A Hard Real Time Demo for Dynamic Ticks and Timed SCCharts

Andreas Boysen, Alexander Schulz-Rosengarten, Reinhard von Hanxleden
Kiel University, Kiel, Germany, {abo | als | rvh}@informatik.uni-kiel.de

Abstract

Synchronous programming languages, such as Esterel, Lustre, SCADE or SCCharts, have been developed for designing reactive systems. They abstract from computation times and assume that outputs are synchronous with their inputs. This leads to a deterministic semantics, without race conditions, which makes synchronous languages particularly suitable for safety-critical systems. However, even though synchronous languages have been designed with real-time applications in mind, the handling of physical time is traditionally left to the execution environment. This makes e.g. the expression of arbitrary time-outs difficult and may lead to excessive "busy waiting" computations.
The recent proposal of dynamic ticks alleviates this by making physical time a first-class citizen within the synchronous programming model. In this paper, we explore and demonstrate the practical merits of dynamic ticks, including improved timing accuracy and reduced computational requirements, in the context of Timed SCCharts. As demonstration platform, we present a hardware/software platform that involves two stepper motors whose operation must be synchronized at microsecond accuracy to avoid physical damage.

1 Introduction

Synchronous languages are well-established for the modeling and programming of reactive systems. In particular for safety-critical applications, such as flight control, automotive applications or in the medical sector, the deterministic semantics and formal grounding of synchronous languages have proven their practical value [2]. The synchronous paradigm, which states that outputs of a system are "synchronous" with their inputs, divides computations into discrete "ticks" that conceptually take zero time. This is an abstraction from reality, since the computation of one tick, or one reaction, does of course take time. However, this abstraction is the basis for defining a concurrent semantics without race conditions, just like boolean logic gives a well-founded, deterministic semantics to circuits that abstracts from their physical implementation and actual stabilization delays. Classical synchronous languages include Esterel, Lustre and Signal [2]; more recent languages include the hybrid modeling language Zélus [3] and the statechart dialect SCCharts [9], which now is used in the railway domain; commercially most successful at this point is probably SCADE [6], with its qualified compiler that is routinely used by Airbus and other industrial players.
Clearly, synchronous languages have been developed for real-time applications. However, unlike other languages developed for that domain, such as Ada, traditional synchronous languages do not include language features that explicitly address physical time. Instead, time is typically modeled by counting occurrences of input signals that denote the passage of a certain amount of time, or by simply counting ticks if ticks are known to occur at a certain, fixed frequency. This mechanism is rather crude and has practical disadvantages, as observed by Bourke and Sowmya [4]. For example, if some input signal msec1 denotes the passage of 1 millisecond and another input signal msec10 denotes the passage of 10 milliseconds, a timeout waiting for 10 occurrences of msec1 does not necessarily take the same amount as another timeout that waits for one occurrence of msec10, as the actual waiting time depends on how the the timeouts are aligned with the timing input signals.
As von Hanxleden, Bourke and Girault have argued [10], one limitation of the traditional synchronous setting is how reactions are triggered. Specifically, it is traditionally the environment that decides on when reactions are computed. Typically, one of three options is used: 1) a *time-triggered* execution, where a reaction is computed for example once per millisecond; 2) an *event-triggered* execution, where reactions are computed whenever some input event occurs, such as for example the press of a button; or 3) an *asap* execution, where the next reaction is triggered as soon as the previous reaction is finished. Note that the triggering mode does not affect the synchronous scheduling of the reaction calculation itself. Each of these options has its merits and is fairly easy to implement, but neither of them is particularly suitable for handling precise, fine-grained real-time requirements. However, it turns out that the synchronous paradigm can be seamlessly extended with a fourth option that is more amenable for real-time requirements. Specifically, the recently proposed *dynamic ticks* [10] give the synchronous program control not only about how it reacts to current and past inputs, but also *when* the next reaction should occur. Their proposal included a prototypical realization in Esterel, and the theoretical advantages of that approach seem rather clear. Subsequently, the concept of dynamic ticks was incorporated in Timed SCCharts [7], which basically augment SCCharts with clocks as used in timed automata [1]. However, what was lacking so far was a practical evaluation, with very tight (i. e., microsecond scale) timing constraints, and a demonstrator with a hard

