Digital Force White Paper

Date: 13th February 2005
(Last Build: February 13, 2005)

Realtimify — A small Tool for Real Time SystemC™ Simulations

Mario Trams
Mario.Trams@digital-force.net

Digital Force / Mario Trams
http://www.digital-force.net

Realtimify — A small Tool for Real Time SystemC™ Simulations

Mario Trams

Mario.Trams@digital-force.net

Abstract

This paper discusses Realtimify — a small helper module for SystemC applications. This module is intended as a simple to use approach to carry out SystemC simulations in real time. Realtimify can be easily plugged as it is into most SystemC applications. However, because of its simplicity it can be also easily adapted to the special needs of the according application.

1 Introduction

Realtimify (pronounced like "real"–"time"–"ify") is a tiny helper module for SystemC models that are intended to be simulated in real time. That is, one simulated second should stretch across one second in real time, for instance.

There are various examples where such "realtimification" is suitable or even necessary. One good example are those models that include interactive control and visualization over the simulation process.

But there is also opened another completely new application of SystemC as such. That is, **exactly** the same SystemC model that is used for pure simulation purposes can be used as primary controlling instance for the final application. Imagine a machine consisting of sensors, actuators, and a controller that is generating new actuator commands depending on the current state and sensor inputs. This controller might be more or less complex and can be modeled in SystemC. (Notice that we are not necessarily speaking about some synthesizable model that later will become some piece of hardware or a combination of hard- and software. We are just speaking about a general behavioral model.) This controller could be simulated and debugged based on artificial sensor data. Later, the artificial sensor stream is "simply" replaced by readings from real sensors and the model controls real actuators rather than virtual ones.

In that sense, SystemC is not just used for simulation and modeling purposes. Instead, it becomes the runtime environment for the actual application.

Of course, in many cases the dramatic performance losses due to the significant overhead induced by SystemC will be the most limiting factor making this approach unsuitable for many applications.

2 Operational Principle

The basic operational principle is very simple so that it is almost not worth to speak about it.

In regular application–specific intervals the current simulation time t_{sim} is compared against the real run time t_{run} that has been spent so far. The difference $t_{diff} = t_{run} - t_{sim}$ between both times is the time the simulation is ahead of real time. For a real time simulation t_{diff} should be zero. So all that needs to be done is to suspend the simulation process for t_{diff} before proceeding.

Realtimify is based on `gettimeofday()` and `usleep()` which are both standard operating system calls. That is, the smallest resolution Realtimify can deal with is one micro second.

3 Limitations

As just described, Realtimify can work with resolutions up to one micro second. Formally, the resolution could be increased even more. For instance with the use of RDTSC on IA32 systems in junction with `nanosleep()`. But even a resolution of one micro second is questionable considering the power of today's computers. Realistic resolutions probably start at hundreds of micro seconds. In the end, this does also heavily depend on the application's complexity as well as the actual application needs. For simple interactive simulations a resolution of 0.1s to 0.04s should be fine in order to achieve a smooth behavior.

The precision of the realtimification depends mostly on the operating system characteristics. That is, for hard real time requirements one also needs an according real time operating system. On standard Unixes (including Linux), the suspension of the process with `usleep(100)` means a suspension of at least 100 micro seconds. Depending on the process scheduler and other pending activities, it might take slightly more time. Fortunately, Realtimify is based on absolute time measurements. This avoids systemic errors that are accumulating over the time. That is, when a certain `usleep()` took somewhat longer than planned, the next `usleep()` will automatically suspend the process for a shorter period.

Of course, Realtimify can only compensate run time differences when the simulation time is ahead of real time (i.e. $t_{diff} > 0$). It is not a time-machine that could compensate those cases where real time is ahead of simulation time. When this is the case, the simulated model is just too computing–intensive to be simulated in real time.

4 Using Realtimify

Using Realtimify is fairly simple. All that is needed is to instantiate the Realtimify module. The synchronization interval is to be specified directly along with the module instantiation. The interval can be specified either as `double`/`sc_time_unit` pair or as `sc_time` object.

