INSTITUT FÜR INFORMATIK

Time in SCCharts

Alexander Schulz-Rosengarten,
Reinhard von Hanxleden, Frédéric Mallet,
Robert de Simone, Julien Deantoni

Bericht Nr. 1805
July 2018
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Department of Computer Science
Kiel University
Olshausenstr. 40
24098 Kiel, Germany

Time in SCCharts

Alexander Schulz-Rosengarten, Reinhard von Hanxleden,
Frédéric Mallet, Robert de Simone, Julien Deantoni

Report No. 1805
July 2018
ISSN 2192-6247

E-mails:
{als, rvh}@informatik.uni-kiel.de
{frederic.mallet, robert.de_simone, julien.deantoni}@inria.fr

This is an extended version of a paper submitted to the
Forum on specification & Design Languages (FDL) 2018.

Abstract

Synchronous languages, such as the recently proposed SCCharts language, have been designed for the rigorous specification of real-time systems. Their sound semantics, which builds on an abstraction from physical execution time, make these languages appealing, in particular for safety-critical systems. However, they traditionally lack built-in support for physical time. This makes it rather cumbersome to express things like time-outs or periodic executions within the language.

We here propose several mechanisms to reconcile the synchronous paradigm with physical time. Specifically, we propose extensions to the SCCharts language to express clocks and execution periods within the model. We draw on several sources, in particular *timed automata*, the Clock Constraint Specification Language, and the recently proposed concept of *dynamic ticks*. We illustrate how these extensions can be mapped to the SCChart language core, with minimal requirements on the run-time system, and we argue that the same concepts could be applied to other synchronous languages such as Esterel, Lustre or SCADE.

Keywords: real-time systems, reactive systems, synchronous languages, timed automata, timing specification, dynamic ticks

This is an extended version of a paper submitted to the Forum on specification & Design Languages (FDL) 2018.

Contents

1 Introduction

Cyber-physical/embedded systems are typically *reactive*, meaning that they have to continuously react to their environment, and that these reactions must meet certain timing contraints. Real-time aspects may be rather simple, such as "the system must run at least at 10 KHz," or it may be quite intricate, like "coil A must be activated 27.3 msec after coil B." A long-standing challenge in the design of such *real-time systems* is to reconcile concurrency and determinacy. As it turns out, time there plays a rather adversarial role in that standard mechanisms to handle concurrency, such as Java/Posix *threads*, are rather sensitive to how long individual computations take; determinacy is easily compromised by *race conditions* [17]. Synchronous languages address this challenge by abstracting from execution time; their semantics rests on the assumption that computations take zero time, and that outputs are synchronous with their inputs [4]. The synchronous programming paradigm has been explored since the 1980s, and for example SCADE (Safety-Critical Application Development Environment) and its certified code generator are routinely used for avionics control software [8].

The abstraction from time in synchronous languages typically comes at the price that all references to physical time must somehow be resolved by the environment. Unlike for example Harel's statecharts [13], which already included a mechanism to express timeouts, physical time is traditionally not a first-class citizen in synchronous languages; they instead build on a *multi-form notion of time*, where time is expressed by counting events (detailed further in Sec. 3.3). This is consistent with the synchronous abstraction, but in practice does not help the programmer, who at the end of the day must express the required real-time behavior.

In this paper, we investigate how we can incorporate physical time into the synchronous model of computation. We do so using the SCCharts language [25], however, the concepts presented here can be applied to other synchronous languages as well.

1.1 Contributions and Outline

- We show how timed automata, which model time with real-valued clocks, can be expressed in a synchronous setting with discretized execution (Chap. 2). Our proposal, which includes a new type clock for SCCharts, uses on-board mechanisms of synchronous languages (in particular *during actions*) to faithfully model clocks and imposes minimal requirements on the execution environment.
- We investigate the suitability of different execution regimes in a timed setting (Chap. 3) and argue that *dynamic ticks* [24] are a natural fit for realizing timed automata.

- We present an approach to implement dynamic ticks in a synchronous setting, where the compiler deduces anticipated tick durations from timing constraints in the model (Chap. 4).
- We propose a language extension of SCCharts (period) that allows to model multi-clocked systems based on periodical activation of different subsystems and that maps naturally to real-valued clocks (Chap. 5).
- Finally, having clocks as first-class citizens we use them to map one abstract clock constraint, expressed in the Clock Contraint Specification Language (CCSL), to SCCharts (Chap. 6). This allows to not only relate the activation of subsystems to physical time, but also to the activation of other subsystems.

We briefly discuss further related work in Chap. 7 and conclude in Chap. 8.

2 Timed Automata in SCCharts

Timed automata, proposed by Alur and Dill [2], are a formalism to model the behavior of real-time systems over time. Timed automata consist of state-transition graphs with timing constraints using real-valued *clocks*. A timed automaton accepts *timed words*, which are (infinite) sequences in which a real-valued time of occurrence is associated with each element of the timed word.

