

Comparison of Teja and Simulink/Stateflow for real-time implementation of continuous-time dynamic systems

Tunç Şimşek

April 27, 2001

Introduction

We have compared the implementation of a simple continuous time system in Teja and Simulink/Stateflow. Original versions of the Simulink/Stateflow models were provided by Jason Souder.

There are two variables of interest related by:

$$\begin{aligned}\dot{x} &= 1 \\ \dot{y} &= x \\ x(0) &= y(0) = 0\end{aligned}$$

We are **explicitly** given the computational constraint that x must be computed once every 1 seconds and y every 1.5 seconds.

We are interested in all of the following aspects of the implementations that are possible with Teja and Simulink/Stateflow.

Representation of the Model In particular, the compactness of the representation and the possible alternate representations.

Quality of the approximation Here we assume that any implementation will use a forward Euler approximation. Observe that (with respect to the given constraints), the most accurate approximation will be given by:

time	x	y
0	0	0
1	$x(1) = x(0) + 1 = 1$	-
1.5	-	$y(1.5) = y(0) + 1.5x(0) = 0$
2	$x(2) = x(1) + 1 = 2$	-
3	$x(3) = x(2) + 1 = 3$	$y(3) = y(1.5) + 1.5x(1.5) = 2.25$
4	$x(4) = x(3) + 1 = 4$	-
4.5	-	$y(4.5) = y(3) + 1.5x(3) = 6.75$

(where a '-' means that no computation is allowed for that variable) We will compare results with these values of x and y .

Efficiency of the implementation The actual code that is executed to obtain $x(n)$ and $y(n * 1.5)$ should consume as little resources as possible (eg. sleep as much as possible) while respecting to the computational constraints.

1 The Teja implementation

In the Teja implementation we model the system as two classes illustrated in Figures 1 and 2. The rationale is that instances of each class will be scheduled at different rates. For the class **foo**, a self-transition is scheduled every 1 seconds and for **bar**, a self-transition is scheduled every 1.5 seconds. Our interpretation of the Teja semantics states that:

- continuous variables (such as x and y are implicitly updated only on discrete transitions)
- the update rule is given by $x(n + 1) = x(n) + 1$ and $y((n + 1) \cdot 1.5) = y(n \cdot 1.5) + 1.5x(n \cdot 1.5)$.

The system sleeps most of the time – waking up only on discrete transitions (that is, at $t = 1, 1.5, 2, 3, 4, 4.5, \dots$) Our experiment validates our interpretation:

t	x	y
0	0	0
1	1	-
1.5	-	0
2	2	-
3	3	-
3	-	2.25
4	4	-
4.5	-	6.75

where the '-' means that the variable is not updated. Note that $x(n \cdot 1.5)$ for odd values of n is obtained as:

$$x(n \cdot 1.5) = x(n) + 0.5$$

That is, the value is a first-order approximation of the last known value of x and the last known value of the right hand side of $\dot{x} = 1$.

Representation of the model The semantics of Teja state that computations are only performed on discrete transitions. In particular, the system supports a time-triggered transition called a **proaction**. The key feature of proactions are that they may be explicitly scheduled by analytic threshold conditions on continuous variables (for example, given that $timer = 1$, a proaction may be scheduled when $timer \geq 1$). At time 0, the system looks at the right hand side of the differential equation $timer = 1$ and performs a linear approximation of $timer$ as

$$timer(t) = timer(0) + t \cdot 1 \text{ for all } t \geq 0$$

where t is the **real-time**. Then, the system can analytically determine that the threshold $timer \geq 1$ should be scheduled for the real-time $t = 1$.)

In our example, we have explicitly used timers in this fashion to schedule the computations of $x(n)$ and $y(n \cdot 1.5)$ for all $n \geq 0$. The advantage is that we know the exact outcome of the computation and the amount of resources used (that is, we know by the semantics that the system will sleep all the time, waking up only on the scheduled proactions).