The following example sets the interval to one second using the first approach.

```
realtimify REALTIME("realtimify", 1.0, SC_SEC);
```

Setting the interval using an `sc_time` object does not differ very much:

```
sc_time interval(1.0, SC_SEC);
realtimify REALTIME("realtimify", interval);
```

Together with Realtimify there is also provided a small example demonstrating the application.

5 Technical Issues for the Use of Realtimify

The synchronization interval should be well selected to match the requirements of the application. For instance, when there is to be written out a message every tenth part of a second, the resolution should be at least 0.1s. When the resolution would be 1 second for this example, one would receive every second a bunch of ten messages, which might be certainly not desired. Notice that the exemplary model might still incorporate other internal processes working at a much higher resolution than just 0.1s! The only important thing is that interactions with the outside world are handled in real time.

When the Realtimify process becomes scheduled by the SystemC kernel, this happens in the evaluate phase of the very first delta cycle of the according simulation time. That is, the update phase of the first delta cycle as well as additional delta cycles will be processed after the synchronization. So it is guaranteed that no signals become changed prior to the corresponding real time.

However, because the scheduling process of SystemC is not deterministic, there cannot be guaranteed that no other SystemC thread/method becomes scheduled prior to the Realtimify method. As just stated above, this still means that no signals become changed too soon, as this will be done in the following update phase of the delta cycle. Nevertheless, it might be that some external actions are carried out too early. We can consider again a process that is writing out a message every second. When a message is to be written out at simulation (= real) time of 10 seconds, the message should appear in the vicinity of 10 seconds rather than 9 seconds, for instance.

Basically, there are two work–arounds to deal with that situation:

1. Increase the Resolution.
   Although a realtimification would call for a certain resolution (for instance, say 1 second), the actual resolution could be increased. When we are speaking about some human interface that is generating one message every second, the resolution could be set to 0.1s. Considering human perception, it will certainly make no difference when a message that is to appear at offset 10.0s actually appears at offset 9.9s. The same is true when we are talking about interactive systems and relative times. When a reaction for a certain action is expected after 1.0 seconds, one won't notice when the reaction appears already after 0.9 seconds.
2. Insert a Delta Cycle.
   Before doing anything worthwhile, the application processes could simply delay its actions by another delta cycle. In case of `SC_THREAD` this can be easily accomplished by `wait( SC_ZERO_TIME )`. In case of `SC_METHOD` this is problematic. Unfortunately, `next_trigger( SC_ZERO_TIME )` does not pass the control to other processes, and hence evenly not to the Realtimify process.
   Anyway, in many applications the additional delta cycle appears inherently. That is the case, when the critical processes are triggered indirectly by events caused by other processes.

6 Realtimify Code

The code of Realtimify is rather simple and widely self–documented. Listing 1 shows the header file. Because of the additional constructor arguments the code differs a little bit from the conventional way.

The implementation code of Realtimify is shown in listing 2 and 3.

The constructors are more or less just wrappers for the actual initialization function `init()`. The primary task of the `init()` function is to determine a scale factor (or rather divisor) that is later needed for converting the result of `sc_simulation_time()` into micro seconds. As stated by the comments, the calculation of the scale value is somewhat strange because of floating point issues.

Also notice that the starting real time of the simulation is not taken during the initialization but at the

```
29 SC_MODULE( realtimify ) {
30
31   // Declaration of public module constructors.
32   SC_HAS_PROCESS( realtimify );
33   realtimify(sc_module_name name_, double interval_time, sc_time_unit interval_unit);
34   realtimify(sc_module_name name_, sc_time interval_);
35
36  protected:
37
38   // Declaration of the actual synchronization function.
39   void process();
40
41   // Some additional helper function...
42   void init();
43
44   // interval stores the synchronization interval (resolution)
45   sc_time interval;
46
47   // start_real_time_usecs stores the real time when the simulation has been started
48   unsigned long long start_real_time_usecs;
49
50   // scale stores a scalar used to convert the result of sc_simulation_time into
51   // micro seconds
52   double scale;
53
54 };
```