Timed automata and their variations have been extensively studied for verification purposes [2, 1, 20]. We here want to use them for synthesis purposes as well. That is, we investigate how to model the behavior of real-time systems such that the model can also be synthesized into a piece of software or hardware.

Timed automata have been extended in various ways, one example are *multirate timed automata* (or *multirate timed systems*) [1], where each clock has its own speed, possibly varying between a lower and an upper bound. Lee and Seshia [18] discuss (multirate) timed automata in the context of cyber-physical system design. One of their illustrating examples is the traffic light controller introduced in the next section.

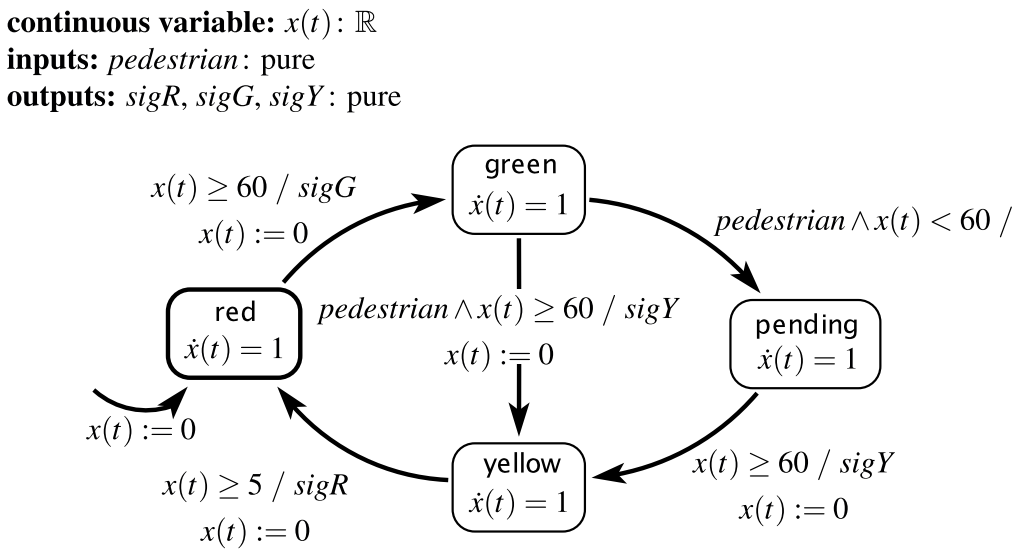


Figure 2.1: Trafficlight controller modeled as timed automaton. From Lee and Seshia [18] (CC BY-NC-ND 4.0).

2.1 The Traffic Light Controller Example

We use the traffic light controller shown in Fig. 2.1 as running example. The traffic light has three lights green, yellow, and red to control the car traffic and a button for a pedestrian to request secure crossing of the street, which should cause the traffic light to switch temporarily to a red light to stop the traffic. The automaton of the controller has a real-valued clock x, an input pedestrian indicating whether a pedestrian requests crossing the street, and three outputs sigR, sigG, sigY. The type pure denotes "pure signals" present or absent at each reaction and carrying no further data. The outputs do not directly indicate the light *states*, but rather constitute *events* that indicate color changes. It is assumed that initially the red light is turned on; emitting the event sigG switches off red and switches on green, and so on.

As shown in this example, a clock is represented by a first-order differential equation on a real number and can be explicitly set and used as transition guard. While in state red, time progresses with a slope of one ($\dot{x}(t) = 1$), this is also the case for all other states. Time is expressed in abstract time units; for simplicity, we assume for this example that one time unit corresponds to one second. Each transition has a *guard*, which consists of a *condition* (such as pedestrian) and a *timing constraint* (such as $\mathsf{x} \geq 60$), both of which are optional. When clock x has reached or surpassed 60, the automaton transitions to green emitting the green light and resetting the time to zero. Now the system waits for a pedestrian to push the button. When the pedestrian input is present, the reaction depends on the passed time. Case 1, if less than 60 sec passed since entering green, the automaton will transition to pending, but x is not reset. It remains there until the time has reached at least 60, then the yellow light is turned on, the timer is reset and the state is switched to yellow. Case 2, if the pedestrian event occurs after at least 60 sec in green, the automaton transitions directly to yellow with the same output and reset. After at least 5 sec, the automaton leaves the yellow state for red and activates the red light and again resets the time.

