# A Hard Real Time Demonstrator for Dynamic Ticks and Timed SCCharts

Andreas Boysen, Alexander Schulz-Rosengarten, Reinhard von Hanxleden
*Department of Computer Science, 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.**

***Index Terms*—Real-time systems, reactive systems, synchronous languages, dynamic ticks, FPGA**

## I. 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 constructive logic gives a well-founded, deterministic semantics to circuits that abstracts from their physical implementation and actual stabilization delays. Classical synchronous languages include Esterel [3], Lustre [10] and Signal [2]; more recent languages include the hybrid modeling language Zélus [4] and the statechart dialect SCCharts [18], which now is used in the railway domain; commercially most successful at this point is probably SCADE [7], with its qualified compiler that is routinely used by Airbus and other industrial players.

978-1-7281-8928-4/20/$31.00 ©2020 IEEE

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 [5]. 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 [17], 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* [17] give the synchronous program control not only about how it reacts to current and past inputs, but also *when* the next reaction should occur. The original 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 [14], 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,

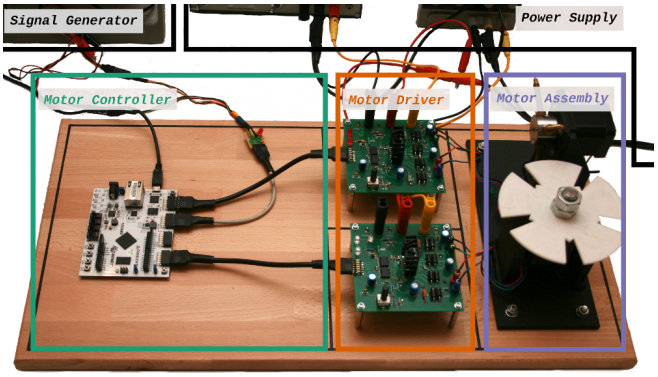



Fig. 1. Demonstrator setup with annotations.

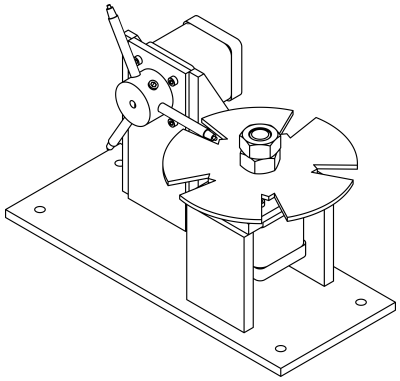


Fig. 2. Technical drawing of the DS motor assembly.

and a demonstrator with a hard real-time application, which is where this paper comes in.

*Contributions and Outline*

- 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. II).
- We present a Timed SCChart model of a DS demo controller that illustrates the usage of dynamic ticks and clocks (Sec. III).
- We describe how a dynamic tick environment can be implemented in hardware and software (Sec. IV).
- 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. V).

We briefly discuss further related work in Sec. VI and conclude in Sec. VII.

We present the main aspects of the DS demo here. More detailed information, e. g. on the hardware, can be found in an extended report [6].

## II. The Disk-and-Sticks Demonstrator

Fig. 1 shows an annotated image of the DS demonstrator. While the name may suggest a link to storage devices, such as hard disks and USB sticks, 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[1] (Sec. IV 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 PMOD headers are used to connect the controller to the two motor drivers in the middle.

The main components of the *motor drivers* are two H-bridges to drive the coils of the stepper motors. The motors consume significant power and require a high voltage, especially for high speeds, hence they cannot be powered by the controller directly. The two H-bridges on each motor driver convert the digital low power control signal into high power signals for each coil of the stepper motor. The driver board is designed to have easily accessible measurement points for benchmarks and includes a galvanic isolation of the controller and the half-bridges components, to protect the more expensive controller board from potential failures and the high voltages in the motor driver.