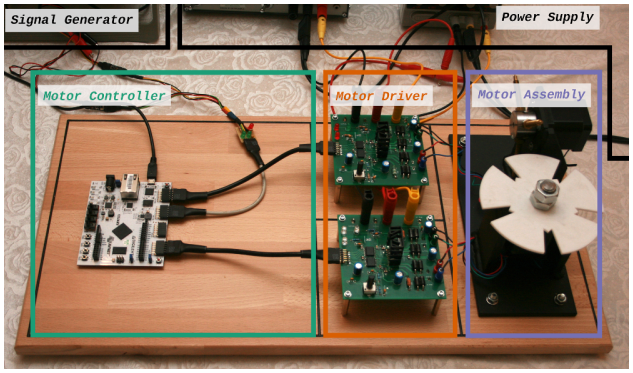

Figure 1 Demonstrator setup with annotations.

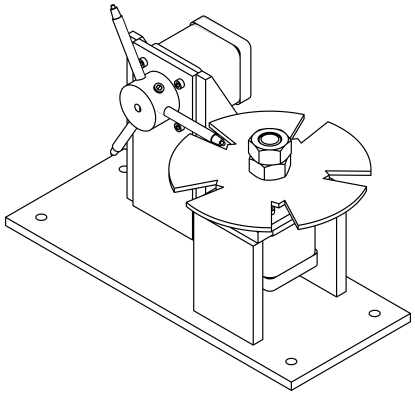

Figure 2 Technical drawing of the DS motor assembly.

real time application, which is where this paper comes in. The contributions and the organization of this paper are as follows.

- We describe a physical demonstrator, referred to as "disk and sticks demonstrator," or "DS demo" in short, that is reasonably cheap and easy to implement but embodies a hard real-time problem with scalable timing constraints (Sec. 2).
- We present a Timed SCChart model of a DS demo controller that illustrates the usage of dynamic ticks and clocks (Sec. 3).
- For the DS demo, we evaluate different controller platforms, comparing hardware (FPGA) and software alternatives, and evaluate the effect of dynamic ticks on reaction time, jitter and computational effort (Sec. 4).

We present the main aspects of the DS demo here. For more detailed information, we refer the reader to an extended report [5].

2 The Disk-and-Sticks Demonstrator

Figure 1 shows an annotated image of the DS demonstrator. While the name may suggest a link to storage devices, such as a hard disk, the DS demo is in fact based on two stepper motors that control the rotations of a disk and sticks.
On the left is the *motor controller*, in this case a Digilent Arty A7 35 FPGA board (Sec. 4 introduces a software alternative). It is connected to a 5V power supply and a signal generator to control the target speed of the system with a square wave input signal.
The main components of the *motor drivers*, in the middle, are two half-bridges to drive the coils of the stepper motors. An important feature of the motor drivers is the current sensing. If a motor draws too much current, an overcurrent signal is sent to the controller. This signal is used to implement an overcurrent protection that allows the motors to be operated with higher voltages without damage.
The *motor assembly*, on the right, contains two stepper motors arranged in a 90 degree angle. Figure 2 illustrates the detailed setup of the motor assembly. One motor has a disk and the other has three sticks mounted to its shaft. If the motors are running synchronized in an exact 3 to 5 ratio, they can pass through each other; any deviation from that ratio may cause a stick to hit the disk. Furthermore, the discrete nature of stepper motor control implies that timing errors typically do not lead to a gradual deviation from the desired speed, but may stop or even damage a motor.