2.2 From Specification to Behavior—the Eager Semantics

Even though this traffic light controller specification seems rather clear and straightforward, it turns out that there is still some variation as to how the controller may behave in a specific scenario. The original definition of timed automata [2] is based on timed regular languages, where symbols in a word are associated with a real-valued time stamp. Formally, a *timed word* is a pair (σ, τ), where $\sigma = \sigma_1, \sigma_2, \ldots$ is an infinite word over some alphabet Σ of events, and a *timed sequence* $\tau = \tau_1, \tau_2, \ldots$ is an infinite sequence of time values $\tau_i \in \mathbb{R}$ that satisfies certain constraints (monotonicity and progress). Given a timed word, a *run* of a timed automaton is an (infinite) sequence of state transitions, analogous to standard regular languages defined by standard automata. For convenience, we extend the concept of timed words such that the inputs σ_i do not have to consist of exactly one event, but constitute arbitrary *input valuations* that assign a value and/or presence status to each input variable.

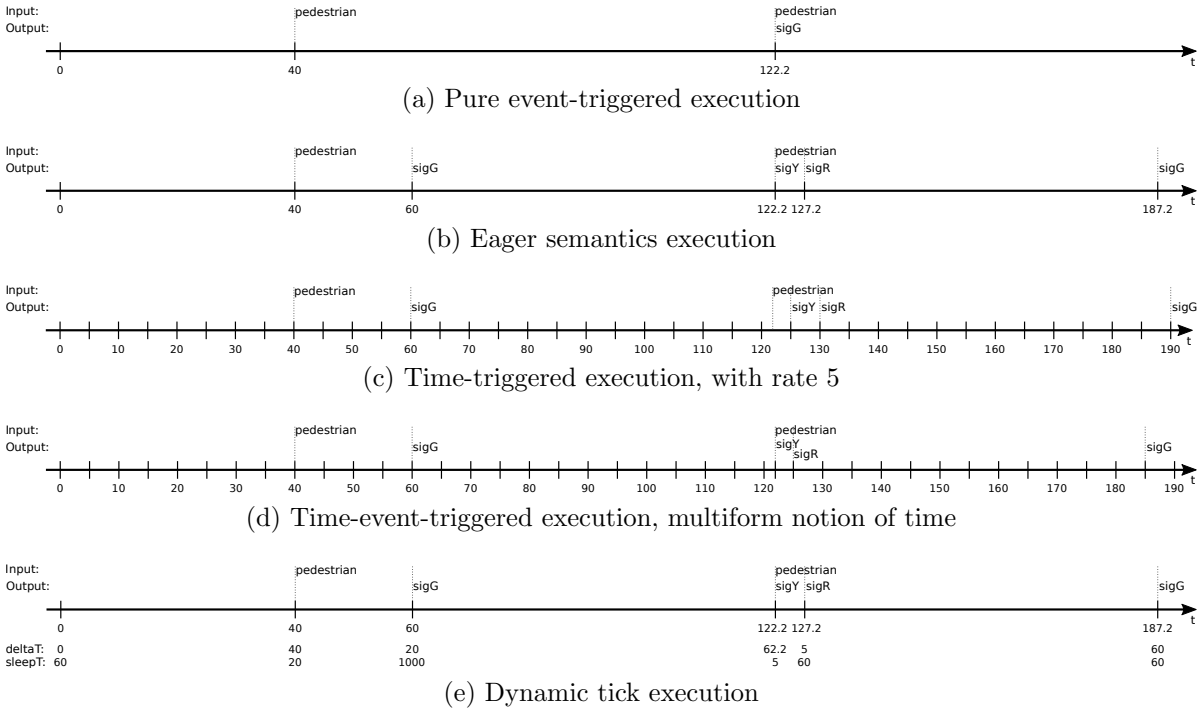


Figure 2.2: Execution traces of the traffic light controller based on different semantics. Vertical strokes denote reactions.

To make things concrete, assume that in our traffic light controller the pedestrian button is triggered at times 40 and 122.2. We denote this as input trace (timed word) $(\langle \mathsf{pedestrian}, 40\rangle, \langle \mathsf{pedestrian}, 122.2\rangle)$; we thus allow input sequences to be finite, and we use a notation that associates each input valuation directly with a time stamp. Given such an input sequence, our timed automaton performs a sequence of *reactions*, or *ticks*, one for each time-stamped input valuation.

A first non-obvious question this raises is how system initialization should be handled. In principle, there is nothing that requires that the first reaction of the system must occur at time zero. Furthermore, the "initial transition" to state red is not really a transition, but rather a convenient way to specify initial values for variables, including clocks. However, it does seem reasonable to let the clock x assume the initial value 0 at time zero, and to make this explicit by performing an initial reaction with an empty input (denoted ϵ) at time zero. The resulting input trace is $(\langle \epsilon, 0\rangle, \langle \mathsf{pedestrian}, 40\rangle, \langle \mathsf{pedestrian}, 122.2\rangle)$.