Furthermore, we have been told by Teja representatives that our implementation of the system is not the most efficient one. In particular, we were told that we need not use explicit timers to schedule proactions. Thus, we conclude that Teja provides several methods of implementing a given system.

Quality of the approximation With Teja, we have obtained the best possible approximation possible with respect to the given computational constraints. Furthermore, the system is equipped with a **slip-control** algorithm that provides 1st-order corrections to all continuous variables that may occur due to interrupt delivery latencies of a given underlying operating system. This, however, is not the subject of this report.

Efficiency of the implementation The system is as efficient as possible. That is, the system sleeps all the time, waking up only on the scheduled proactions. We do not, however, know the efficiency of the dynamic scheduler which executes on every discrete transition.

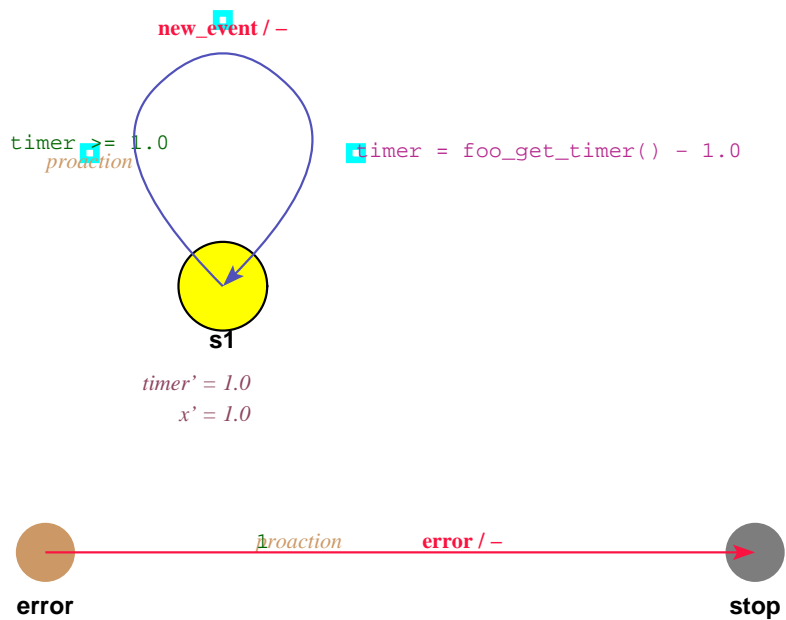


Figure 1: Teja implementation of $\dot{x} = 1$ at a rate of once every second. The class is named **foo**.

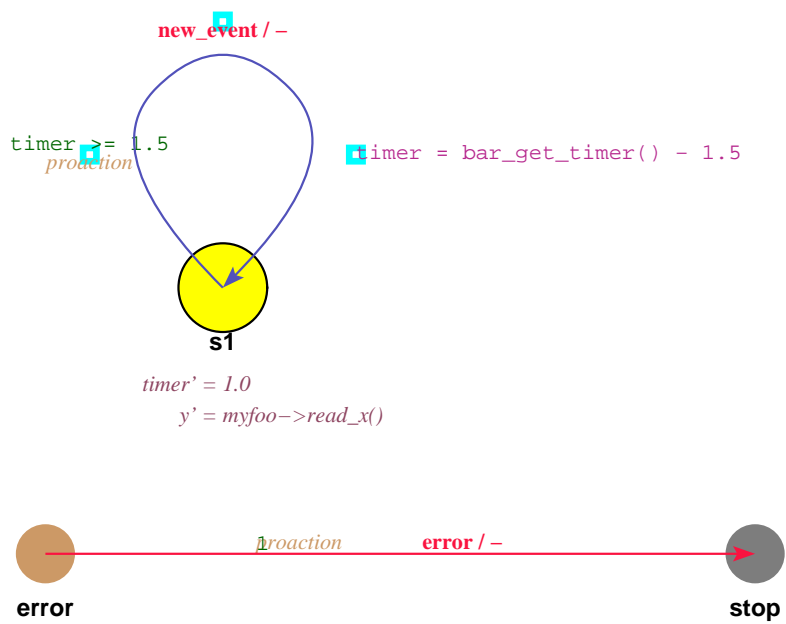


Figure 2: Teja implementation of $\dot{y} = x$ at a rate of once every 1.5 seconds. The x is obtained by explicit reference to **foo**.