3 Modeling a DS Controller with Timed SCCharts

Figure 3 shows the top-level SCChart running on the controller board. It uses a dataflow notation to connect various SCCharts modules handling different tasks. At the center is the SSD component, the speed signal divider, which converts the input speed into step instructions for each of the two motors in a strict 3 to 5 ratio to avoid collisions. The SSD component also receives inputs of the four buttons on the controller board that can be used to calibrate the initial positions of the disk and sticks. The preprocessing components (mc* and ed*) debounce and capture changes in the button signals. Each motor has its own MotorStateMachine that models the state of magnetization in the motor and allows to perform a step by (de-)activating the correct coils in the motor. In addition to the positive or negative magnetization of the coils, the motors need to be enabled to perform a step. These outputs are post-processed by an Ocp* module for each motor that handles overcurrent events and performs a timed cooldown for each coil.
Figure 4 shows the timed SCChart performing the overcurrent protection. The basic idea is to use the coil of the motor, the H-bridge and the protection diodes as a Buck converter where the continuous conduction mode keeps the coils always magnetized. The power is disconnected, via the enableOut output, for a constant time. During this *off time*, the coil is discharged through the protection diodes. The enableIn input triggers alternation between the Wait and Power states. When the motor is powered, an overcurrent event may occur, which triggers entering the Cooldown state and disabling the coil. However, this transition has an additional timing constraint that only allows this transition to be taken if at least a BLIND_TIME of

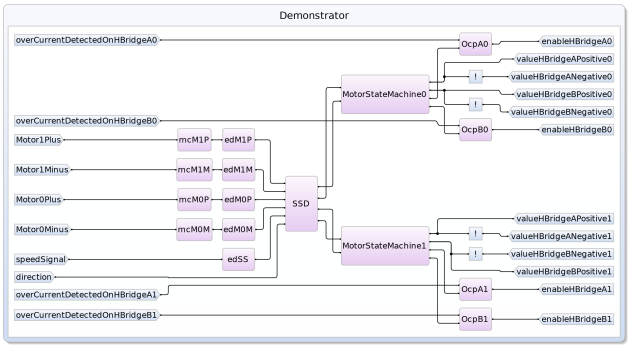

Figure 3 Top-level SCChart controller connecting multiple SCCharts modules handling sub tasks.

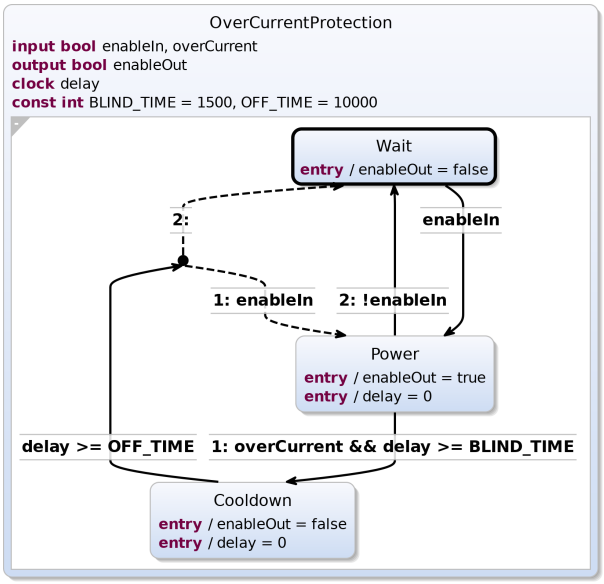

Figure 4 Timed SCChart to handle overcurrent events.

1500ns has passed since entering the Power state. This blind time has its origin in the parasitic induction of the resistor used to measure the current.