As illustrated in Fig. 2.2a, the traffic light controller reacts to this input trace by initializing itself at time 0, doing nothing at time 40, and then, at time 122.2, transitioning to green and emitting sigG. Then there is no further reaction due to the absence of further input events. However, this behavior is probably not what the creator of the traffic light controller intended. For example, the output sigG should probably not occur at time 122.2, even though $122.2 \geq 60$ certainly holds, but rather at time 60. Thus, we conclude that just the passage of time (without further input events) should also be able to trigger a reaction, in particular if the automaton contains transitions that are

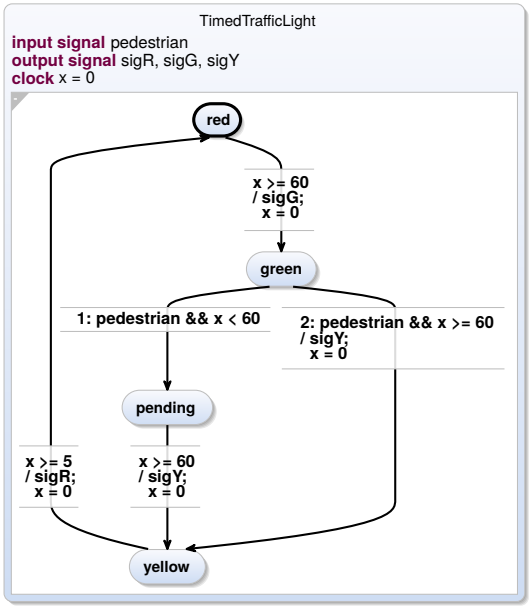
guarded solely by timing constraints. Lee and Seshia [18] resolve this by assuming that a transition is taken as soon as it is enabled. This assumption, which we denote as *eager* semantics, leads to the trace in Fig. 2.2b, which augments Fig. 2.2a by further reactions, all with empty input valuations, at times 60 (emission of sigG, transition to green), 127.2 (emission of sigR), and 187.2 (sigG again). The remaining traces are explained in Chap. 3, along with their execution concepts.

2.3 Timed SCCharts

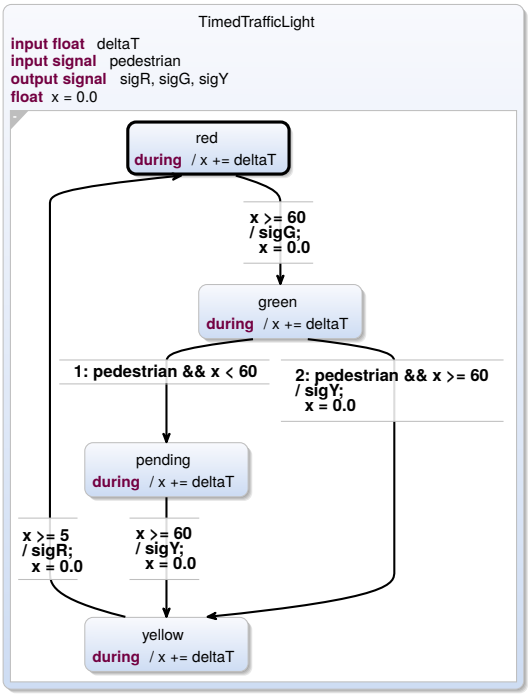
As it turns out, the synchronous model of execution fits quite naturally for timed automata as well. We here illustrate this with the SCCharts language. SCCharts provide many different language features, however, most of these are *extended features* that can be mapped to a very small set of *core features*. These extended features can be considered just as syntactical sugar, and the SCCharts compilation consists largely of model-to-model transformations that replace extended SCChart features by simpler features [25].

As we illustrate now, the timed-automata clocks can be added as an *extended SCChart feature* [25] without too much difficulty. Fig. 2.3a shows the SCChart realization of the traffic light controller. Despite some minor syntactical differences, the structure of the state machine itself and its transitions and their effects are the same as in Fig. 2.1. The new SCChart keyword clock here declares a clock x, which then, as in timed automata, can be set to arbitrary values and can be used to guard transitions. We here use the float data type for clocks, other types (including integral types) would also be possible.