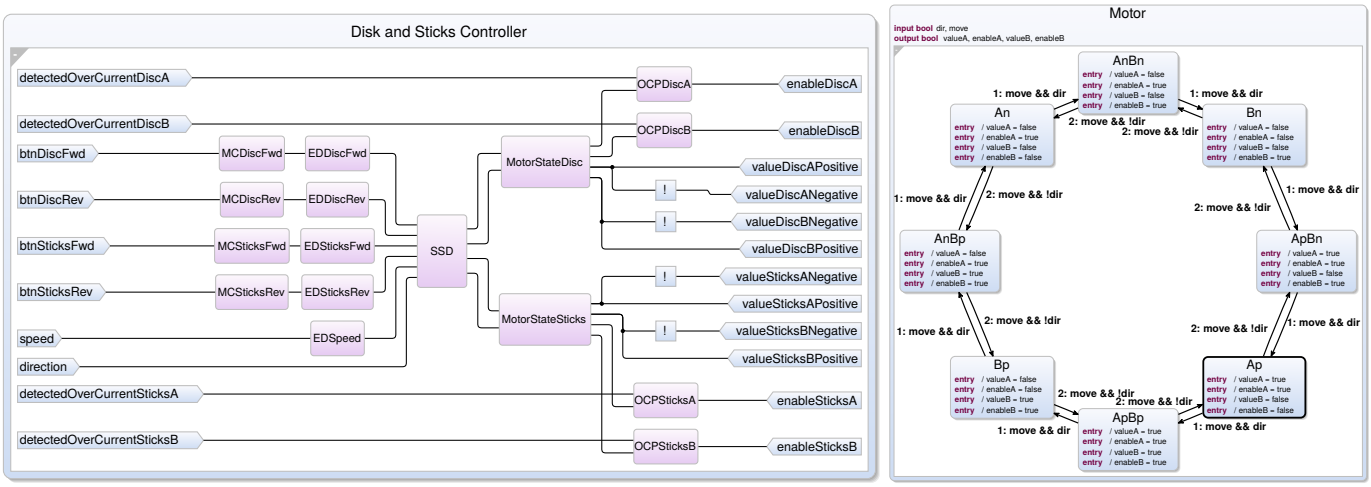
An important feature of the motor drivers is the current sensing. The driver board reads the current drawn by the motor and compares it with a reference value. If a motor draws too much current, an overcurrent signal is sent back to the controller. This signal is used to implement an overcurrent protection 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. Fig. 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, the sticks can pass through the disk; 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 the motor completely.

The demonstrator uses stepper motors, as these are cheap and flexible motors often used in industry and they provide good repeatability, considering the step locations they lock into. Most importantly, when controlling a stepper motor directly, the steps in the real world directly correspond to reactions in software. As a consequence, there are high real-time requirements on the software and, in our example, on the efficiency and coordination of ticks, as described in Sec. IV.

In general, a stepper motor creates a rotating magnetic field by magnetizing coils to move the shaft. The demonstrator uses two-phase bi-polar hybrid stepper motors. This type uses a permanently magnetized rotor, surrounded by a stator containing two separately controllable coil sets. Both the stator and rotor are multi-toothed, and energizing a set of coils

[1] https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual




(a) Top-level SCChart controller connecting multiple SCCharts modules handling sub tasks.

(b) Motor controller SCChart.

Fig. 3. SCCharts models for the top-level controller and one of its submodules.