Using the clock construct, an extended feature provided by Timed SCCharts, this timeout is straightforward to express. The clock delay is reset when the Power state is entered, and is compared with BLIND_TIME in the predicate of the transition to Cooldown. When the H-bridge is activated, the coil is still charged and the current flowing through the coil tries to flow to the ground through the resistor. However, the parasitic inductance of the resistor is not yet charged and blocks the current flow. This leads to a brief voltage spike until the inductance is overcome. This voltage spike would unnecessarily trigger a transition into the Cooldown state and would repeat itself until the coil is completely discharged. The length of the blind time depends of the parasitic induction. When in the Cooldown state, the coil stays disabled for 10,000ns (OFF_TIME) and then returns either to Wait or Power, depending on Cooldown. This is again modeled with the delay clock. The off time has to be selected based on the speed of the H-bridge, overcurrent detection, the coil, and the possible current change during a PWM cycle.

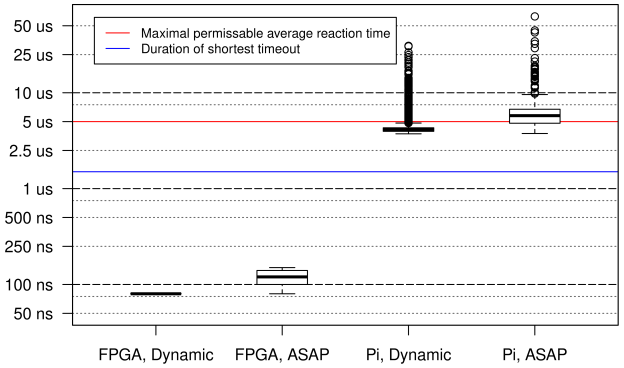

Figure 5 Comparison of reaction time between FPGA and Raspberry Pi with asap and dynamic tick environment (logarithmic scale).

4 Dynamic Ticks in HW and SW

As software-based alternative to the FPGA-based controller shown in Figure 1, an alternative controller board, based on a Raspberry Pi model 3B, connects to the same motor driver interface. KIELER [8] was used to compile the SCCharts controller into either VHDL, for the FPGA board, or C code, for the Raspberry Pi. The FPGA board itself serves as logic analyzer to capture the data. This approach has the disadvantage that the logic analyzer and the controller share the same 100MHz clock when measuring the FPGA. Hence, the FPGA-based motor controller has an advantage of up to 10ns in reaction time; these 10ns were added to the analysis results to display worst case behavior of the used implementation.

A first experiment measured the reaction time of the tick function at 400 steps per second and 5V motor supply voltage. The reaction time is measured as the delay between the input of the function generator and the resulting output change. Hence, it includes the tick calculation time and the offset caused by the environment. These quantitative evaluations were performed with for controlling one stepper motor in isolation, outside of the DS demo, to avoid possible damage to the demonstrator. The results are presented in Figure 5. The horizontal red line indicates the reaction time that allows safe operation with correct overcurrent protection. The dynamic tick environment on the FPGA has a constant reaction time, since there are no two events close enough together to influence each other. The asap environment, on the other hand, has a variance from 1 to 2 calculation times. This is because the events can occur at any time, even during an active tick calculation, adding the remaining calculation time to the reaction time. The Raspberry Pi controller performs worse, as expected for a general purpose processor with an Linux operating system. Despite that, the most obvious difference to the FPGA-based controller are the outliers in the reaction time. These outliers are caused by the kernel and have a calculation time that is up to 10 times bigger than their average. In the asap environment the variance is again higher, since events usually occur during the tick calculation. A real-time kernel and isolation of the controller process on a

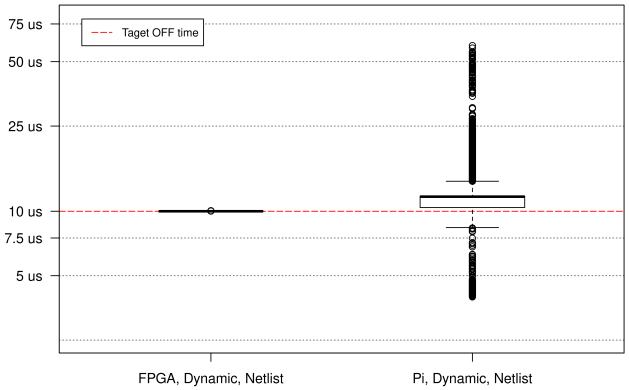

Figure 6 Comparison of jitter in the actual off time between the between FPGA and Raspberry Pi with dynamic ticks (logarithmic scale).

single core are measures that would reduce the number of outliers, but cannot remove them.