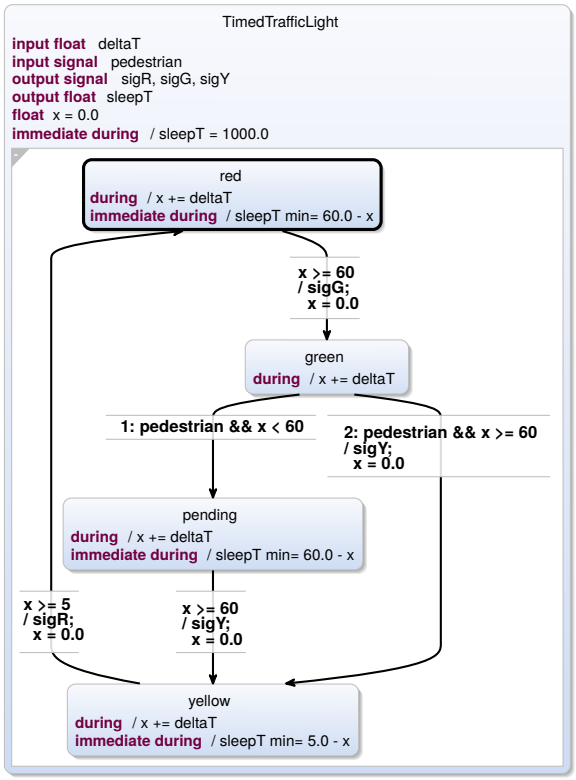
Fig. 2.3b presents the compiled intermediate result of TimedTrafficLight, revealing its actual internal implementation and behavior. In comparison to the original model in Fig. 2.3a, x is now an ordinary floating point variable, and the SCChart has an additional input deltaT. The only obligation on the run-time environment is, at each tick, to set deltaT to the time passed since the last tick. Based on these time increments, the SCChart itself keeps track of the progression of clocks. Specifically, the progression of time for the clock x is represented by during actions in each state, which increase the clock x by deltaT multiplied by the slope, which we omit here since it is 1. A during executes its effect in every tick its state is active, *except* for the tick the state is entered; this is important since only the time passed inside the state should be considered. Note that x may instantaneously assume up to three different values within a tick: the value at the beginning of a tick, the incremented value computed by the during action, and the reset value when a transition is taken that resets x to zero. This is no problem under the sequentially constructive (SC) semantics of SCCharts [25]. Applying the same idea to classical, non-SC synchronous languages would be a bit more involved, but with for example SSA-like renamings a synchronous language such as Esterel can also support multiple values per tick [21, 22].



(a) Original SCChart, with clock declarations



(b) Transformed SCChart



(c) Transformed SCChart with dynamic ticks

Figure 2.3: Traffic light controller modeled as as timed automaton in SCCharts, with various compilation/expansion stages.

3 When to React?

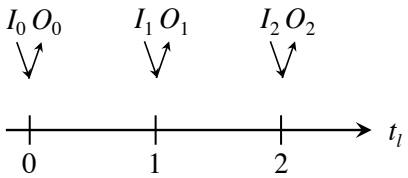
Timed automata allow to add timing constraints to transitions based on a real-valued clock. It is clear that if the constraint is not met, the transition must not be taken. When the constraint *is* satisfied, the automaton *can* react. As discussed in Sec. 2.1, it seems advisable to tighten this by saying that we want to react as soon as possible, which we denoted as the eager semantics. Still, the non-trivial question remains of how to make sure in practice that reactions occur on time to implement an eager semantics, or how to at least approximate it in some reasonable manner.

As it turns out, the question of when an automaton should react is not restricted to the "timed" setting presented here, but arises in synchronous programming in general. There, time is separated into discrete instants at which the system performs a reaction (*tick*). The *synchrony hypothesis* states that the reaction itself takes conceptually no time and that the actual time passes *between* reactions. Fig. 3.1a illustrates this concept.

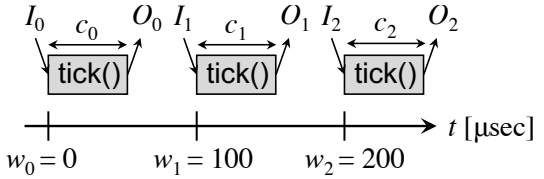
In practice, a synchronous program is synthesized into a tick function to perform reactions. Executing a tick takes computation time and separates inputs from outputs, as shown in Fig. 3.1b.

3.1 Event-Triggered Execution

In an entirely event-triggered execution, a reaction is triggered when an input (signal) changes. Hence our traffic light example would only react if the pedestrian input event occurs, as already illustrated in the trace in Fig. 2.2a. This execution regime is obviously insufficient as e. g. it does not trigger transitions with only timing constraints, as



(a) Logical time: time is discretized into logical ticks 0, 1, etc. Input I_i is synchronous with output O_i, the reaction time is abstracted to be 0.



(b) Physical time: the computation of the i-th reaction, corresponding to logical tick i and the i-th call of the tick function, begins at *wake-up time* w_i. Inputs are read at the beginning of the computation, outputs are written at the end of the computation.

Figure 3.1: Different timing abstractions [24].

discussed in Sec. 2.2. Consequently, a concept is needed which performs reactions based on time while handling the continuous nature of time.