will cause the rotor to snap into the next position (step) by aligning itself to the magnetic fields. Furthermore, the motors in this demonstrator are driven in half step operation mode to increase the resolution, such that the motion is smoothened and vibrations are reduced. Consequently, this doubles the number of control signals to perform a full rotation and imposes higher reaction speeds on the controller, compared to a full step mode. A single (half) step is performed by (de-)energizing the appropriate coils but the electromagnetic magnetization is affected by the inductance of the coil. It takes some time to increase the current flowing through the winding after voltage is applied to it. To speed up this process and thus the maximal possible rotational speed of the motor, it is common to apply higher voltages. The stepper motors used in this demonstrator have a nominal voltage of 4.2V but can easily reach about 11,000 rounds per minute when powered with 30V. However, increasing the voltage also increases the current drawn by the coil when powered over time and this will destroy the coil. The nominal voltage is a safe value that will not harm the motor in a steady state with permanently energized coils. Hence, to safely operate the motor at higher speeds with higher voltages, an overcurrent protection is needed that decreases the supply voltage to a coil for some time, to limit the resulting current. This could be done directly in hardware on the motor driver board, but we decided to only sense the overcurrent event there and to feed it back to the controller, to create a critical hard real-time environment for benchmarking the control software. Then the controller can disable the half bridge for a short period, such that the coil in the motor is partially discharged via protection diodes. Our setup uses the principle of a Buck converter in a continuous conduction mode and keeps the coils always magnetized, without exceeding the current limit.

The modular structure of the demonstrator allows to connect different controllers to this setup. As a software-based alternative to the FPGA-based controller shown in Fig. 1, we also use a Raspberry Pi model 3B as controller. It requires a simple converter board to connect the GPIO pins with PMOD jacks for the driver board cables. Sec. IV presents the results of a comparison of these two controller platforms.

## III. Modeling a DS Controller with Timed SCCharts

The controller running the DS demonstrator is modeled using Timed SCCharts. SCCharts [18] is a statecharts dialect and, as such, it primarily consists of states connected via transitions. Due to its synchronous nature, SCCharts have delayed transitions (represented by solid lines) that require to stay for at least one tick in a state before it can be left via this kind of transition, and immediate transitions (dashed lines) that can be taken instantaneously. Using regions, SCCharts can be nested hierarchically and composed in parallel, while the *sequentially constructive* semantics ensures determinism for concurrently shared variables. Each SCChart can declare variables, including those for inputs and outputs to communicate with its environment or another SCChart it is embedded into. SCCharts have a large set of extended features, which can be considered syntactic sugar, since they are usually replaced by simpler but larger constructs during compilation. For example, *entry actions* will be executed immediately when a state is entered. At its core, SCCharts is a control-flow-oriented statechart but it also supports a hybrid design where some aspects are expressed as data-flow diagrams. Another recently added feature are Timed SCCharts [14], that introduce clocks that progress with the real time and allow time-based transition triggers. The concept is based on timed automata [1] in which clocks are continuous variables that represent time. In SCCharts, the progress of time in variables of type clock is handled automatically and the modeler can read and set their value, to track time and react to timing events.

Fig. 3a shows the top-level SCChart of the DS controller. 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 MotorState that models the state of magnetization in the motor and allows to perform a step by (de-)activating the correct coils in the motor. Fig. 3a illustrates the corresponding SCChart, where each state corresponds to a combination of positive, negative, or disabled state of each coil. Note that the variables for the two H-bridges in this SCChart (*A and *B) will be bound accordingly when instantiated by the top-level SCChart. In addition to the positive or negative magnetization of the coils, the H-bridge also needs to be enabled to perform a step. These enabling outputs are post-processed by an OCP* module for each H-bridge that handles overcurrent events and performs a timed cooldown for the coils.

Fig. 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 SCChart has three states: Wait, Power, and Cooldown. The enableIn input triggers alternation between the Wait and Power states in normal operation mode. 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 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, 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. The blind time has to be much smaller than the off time, otherwise there would be a risk of a run away situation in which the current decrease during the cooldown is smaller than the increase during the blind time. 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.