The dynamic and asap operation modes also differ significantly in the number of ticks computed, since asap triggers "superfluous" reactions without output change. At one rotation per second, corresponding to 400 steps per second, the asap mode executes about 1.25×10^7 ticks/sec on the FPGA, and, depending on the synthesis approach, between 65,000 and 260,000 ticks/sec on the Raspberry Pi. The dynamic mode requires about 1000 ticks/sec on both platforms.

In a second experiment we measured the timing precision of the overcurrent protection by capturing the actual off time produced by the Cooldown state. The results are plotted in Figure 6. This test is performed by setting the motor speed to 0 in a motor state that powers both coils, with a supply voltage of 10V and a current limit at 0.5A. Hence the overcurrent protection of both coils constantly triggers. The length and variance of the off times are used to measure the reaction time jitter. The FPGA timing is perfect with the exception of a few outliers. The maximal outliers are less than one tick calculation time bigger than the expected value. These outliers are created by an overcurrent event that is less than one tick calculation prior to the timing event. The results of the Raspberry Pi show outliers that are too long, similar to the previous test, and outliers with off times too small. These early reactions are indirectly created by the slow outliers. If a long tick calculation time puts calculations behind the real-time, the calculations try to catch up with the real-time, resulting in shortened off-times.

5 Wrap-Up

The DS demo has achieved its objective of being a clear demonstration of reactive system programming with a synchronous language, in this case Timed SCCharts, and the potential merits of using dynamic ticks. With the FPGA-based controller, the DS demo was operated successfully with up to 100 stick/disk crossings per second. This corresponds to 1200 RPM for the disk. With 400 steps per rotation, and at least 10 ticks per step due to the sampling/synchronization logic, this corresponds to 80,000 ticks per second, or 12.5 μs per tick. At this speed, each tick requires stable and precise timing in the microsecond range. The dynamic tick environment facilitates such precise reactions and the FPGA provides a fast and jitter-free execution platform. The Raspberry Pi, as expected, has a lower performance and occasionally misses steps due to outliers in the reaction time. These interruptions are not only limiting the maximal possible RPM, but are already audible at lower speeds.

There are several avenues for future work, including improved hardware synthesis that allows higher clock rates and software solutions with reduced operating system disturbances.

6 Literature

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, Jan. 2003. IEEE.

[3] T. Bourke and M. Pouzet. Zélus: a synchronous language with odes. In *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013*, pages 113–118, Philadelphia, PA, USA, Apr. 2013.

[4] T. Bourke and A. Sowmya. Delays in Esterel. In *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*, number 09481 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 Nov. 2009.

[5] A. Boysen, A. Schulz-Rosengarten, and R. von Hanxleden. An FPGA-based Demonstrator for Dynamic Ticks. Technical Report 2001, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2020.

[6] J. Colaço, B. Pagano, and M. Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering TASE*, pages 1–11, Sophia Antipolis, France, Sept. 2017.

[7] A. Schulz-Rosengarten, R. von Hanxleden, F. Mallet, R. de Simone, and J. Deantoni. Time in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL '18)*, Munich, Germany, Sept. 2018.

[8] S. Smyth, A. Schulz-Rosengarten, and R. von Hanxleden. Towards interactive compilation models. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, volume 11244 of *LNCS*, pages 246–260, Limassol, Cyprus, Nov. 2018. Springer.

[9] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

[10] R. von Hanxleden, T. Bourke, and A. Girault. Real-time ticks for synchronous programming. In *Proc. Forum on Specification and Design Languages (FDL '17)*, Verona, Italy, Sept. 2017.