3.2 Time-Triggered Execution

A common alternative to event-triggered execution is a periodical invocation of the tick function. One fixed global period is determined by analyzing the timing constraints of the model and its environment (i.e. poll rate of sensors), and sometimes also its worst case reaction time, to allow on-time executions of ticks. Fig. 2.2c illustrates a trace with this execution semantics for our example in Fig. 2.3a. The period is 5, which is the greatest common divisor of the two relevant timing constants 5 and 60 in the model, and hence a sufficient sample rate for the systems timing constraints *if* events are discretized to this rate as well. The system only reacts every 5 sec, which causes the pedestrian input occurring at time 122.2 to be processed *only* in the next period at time 125, consequently the sigR signal is also emitted at time 130. This behavior might be sufficient, especially when there are corresponding hardware sample rates for hardware sensors such as the pedestrian button.

Drawbacks of this execution regime are (1) the discretization of events (the pedestrian event is processes 2.8 sec after its occurrence) and (2) efficiency. For example, for a delay of 60 as in red, there are always 12 ticks executed, even though the transition can only be taken in the 12th tick. The previous invocations are wasted processor time and energy, which is problematic especially in embedded use-cases.

3.3 The Multiform Notion of Time

When modeling temporal behavior, classical synchronous languages, such as Esterel, consider time as an arbitrary discrete input event to the program. For example, this could be a signal that is present in each tick a second has passed; however, equivalently, one could choose a signal that represents that a travelled distance has increased by one meter. The progression of time is measured by counting occurrences of some signal. This is also referred to as the *multiform notion* of time. This concept is quite flexible; however, in particular if multiple input signals are used to model time, say one signal for milliseconds and one for microseconds, this concept can easily lead to temporal inconsistencies, as discussed further by Bourke and Sowmya [7].

For our SCChart in Fig. 2.3a, the trace in Fig. 2.2d represents an execution semantics using discrete timing events in combination with input event triggering. Since the model has two timing-related guards, 5 in state yellow and 60 in the others, we again opt for the greatest common divisor and use a timing event, let's denote it as fivesec, that indicates that 5 sec have passed since the last occurrence of fivesec. As the trace illustrates, the system reacts every 5 sec, always with fivesec present, and additionally at time 122.2 sec, when pedestrian is present, but fivesec is absent. We call this *time-event-triggered* execution, since a reaction is triggered when either the timing-event fivesec or some other

event occurs.

Consider time 122.2, when the pedestrian input is processed and sigY is emitted. Since time is measured by counting fivesec events, and the last such event has occurred at time 120, the pedestrian event is effectively considered to have taken place at time 120. Consequently, sigR is *already* emitted at time 125 instead of 127.2; thus not 5 sec have passed since sigY, but only 2.8 sec, which is not compliant with the original traffic controller specification. Similarly, sigG is emitted at time 185, which is also earlier than in the trace in Fig. 2.2b. For this input trace, one could comply to the eager semantics by increasing the granularity of the discrete time event, i.e. using an event for 0.1 sec passed. However, this would increase the number of reactions and load on the system significantly, while most of the reactions would not actually affect the state of the automaton.

3.4 Dynamic Ticks

To circumvent the difficulties of the execution regimes discussed so far, we here propose to not discretize time beforehand and to not model time by counting events, but propose to model time as continuous entity. Note that we still perform discrete reactions, only the time stamps are chosen from a real-valued domain, and in practice, this domain is also approximated by discrete types such as float.

This view of time as a continuous entity can be naturally combined with the concept of *dynamic ticks* [24], where the program itself outputs a request how long the environment can wait or *sleep* until the tick function should be executed again, the *wake up* time. Dynamic ticks can be combined with event-triggered execution, thus one may again react to both the passage of time and external events. Note that this concept preserves the determinism of the synchronous system [24].

This results in a dynamic and efficient execution, as illustrated in Fig. 3.1b. The wake up time w can either be set by an external global period or with dynamic ticks by the preceding tick function, adapting to the actually enabled reactions. Additionally, in situations where the reaction of the system depends on input events rather than time, dynamic ticks should be combined with event-triggered ticks, since no definite wake up time can be determined. As discussed in the next section, dynamic ticks in combination with event-triggered execution allow the implementation of the eager semantics (trace in Fig. 2.2b).

4 Dynamic Ticks in SCCharts

Fig. 4.1 illustrates the general structure that we propose to incorporate physical time into a reactive execution setting. As usual for an embedded system, a Tick Function communicates with its Environment, reading inputs from Sensors and conveying outputs to Actuators. Additionally, there is a Trigger Unit that calls the tick function, i. e., triggers one reaction (a tick). This classical setup is augmented by dynamic ticks, highlighted in red. Not only inputs trigger the execution (event-triggered) but there is also a Time Manager for time-triggered execution. This Time Manager is responsible for providing deltaT, the time passed since the last execution of the tick function, and it performs the waiting for the next time trigger based on sleepT. The new input and output extend the environment of the tick function.