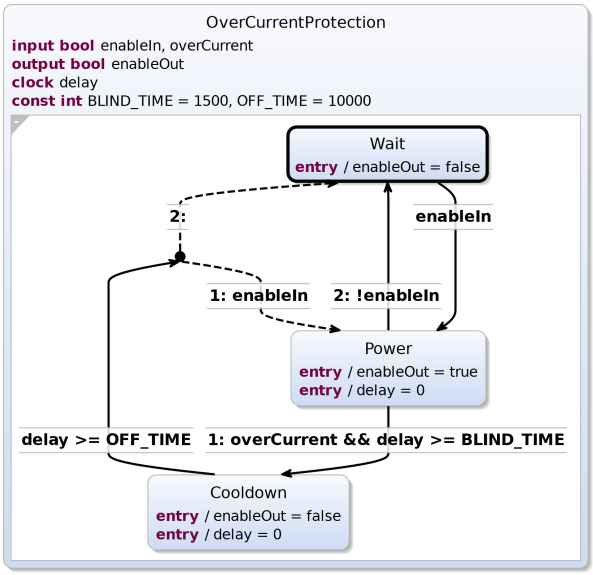



Fig. 4. Timed SCChart to handle overcurrent events.

```
1  TickData data;
2  reset(&data);
3  long preT = currentTime();
4  while (!data.TERM) {
5    long curT = currentTime();
6    data.deltaT = curT − preT;
7    preT = curT;
8    readInputs(&data);
9    tick(&data);
10   writeOutputs(&data);
11
12   // This is omitted for asap execution
13   waitForInputOrTimeout(data.sleepT, preT);
14 }
```

Fig. 5. Simplified structure of a tick environment loop for dynamic/asap execution.

With dynamic ticks, each active state that has a timed transition computes a sleep time that is the remaining time until its timed trigger will be enabled. The overall sleep time of the program is the minimum of all these sleep times and some large default value, e. g. 100 msec [14].

## IV. A Dynamic Tick Environment in HW and SW

The concept of dynamic ticks [17] gives the program access to real time and allows to delay its own logical steps based on time. Timed SCCharts implement this concept and provide high level clock constructs based on timed automata as presented in the previous section. However, even if this enables the SCCharts to influence its own pace, the answer to the main question, “when to react?” (see [14]), still depends on the execution environment.

### A. The Tick Function

Code syntheses for synchronous languages usually generate a tick function that receives the read inputs, computes a single reaction (tick), and produces the outputs. The most simple and common mode of execution is invoking this tick function in a loop as soon as possible (asap) or in a constant rate. Fig. 5 illustrates the general structure of such an asap environment in C. TickData is a struct generated by the SCCharts compilation to hold the variables for inputs, outputs and the internal state. The reset and tick function are likewise synthesized from the

SCChart and initialize/reset the internal state and compute a reaction respectively. The basic structure of this environment is to read the inputs from the hardware, store it in the TickData, invoke the tick function, send the outputs to the hardware and repeat this sequence immediately as long as the program does not indicate its termination. To enable the use of clocks in SCCharts, the asap mode also needs to compute the time passed between invocations of the tick function and store it in the deltaT input (lines 5 to 7).

In a dynamic tick environment, the only but significant difference is that the environment will wait after a tick (line 13). To provide the *eager semantics* [14] necessary for dynamic ticks, both the timeout requested by the SCChart (sleepT output) and input events must be able to start the next tick cycle. This waiting approach requires polling inputs for changes or awaiting interrupts, as well as a timeout mechanism. Compared to the asap approach that reads and processes inputs continuously, dynamic ticks actually allow a faster and more precise reaction to input events, as the evaluation in Sec. V will show.

### B. Managing Time

An important aspect when creating timeouts based on sleepT is to consider the computation time. The tick function holds a synchronous program, hence, it assumes based on the synchronous paradigm that outputs are produced at the same time as inputs are read. In reality, this is infeasible but it means that the sleepT time is also computed based on the deltaT input without considering the time passed after the deltaT sample was taken. Consequently, when setting up a timeout after the tick function call, the time passed since starting of the tick (prevTime in Fig. 5) must be deducted from the sleep time. However, this may result in situations where the requested sleep time has already passed due to a long computation time. This is a clear sign that the platform or the program does not perform well enough to fulfill the timing constrains. If the platform is fixed, a timing analysis of the program can reveal bottlenecks. For example, SCCharts support an interactive timing analysis approach that is incorporated into the graphical modeling process [9]. In Sec. V we also evaluate the influence of the code generation of the performance of the program. Clearly, on an FPGA platform, variation in the computation time or a delayed handling of timeouts is not an issue, but on a software-oriented general purpose processor with limited real-time capabilities, it can have a significant effect on the reaction time.