2 The Simulink/Stateflow implementation

We have tried different implementations. First, a straightforward approach using Simulink alone, then using Simulink together with Stateflow (according to our interpretation of the Ford style guide).

2.1 Simulink only implementation

To be able to generate efficient real-time code we use discrete-time integrators as illustrated in Figure 3. The solver options are set to:

- fixed-step (automatic step size selection)
- Discrete (no continuous states) solver

For single-tasking mode, the simulation produces the results plotted in Figure 4.

Representation of the model The representation is very compact. One can immediately read-off the Simulink diagram that the system is an explicit (discrete-time) implementation of $\dot{x} = 1$ and $\dot{y} = x$.

However, the representation does not suggest how the actual implementation will be. This is mainly because all wires in a Simulink diagram are implicitly continuous. The draw-back is that it is very difficult to use alternate representations of a given model to obtain, say, a more efficient implementation.

Quality of the approximation To interconnect discrete-time blocks with different rates, we are explicitly required to use an intermediate sample and hold block. Thus, it is clear that we cannot get the best approximation possible. That is, we can never get

$$y((n + 1) \cdot 1.5) = y(n \cdot 1.5) + 1.5x(n \cdot 1.5)$$

for odd values of n since $x(n \cdot 1.5)$ will not be available for odd values of n . The solution for x and y are illustrated in Figure 4.

Efficiency of the implementation Simulink provides several real-time code generators for several different targets. However, we do not know when (and if) the system actually ever sleeps. In particular, we cannot infer how the system is scheduled to sleep and wake-up from the representation of the model.

Furthermore, we are not permitted to select multi-tasking mode for our example (for multi-tasking mode we are required to have sample rates which are integer multiples of each other).