4.1 The Traffic Light Controller with Dynamic Ticks

Our SCCharts traffic light control example in Fig. 2.3b can easily be further extended to use dynamic ticks, resulting in the SCChart shown in Fig. 2.3c. It has an additional output sleepT for the time span until the next time-related wake-up. In the root state there is an additional immediate during action, which executes its effect at every tick the state is active, including the tick the state is entered, due to the immediate modifier. It sets sleepT to an appropriate default value (1000.0 in this example), which is then updated in the states requesting an earlier wake up. This is done by further immediate during actions which register the remaining time until a guard of this state can be activated. The min= is an *update assignment* that assigns the minimal value between the current value of sleepT and the rhs expression. (As detailed further elsewhere [25], the

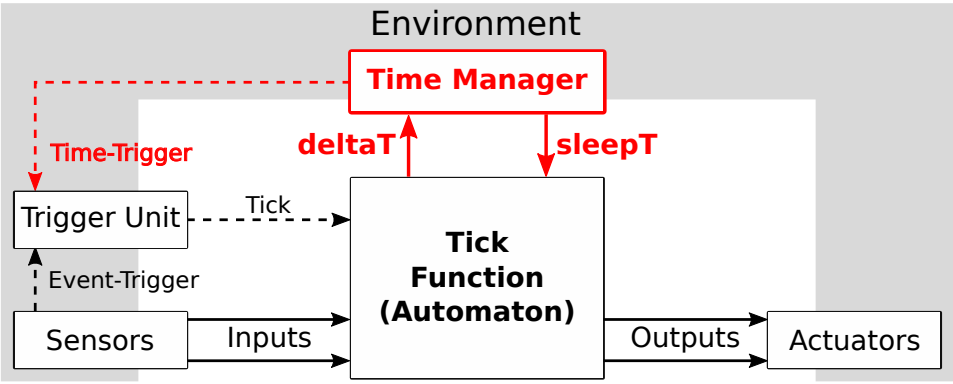


Figure 4.1: Controller and environment of a dynamic tick function.

SCChart semantics deterministically schedules "updates" such as +=, *=, etc. after other assignments, hence there is no race condition between the assignment of the default sleep time and the min= assignments.) The requested sleep time is calculated from the timing constraints of the outgoing transitions, further discussed in Sec. 4.2.

The resulting behavior is illustrated in Fig. 2.2e. The dynamic reaction times emulate the eager semantics (Fig. 2.2b), which we chose as the preferable execution semantics for timed automata. The system reacts to the pedestrian input at time 40, the state of the automaton does not change; however, as illustrated by deltaT and sleepT presented under the time line, the dynamic ticks adapt to the event-triggered invocation and correctly compute a new sleep time of 20. After the output of sigG at time 60, no wake up time can be computed since no transitions primarily depends on timing constraints, hence the default sleep time of 1000 is taken. The trace also shows that the reaction to the pedestrian event at 122.2 is also "on time," and the output of sigR is exactly 5 sec after this event. Dynamic ticks use only a minimal number of reactions, as necessary to process all events and to perform all transitions at their expected time.

4.2 How to Compute Sleep Times

The main task in computing the sleep time is to detect if and which passage of time causes a transition to be enabled. Our SCChart compiler computes sleep times based on a static analysis of the timing bounds in the outgoing transitions of a state, with certain restrictions of timing constraint specifications to facilitate their implementation. More specifically, we look for timing constraints of the form $c \geq ltb$, where c is a clock and ltb some expression that we refer to as *lower timing bound*. We compute the corresponding sleep time as the difference between ltb and the current clock value. For example, state red in Fig. 2.3c has an outgoing transition with guard x $\geq$ 60, hence red gets augmented with an immediate during action that computes sleepT min= 60.0 - x. If a state has multiple outgoing transitions with lower timing bounds, we assign the minimal positive sleep time. To simplify the detection of lower timing bounds, our implementation rules out negations of timing constraints, but that does not limit expressiveness; for example, !(x $<$ 10) should be written as x $\geq$ 10. Furthermore, constraints specifying an *upper* bound do not contribute to the sleep time since they, considered separately, do not require time to pass to be enabled and hence would result in a sleep time of zero.

Our example in Fig. 2.3c shows another case where no sleep time is requested and the value of sleepT should fall back on the default value. In state green, both outgoing transitions primarily depend on the pedestrian input, and x only distinguishes the two paths. The two timing constraints are *non-triggering* in that just the passage of time does not make a difference in whether any outgoing transition is enabled or not. If pedestrian is false, we do not take any transition, and if it is true, we take a transition, no matter what time it is; the time indicated by x solely decides *which* transition we take.