With dynamic ticks, there are different possibilities to handle timing deviations. Either the program is provided with a simulated time where deltaT is always equal to sleepT if no input events occurs during sleep time, or deltaT is the real time passed between ticks. We prefer the latter, as this enables the synchronous program to transparently and deterministically react to timing deviations without assumptions on the environment. Schulz-Rosengarten et al. [14] present different options to react to imprecisions and deviations in the SCCharts model.

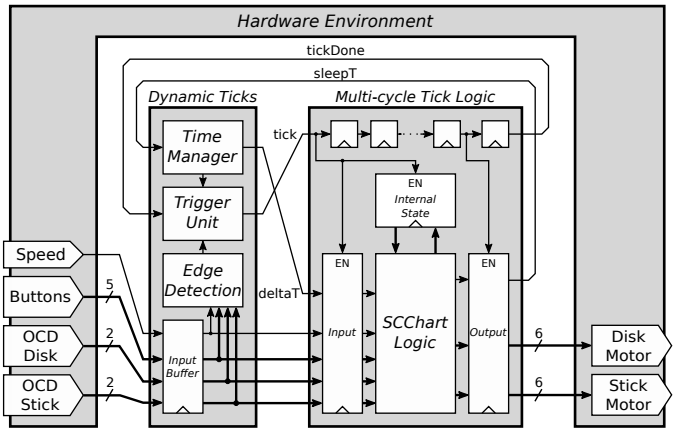



Fig. 6. Implementation of a dynamic tick environment in the DS controller.

### C. Dynamic Ticks for the DS Demonstrator

Fig. 6 illustrates the dynamic tick environment setup implemented for the DS controller interface. The general setup, as presented by Schulz-Rosengarten et al. [14], defines that a Time Manager communicates with the logic via deltaT and sleepT variables. A Trigger Unit starts ticks based on input and timing events. In Fig. 5 the Time Manager is represented by lines 3 and 5 to 7 and the Trigger Unit is implemented by the delay in waitForInputOrTimeout in line 13 or its absence in the asap mode. In the DS implementation these components are also present but in their abstracted form. Additionally, there is a tickDone signal used by the Trigger Unit that enables the start of a new tick. It is part of the multi-cycle tick logic that will be discussed later. The Hardware Environment provides the raw inputs, here, the 4 buttons and the direction switch, the square wave speed signal, and the over-current events of the two H-bridges for each motor. These inputs are first buffered in the Dynamic Ticks handler and then processed by an Edge Detection such that value changes trigger a reaction. The tick signal indicates the start of a new computation. Inputs are copied into the registers of the logic, and when the computation finishes, the outputs are sent to the two motor controllers and the sleep time is fed back to the Time Manager. As presented in Fig. 3a each motor has a positive, negative, and enabling value for each of its H-bridges.

While Fig. 5 illustrates an imperative view on tick environments, Fig. 6 implements a multi-cycle logic environment for the invocation of the tick function that is specific for the use in hardware. In general, synchronous languages are well fitted for a hardware synthesis, as they both use a synchronous clock to drive computations (ticks). SCCharts provide a netlist-based compilation approach [18] that is able to produce VHDL code directly. For an asap or fixed periodic environment it is sufficient to simply wire the inputs and outputs to the hardware interfaces and to configure the clock of the FPGA to drive the logic at a fixed rate. However, for dynamic ticks this undermines the efficiency and precision gained by reacting exactly when it is necessary because the pace during waiting would be bound to the minimal rate of the SCChart's logic,

i. e. the longest path through the generated netlist. Hence, to circumvent this problem, our dynamic tick environment in VHDL uncouples the ticks of the SCChart from the clock of the FPGA and allows the tick computation to spread across multiple clock cycles.

