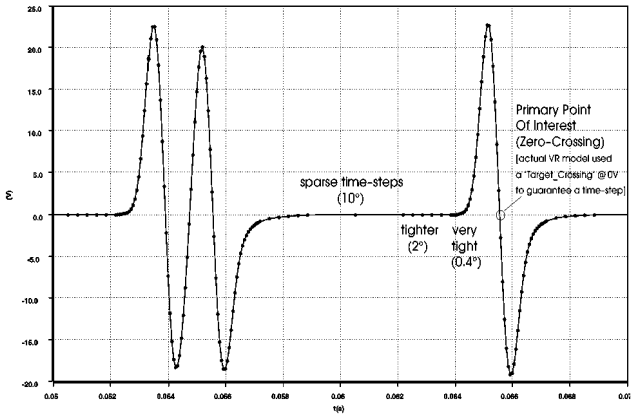

Figure 8 – Angle_DTS Example

About The Author

Jim Bach is currently employed as a Senior Project Engineer in the Hardware Analysis and Simulation group of Delphi Delco Electronics Systems, where he uses Avanti's Saber simulator to create circuit- and device-level electrical and electrothermal models. Jim co-authored two US patents related to solenoid driver circuit design.