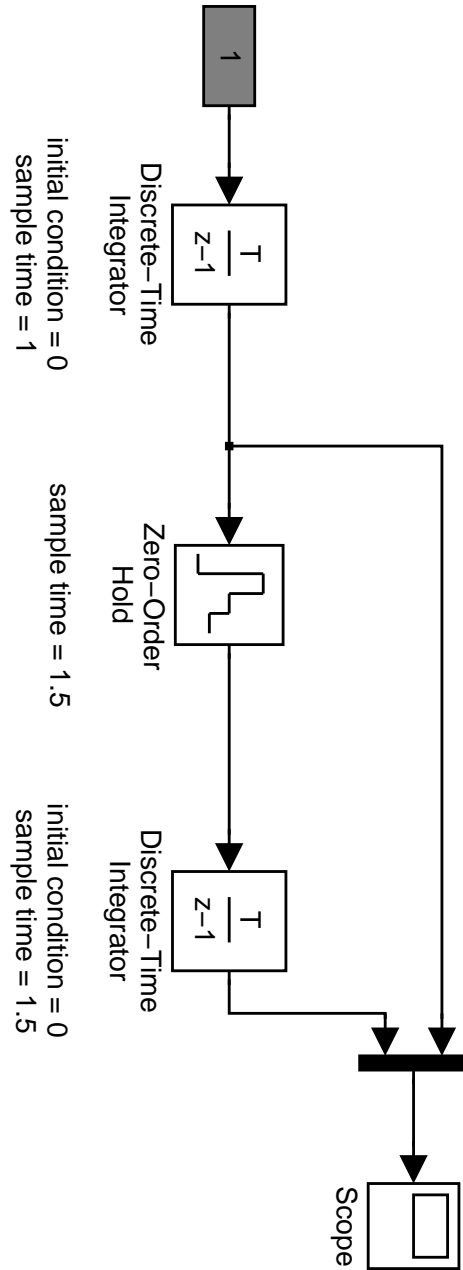
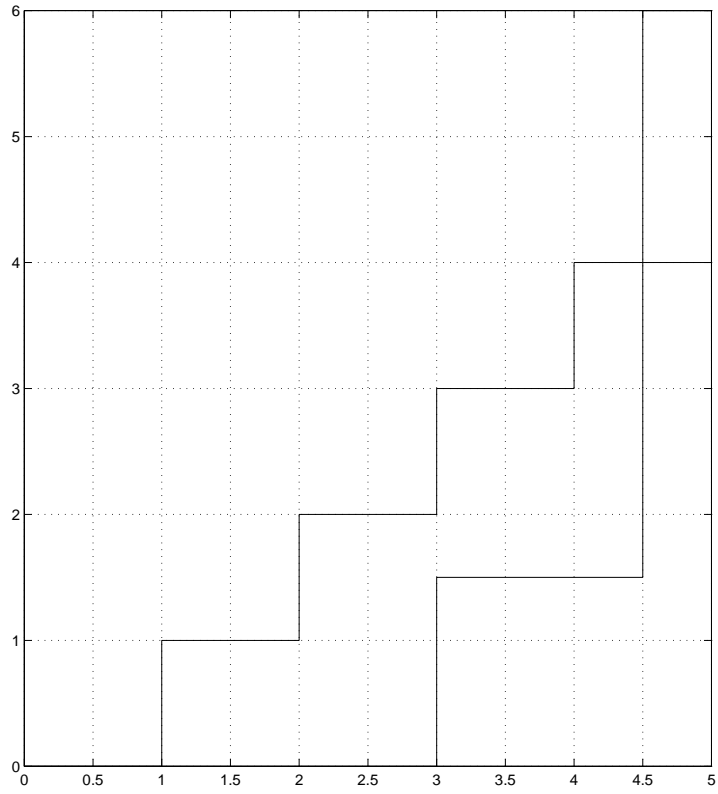


Figure 3: Simulink implementation of $\dot{x} = 1$, $\dot{y} = x$.



Time offset: 0

Figure 4: Simulink plot of x and y for single-tasking, fixed-step discrete solver with automatically selected step size.

2.2 Simulink with Stateflow implementation

The main deficiency of using Simulink alone is that the scheduling of the computations is not explicit in the representation of the model. Thus we cannot know for certain whether we are satisfying the given computational constraints. To overcome this problem we use Stateflow to schedule computations as illustrated in Figures 5-8.

Two clocks are used to generate periodic events with periods 1 and 1.5 seconds. A Stateflow scheduler is triggered by these clocks and produces discrete triggers to two separate discrete integrator blocks which explicitly implement $\dot{x} = 1$ and $\dot{y} = x$ using the forward Euler update rule given by $T/(z - 1)$.

In the implementation we will assume that the clocks are explicitly scheduled so that computations occur only on events generated by the Stateflow scheduler. This assumption is necessary because the Simulink clocks are in fact continuous-time entities and we do not know how they will be implemented.

Representation of the model The representation of $\dot{x} = 1$ and $\dot{y} = x$ is very compact. However, the representation of the scheduler is slightly more involved. In particular, Stateflow does not provide synchronous transitions, thus one must ensure an ordering (or priority) on the events generated by the scheduler. For example, in Figure 5, if we interchange the order of the clocks input to the multiplexer then a bad solution is obtained for x and y (see Figure 10).

The advantage is that the scheduling is explicit. As in the case with the Teja implementation, there are several alternate ways to explicitly schedule computations and to ensure that we satisfy the given computational constraints.