### D. Multi-Cycle Tick Logic

In the multi-cycle logic environment in Fig. 6, the stateless SCCharts Logic is surrounded by various registers. Inputs are held stable during compilation by the registers on the left. They are clocked by the FPGA's base clock but only enabled (EN) when the tick signal is given to start the computation of a tick. The same holds for the registers that store the internal state. The registers for the outputs are enabled when the computation finishes, which is determined by the chain of registers at the top. The tick signal ripples through these registers until the computation is finished and tickDone is signaled.

The number of registers can be adjusted at compile time and allows to set the FPGA's clock at a respective fraction of the logic's computation time. With such a setup, the environment can wait more precisely for timeouts and sample inputs in a finer granularity before triggering the multi-cycle tick computation. Furthermore, since steps in a synchronous program are strictly ordered, a new tick may only be started when the previous reaction is finished, indicated by tickDone, otherwise the internal state and outputs will be corrupted.

### E. Assessment

Dynamic ticks enable the handling of real time and a dynamic reaction pace while retaining the sound semantics and determinism of the synchronous languages. In fact, the actual semantics is not affected at all, since only the environment is adjusted, as illustrated in Fig. 6. deltaT and sleepT simply extend the input output interface of the program similar to the multiform notion of time but without its inconsistencies when it comes to different abstraction of timing events. Regarding verification, a real time input, of course, increases the complexity, however, with SCCharts clocks based on timed automata, there are various approaches and tools to tackle this issue, e. g. as implemented in the Kronos tool [8].

## V. Evaluating Dynamic Ticks using the DS Demonstrator

The DS demonstrator offers a realistic environment based on common hardware with scalable timing requirements to evaluate dynamic ticks. In addition to the FPGA-based controller shown in Fig. 1, the demonstrator can also be controlled by an Raspberry Pi. This allows to compare the performance of Timed SCCharts on a very specialized FPGA with a general purpose ARM processor. In this evaluation the KIELER tool [16] 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.

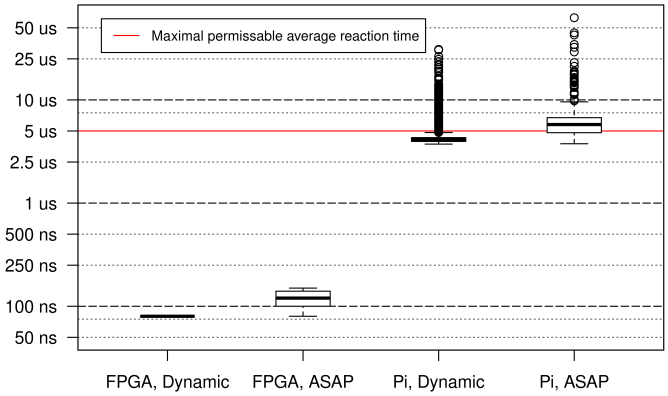



Fig. 7. Comparison of reaction time between FPGA and Raspberry Pi with asap and dynamic tick environment (logarithmic scale).

### A. Reaction Times

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 for controlling one stepper motor in isolation, outside of the DS demo, to avoid possible damage to the demonstrator. The results are presented in Fig. 7. The horizontal red line indicates the reaction time that allows safe operation with correct overcurrent protection ($5\ \mu s$).

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 a 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 single core would reduce the number of outliers, but cannot remove them.

### B. Tick Counts

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 $1.25 \times 10^7$ ticks/sec on the FPGA, and 260,000 ticks/sec on the Raspberry Pi. The dynamic mode