Listing 1: `realtimify.h`

first invocation of the realtimify process. The background of this feature is the point that the elaboration phase of the SystemC application might take a significant amount of time. When the real time would be counted from the very beginning, the real time can be significantly ahead of the simulation time when the actual simulation starts — meaning we get real time violations which might cause undesirable behavior. This means, at the beginning the simulation will progress rather quickly and faster than in real time until the real time is caught up by simulation time.

The actual Realtimify–process (listing 3) is not complicated as well. In case the simulation has been just started, the current real time is determined and stored in `start_real_time_usecs`. Otherwise the difference between current simulation time and real time is determined and the application process becomes suspended accordingly by using `usleep()`. When the difference is negative, there can't be done very much from a general point of view. As it can be seen in listing 3, Realtimify can write out a warning when `DEBUG` is defined for compilation. However, whether there could be done something sensible to deal with this situation depends on the actual application. In most cases there won't be any choice except to try the simulation on a faster machine. But there are also applications where Realtimify could automatically adjust some runtime parameters without hurting the overall simulation result. Imagine some simulation application that is visualizing the result graphically at a standard rate of say 25Hz. In this case Realtimify could be slightly changed so that it can dynamically adapt the display rate depending on the current situation.

7 More Potential for Realtimify Enhancements

Realtimify in its basic form as described herein harbors a problem that is not immediately visible. The point is that results which are needed at real time T are generated beginning with real time T. As the calculation takes some time, there will always be some delay until the results are available. Depending on kind and complexity of the application, this delay could be not desired and/or even harmful — although the application complexity does still allow for a real time simulation.

Imagine some application that does something every second and every step requires 0.5s of computation. This implies two facts: Actually, the model could be twice as complex to be simulated in real time. But in worst case the results are available half a second too late.

Instead of just waiting for a certain real time T, the results that are to appear at T could be already calculated in advance, if applicable. However, this is a little bit difficult to achieve. The problem is that there is needed some closer interaction between the actual application and Realtimify. That is, the application needs to be well aware about the issue and informs Realtimify when (at which delta cycle) it can carry out the synchronization. This renders the presence of a general (transparent) Realtimify module somewhat useless, as in this case the synchronization could be also done directly by the application.

An alternative would be to drive the application in some kind of master–slave fashion. E.g. calculate the results for T at $T - 0.5s$, and just output the results at T.

```
36 //
37 // Realtimify module constructor - checking interval given as double/sc_time_unit pair
38 //
39 realtimify::realtimify(sc_module_name name_, double interval_time, sc_time_unit interval_unit):
40   sc_module(name_) {
41
42   // Convert the double/sc_time_unit pair into an sc_time object.
43   interval = sc_time(interval_time, interval_unit);
44
45   // Call the actual initialization function.
46   init();
47
48 }
49
50 //
51 // Realtimify module constructor - checking interval given as single sc_time object
52 //
53 realtimify::realtimify(sc_module_name name_, sc_time interval_):
54   sc_module(name_), interval(interval_) {
55
56   // Call the actual initialization function.
57   init();
58
59 }
60
61 //
62 // The actual initialization function...
63 //
64 void realtimify::init() {
65
66   // In case the specified interval for performing a real time check and bringing
67   // the application in sync (if needed) is smaller then one microsecond, we issue
68   // a warning.
69   if (interval.to_seconds() < 1e-6) {
70     cerr << "Realtimify Warning: The specified Realtimify interval (" << interval << ")" << endl;
71     cerr << "                    is below micro second resolution (proceeding anyways)!" << endl;
72   }
73
74   // Register the actual Realtimify SystemC process (as SC_METHOD). The method is
75   // self-timed (see below) and is not sensitive to any signal.
76   SC_METHOD( process );
77
78   // Later, start_real_time_usecs holds the real time when the simulation has been
79   // started. A value of zero means that this starting time has not yet been taken.
80   // Note: We do not take the starting time right now, i.e. during the elaboration
81   //       phase. Instead, it is taken at the first simulation cycle. This is, because
82   //       depending on the simulation model the elaboration phase might take a
83   //       significant amount of time.
84   start_real_time_usecs = 0;
85
86   // Get the default time resolution and convert it to micro seconds.
87   // This is needed, because we have to qualify the result of sc_simulation_time(),
88   // as this is unit-less.
89   sc_time default_time = sc_get_default_time_unit();
90   double default_time_usecs = default_time.to_seconds() * 1000000;
91
92   // Determine the scale value used to convert default time units into micro seconds.
93   // Note: This is not done by a simple division as this can introduce significant
94   //       rounding errors. Instead, we do trial&error ...
95   // Note: A check for "< 0.5" means actually "< 1" and a check for "> 5" means
96   //       actually "> 1". This is because due to rounding issues the value that
97   //       is being checked might never have a value of exactly 1 (e.g. 0.999999...
98   //       or 1.000000000...1).
99
100  scale = 1;
101  if (default_time_usecs <= 1) {
102    while ( (scale * default_time_usecs) < 0.5 ) {
103      scale *= 10;
104    }
105  } else {
106    while ( (scale * default_time_usecs) > 5 ) {
107      scale /= 10;
108    }
109  }
110
111  return;
112
113 }
```