Quality of the approximation As in the case with the Simulink implementation, it is not possible to access the value of $x(n \cdot 1.5)$ for odd values of n . This is because the system is governed by a discrete-time scheduler which has no implicit support for time management. Thus, it is clear that we cannot get the best approximation possible. The solution for x and y is illustrated in Figure 9.

Efficiency of the implementation Since the scheduling of computations is explicitly performed by the Stateflow scheduler, we expect that the efficiency of the implementation will be governed by the efficiency of the code generated for the scheduler. The Stateflow manual states that the scheduler will sleep all the time, waking up only when it is triggered by the clocks. Thus the implementation is efficient.

Once again, we were not allowed to use multi-tasking since our clock rates are not integer multiples of each other.

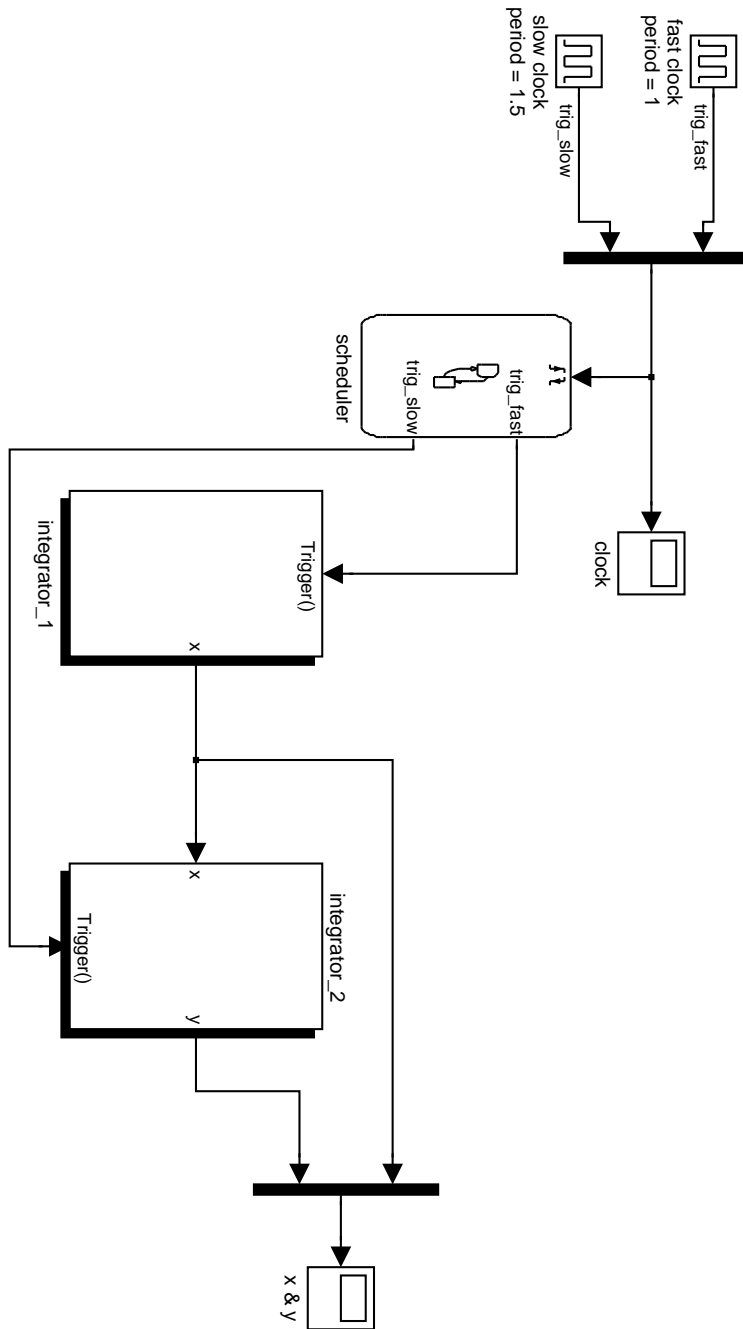
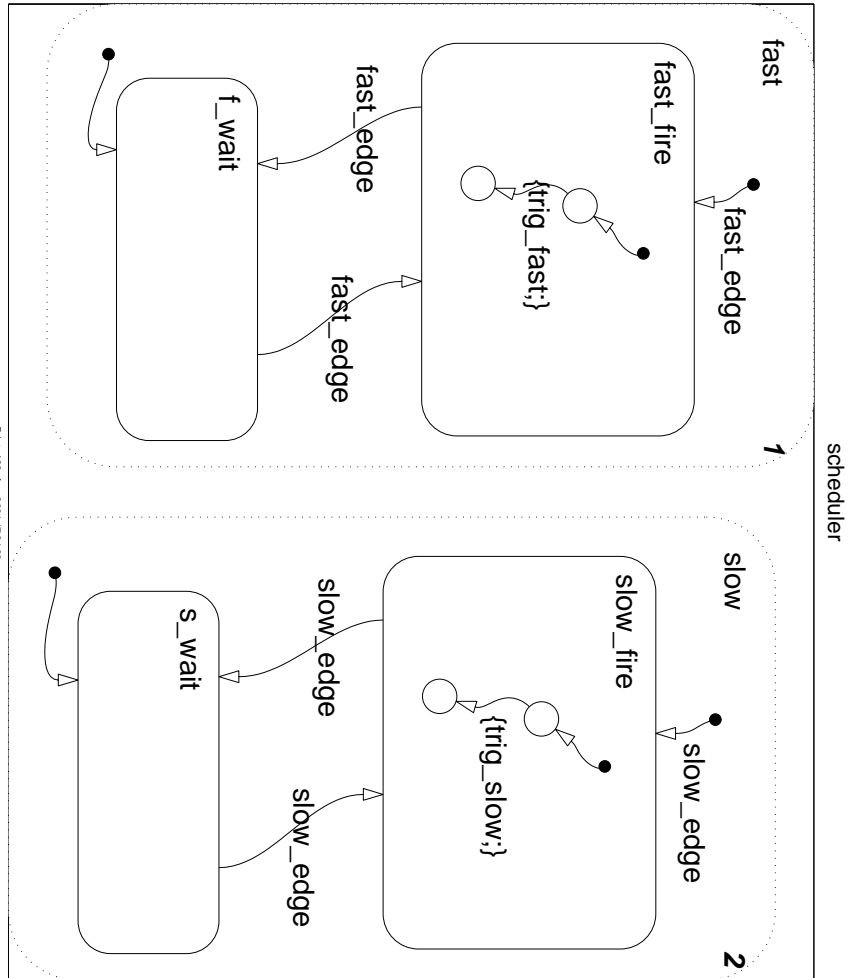