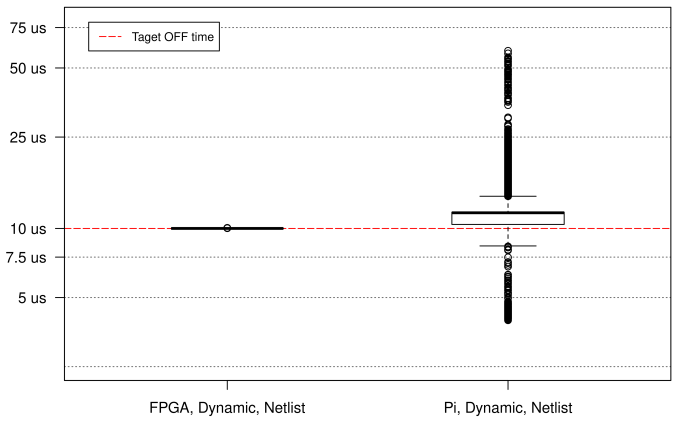



Fig. 8. Comparison of jitter in the actual off time between the between FPGA and Raspberry Pi with dynamic ticks (logarithmic scale).

requires about 1000 ticks/sec on both platforms, as only relevant inputs and points in time are processed.

### C. Timing Precision

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 Fig. 8. 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 needs to be constantly active. 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.

The results show that dynamic ticks not only reduce the computation effort compared to an asap mode but also allow more accurate reactions, since it is less likely that a tick computation is running while an important event, such as overcurrent, occurs. As expected, the FPGA controller performs well, but the experiments using the Raspberry Pi illustrate the limitations of a general purpose platform in such a scenario.

### D. Evaluating Compilation Approaches

The netlist approach is the most fitting compilation strategy for an FPGA controller and consequently it is only fair to use the same on the Raspberry Pi. However, SCCharts also support alternative, more software focused compilation approaches, which were tested using the DS demo. The *priority-based approach* [18] separates the program into multiple partitions that are executed and scheduled by a lightweight runtime scheduler using static priorities for concurrent parts. The *state-based approach* [15] stays close to the statemachine structure in SCCharts and generates functions for each state and region that are invoked based on active states and control flow.

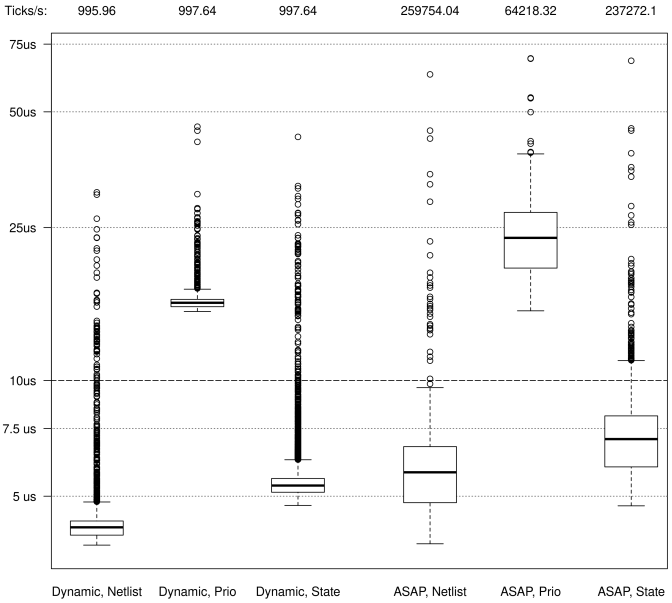



Fig. 9. Comparison of different compilation approaches for SCCharts with dynamic ticks and asap environment on the Raspberry Pi (logarithmic scale).

Fig. 9 shows the results of all compilation approaches for both the asap and the dynamic tick environment on the Raspberry Pi. The test setup is the same as in the first experiment (Fig. 7). Again, the asap mode leads to greater deviations in the reaction time and performs far more ticks per second, as the numbers on top of the diagram indicate. The netlist-based approach performs best. This might come from its simple structure and many optimizations that are implemented for this approach in the KIELER tool. The priority-based approach performs worst, despite the fact that it is primarily designed for a software platform. It uses a rather minimal runtime scheduler implemented in C macros with computed gotos, but it seems this still generates an overhead that puts this approach behind the others. Usually, it stands out in supporting loops and large programs where only a small portion is actually executed per tick. Both properties do not hold for this controller. The rather new state-based approach performs quite well and is only a few milliseconds behind netlist. We think these results are remarkable, considering that the primary focus of this approach is on readability and manual verification of the generated code. It seems the compiler is able to handle the extensive use of functions and function calls, probably via inlining, and produces a relatively efficient executable, in comparison to netlist. Thus we intend further to develop and improve this approach in the future.