Listing 2: `realtimify.cpp` (Part 1)

```
115 //
116 // The process (SC_METHOD) that is regularly synchronizing simulation time and
117 // spent real time.
118 //
119 void realtimify::process() {
120
121   unsigned long long current_time_usecs;
122   struct timeval tv;
123   unsigned long long current_real_time_usecs;
124   double current_simulation_time_usecs;
125   long long time_difference_usecs;
126
127   // Determine the current time in micro seconds.
128   gettimeofday( &tv, NULL );
129   current_real_time_usecs = (1000000 * tv.tv_sec + tv.tv_usec) - start_real_time_usecs;
130
131   if ( !start_real_time_usecs ) {
132     // This is the very first invocation of Realtimify. So we take the current
133     // real time which becomes the starting time.
134     start_real_time_usecs = current_real_time_usecs;
135   } else {
136     // This is not the very first invocation. So we have to see whether there
137     // have to be taken some actions.
138
139     // First determine the current simulation time in micro seconds.
140     current_simulation_time_usecs = sc_simulation_time() / scale;
141
142     // Next calculate the difference between simulation and real time.
143     time_difference_usecs = (long long) (current_simulation_time_usecs - current_real_time_usecs);
144
145     if ( time_difference_usecs >= 0 ) {
146       // When the difference is positive this means the simulation time is ahead
147       // of the real time. In this case we have to issue an according usleep() call.
148       usleep( time_difference_usecs );
149     } else {
150       // When the difference is negative this means the simulation time is behind
151       // the real time. In other words, the model can't be simulated in real time.
152       // Well, in this case we can't do much from here....
153 #fdef DEBUG
154       cerr << "Realtimify Warning: Real time violation!" << endl;
155 #endif
156     }
157
158   }
159
160   // Schedule the next synchronization task according the application-specific
161   // setting.
162   next_trigger(interval);
163
164   return;
165
166 }
```

Listing 3: `realtimify.cpp` (Part 2)

Though, either solution requires according awareness of the actual application. This might be certainly not desired.

Anyway, those issues could be addressed in future and perhaps according mechanisms could be generalized and placed into an enhanced version of Realtimify.

8 Summary

Realtimify is a neat tool that can be easily integrated into SystemC models in order to achieve an execution of the simulation in real time. In most applications Realtimify can be integrated out of the box without any changes. Because of its simplicity, it can be easily modified in order to meet the application's specific needs.

There have been shown some limitations and odds of this tool as well, leaving space for future optimizations and enhancements.