Figure 5: Simulink/Stateflow implementation of $\dot{x} = 1$, $\dot{y} = x$.



Paper/28-Apr-2001 17:31:30

Figure 6: Stateflow scheduler.

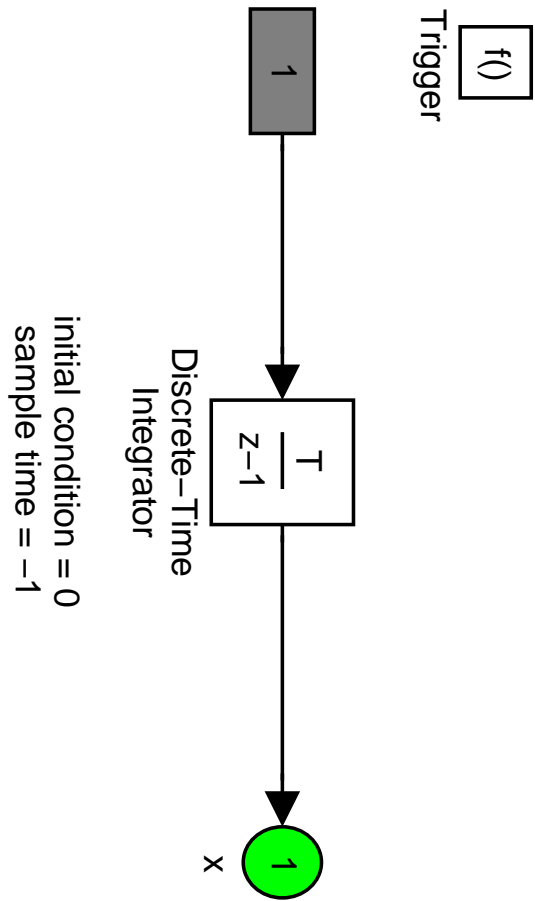