## VI. Related Work

There are many languages, also outside the domain of synchronous languages, that fit well to model a controller for

the DS demonstrator, for example LabVIEW [19]. However, the main focus here is to evaluate dynamic ticks that are designed for synchronous languages. Besides SCCharts, which already provide both an integration of dynamic ticks and a VHDL synthesis, SCADE [7], with Lustre [10] at its core, would be a very capable alternative for modeling the controller. SCADE provides a certified code generator and is used in the industry to model safety critical embedded applications. Additionally, there has been work by Pampagnin and Letellier on a VHDL synthesis from SCADE in an avionics context [12]. Pampagnin et al. also developed a model-based approach dedicated to avionics hardware design [13]. Even if they follow a methodology that is based on a multi model approach to allow different abstraction levels, they define a development process and tool that is relatively similar to SCCharts in KIELER.

Regarding the demonstrator hardware, there are various ways to create a challenging real-time environment. We decided to use an FPGA and stepper motors as these are of-the-shelf hardware widely used in industry and enable proper performance benchmarks [11]. Bourke and Sowmya used a microprinter controller equipped with a stepper motor as their motivating example for illustrating delays in Esterel [5]. This, among others, inspired the development of dynamic ticks. Hence we wanted to evaluate this approach in a similar scenario.

## VII. Conclusions and Outlook

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 $\mu$s 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 and performance improvements based on optimizations of the generated code.

## References


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

[2] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.

[3] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[4] Timothy Bourke and Marc 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, April 2013.

[5] Timothy Bourke and Arcot 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 November 2009.

[6] Andreas Boysen, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. An FPGA-based Demonstrator for Dynamic Ticks. Technical report, Christian-Albrechts-Universität zu Kiel, Department of Computer Science.

[7] Jean-Louis Colaço, Bruno Pagano, and Marc 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, September 2017.

[8] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool kronos. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, pages 208–219. Springer Berlin Heidelberg, 1996.

[9] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. Time for reactive system modeling: Interactive timing analysis with hotspot highlighting. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS '16, pages 289–298, New York, NY, USA, 2016. ACM.

[10] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[11] Ngoc Quy Le and Jae Wook Jeon. An open-loop stepper motor driver based on FPGA. In *2007 International Conference on Control, Automation and Systems*, pages 1322–1326, 2007.

[12] Pascal Pampagnin and Ludovic Letellier. The GENCOD project : Automated generation of Hardware code for safety critical applications on FPGA targets. In *ERTS2 2010, Embedded Real Time Software & Systems*, Toulouse, France, May 2010.

[13] Pascal Pampagnin, Pierre Moreau, Remy Maurice, and David Guihal. Model driven hardware design: One step forward to cope with the aerospace industry needs. In *Proc. Forum on Specification and Design Languages (FDL '08)*, pages 179–184, 2008.

[14] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. Time in SCCharts. In Tom J. Kazmierski, Sebastian Steinhorst, and Daniel Große, editors, *Languages, Design Methods, and Tools for Electronic System Design: Selected Contributions from FDL 2018*, pages 1–25. Springer, 2020.

[15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. Synthesizing manually verifiable code for statecharts. In *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Boston, MA, USA, November 2018.

[16] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard 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, November 2018. Springer.

[17] Reinhard von Hanxleden, Timothy Bourke, and Alain Girault. Real-time ticks for synchronous programming. In *Proc. Forum on Specification and Design Languages (FDL '17)*, Verona, Italy, September 2017.

[18] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen 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.

[19] Guoqiang Wang, Trung N. Tran, and Hugo A. Andrade. A graphical programming and design environment for FPGA-based hardware. In *2010 International Conference on Field-Programmable Technology*, pages 337–340, 2010.