Figure 7: The triggered discrete integrator block for x .

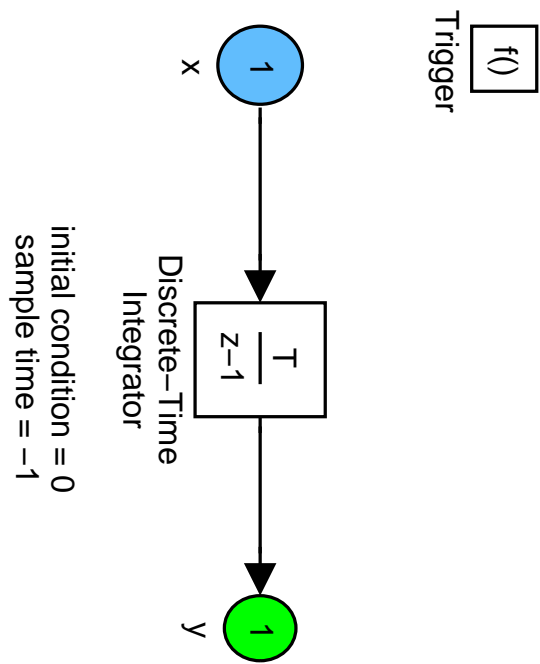
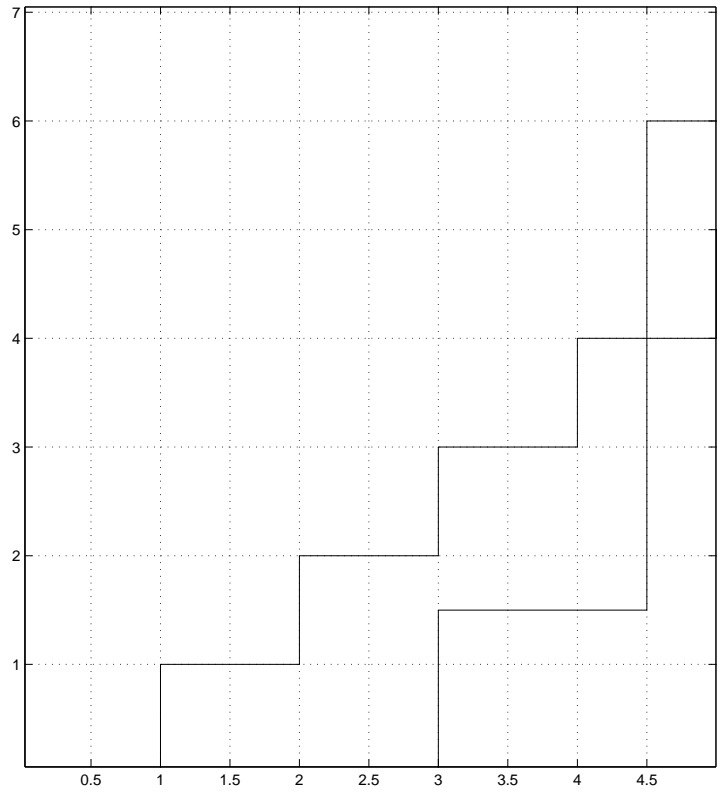
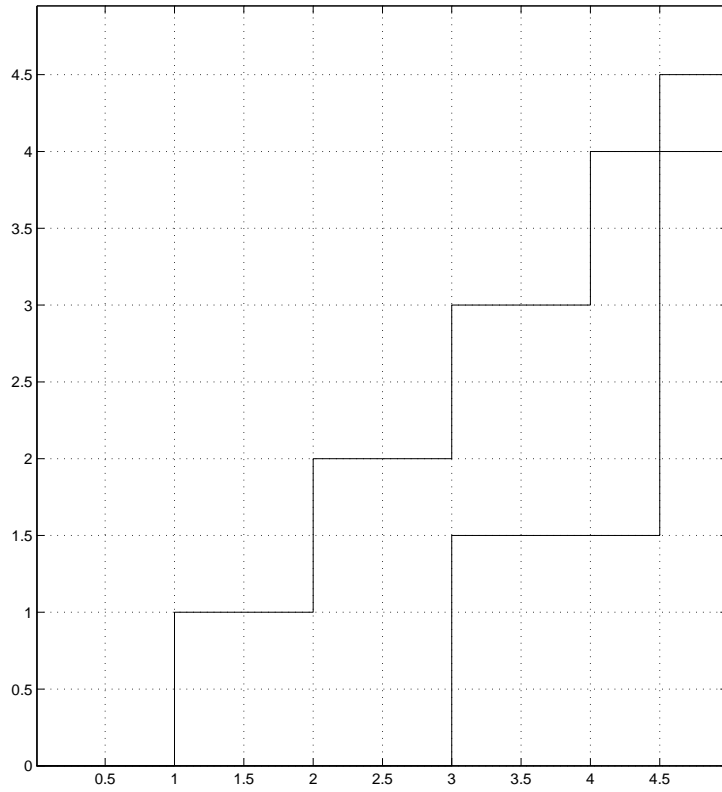


Figure 8: The triggered discrete integrator block for y .



Time offset: 0

Figure 9: Plot of x and y for Simulink/Stateflow implementation.



Time offset: 0

Figure 10: Plot of x and y for Simulink/Stateflow implementation with the ordering of the clocks interchanged.

3 Conclusion

We have attempted to implement a simple double integrator using Teja and Simulink/Stateflow. Our goal was to compare these tools with respect to the representation of the model, quality of the solution and the efficiency of the implementation.

Throughout the document we have discussed the advantages and disadvantages of each tool. Overall, we believe that Teja provides the highest quality solution (*ie.* the best approximation) and an efficient implementation. The biggest advantage is that the efficiency of the implementation is explicit in the representation of the model. That is, the system always sleeps, waking up only on explicitly scheduled **proactions**. In particular, it is possible to speak of alternate representations which may be more or less efficient without ever referring to the underlying code generated by Teja. We are also impressed with the **slip-control** feature which keeps the modeled timers in synchrony with the real-time and produces first-order corrections to continuous variables that may occur due to interrupt delivery latencies of an underlying operating system.

Simulink does not provide explicit support for scheduling computations. Thus, it is not possible to ensure that the implementation of the model will satisfy the computational constraints.

Simulink together with a Stateflow scheduler provides a flexible framework for implementing real-time controllers. When a real-time implementation is desired one should use discrete-time integrators. In this case the solution is not the best that is possible (in our example, we were not able to access values of $x(n \cdot 1.5)$ for odd values of n). As in the case with Teja, the Stateflow scheduler makes the computations and the implementation explicit. Thus it is possible to speak of alternate schedules which may be more or less efficient. However, we observed that the scheduling is slightly more tedious than Teja (in our example, the ordering of the scheduler events were important and the ordering is not immediate in the representation). The efficiency of the implementation is governed by the efficiency of the implementation of the Stateflow scheduler. The Stateflow manual states that the implementation of the scheduler sleeps, waking-up only when triggered by external clocks.