To detect such non-triggering timing constraints, assume that the i-th outgoing transition of some state has a guard $G_i = C_i \wedge T_i$, where C_i is a condition that does not

depend on time and T_i is a timing constraint. Assume that no guard is currently active, i. e., $\bigvee_i G_i$ = false. Furthermore, assume that T_1 specifies a lower timing bound *ltb*. This would usually require the computation of a corresponding sleep time, unless T_1 is non-triggering—which is the case if $\exists i$ such that C_1 implies C_i and $\neg T_1$ implies T_i (i. e., whenever the *ltb* has not been reached yet, T_i holds). In our implementation, we further simplify the conditions and assume that C_1 and C_i are the same boolean guard, and T_1 and T_i are mutually exclusive unbounded timing constraints. As it turns out, the guards on the outgoing transitions from green fulfill that criterion, taking pedestrian && x >= 60 for C_1 and pedestrian && x < 60 for C_2, thus the compiler classifies 60 to be a non-triggering *ltb* and does not compute a sleep time for it.

The concept of computing sleep times based on lower bounds is closely tied to the eager semantics. With an perfectly eager execution, it would by sufficient to write x ≥ 60 as x = 60, but considering a real-valued time and a realistic implementation with possible timer imperfections, the first option is more robust and thus preferable. Similarly, we prefer closed timing intervals as specified with ≥ over open intervals specified with >.

However, the behavior specified by timing constrains can change when using a semantics other than the presented eager one or if the reactions are delayed. If for example more time than the minimum of a specified lower bound passes, it is possible that other transitions get also enabled or disabled, which may change the expected behavior. Assume the example that a state is entered when at least 60 sec passed (x ≥ 60) and is immediately (in the same tick) left when at most 80 sec have passed (x < 80), without any reset of the clock. With eager semantics, the state will be entered after a time of 60 and then left immediately. If the reaction is delayed or another execution semantics is applied and the system is able to react after a time of 80 for the first time, then the state is entered but can never be left, leaving the system in that state forever. One could argue such system is designed badly, but this is the reason why we prefer the eager semantics. Note that for dynamic ticks we only trigger reactions on transitions that require time to pass, such as ≥ constraints, but when a < constraint is the only guard then there is no wake up. Otherwise such constraints with delayed transitions would cause a sleep time of zero which contradicts the concept of a delayed reaction. Note that delayed transitions, in contrast to immediate transmissions, require the state machine to stay for at least one tick in the state before it can be left using a delayed transition. In the previous example the transition x < 80 is not delayed and the state can be left in the same reaction as entered.
Nevertheless, we also want to discuss the loosening of the eager semantics based on dynamic ticks in the next section.

4.3 The Greedy Semantics

Dynamic ticks can be further extended to introduce *soft bounds*, leading to a *greedy* semantics that loosens the regime of the eager semantics. To motivate, consider the minimal SCCharts example in Fig. 4.2. There are two regions Fast and Slow, each one uses a timed automaton to react. Assume that the time scale of this example

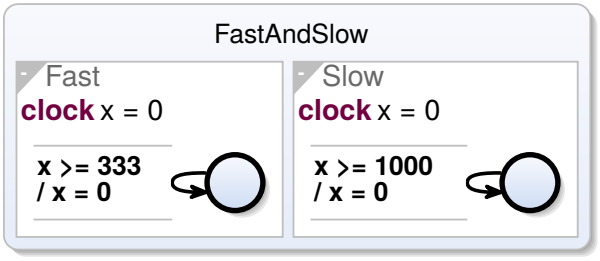


Figure 4.2: SCChart motivating the use of *soft bounds* in dynamic ticks.

is microseconds, thus Slow should react every millisecond and Fast three times faster. Hence, starting at time zero, the third reaction of region Fast will be at time 999, leaving only one microsecond to invoke the reaction of Slow, which might be infeasible for the environment.

To circumvent such short sleep times, we introduce soft bounds. States with an outgoing transitions with a soft bound still compute their own sleep time, but speculate to possibly "piggyback" on a somewhat earlier reaction invoked by another state.

Specifically, the user may, in region Slow, replace the *hard bound* $x \geq 1000$ by the *soft bound* $x \geq 990 \,||\, x \geq 1000$. Our implementation detects this pattern and will request a sleep time of 1000 for this state, as for the original hard bound specified with $x \geq 1000$; however, at run time the transition may already be taken at time 999, thus subsuming the sleep time of 1000. This favors earlier reactions over late reactions, prevents very small sleep times, and possibly reduces the total number of reactions.

4.4 Delayed Reactions

As a result of the eager semantics and its implementation using dynamic ticks, which assumes that the system reacts immediately to the enabling of a transition, the question arises: *Can we react in time?* The greedy semantics already allows to specify a certain amount of slack in the sleep time, but here we want to focus on the problem of possible deviation between the requested wake time and the actual execution time and their effects of the system, relevant for both dynamic ticks and execution with a fixed period. Since the system contains clocks, it requires a time input form the Time Manager, in case of SCCharts deltaT, and there are different options to provide this input.

The first option is an artificial simulation, where the passed time is always the time that should have passed, in case of the dynamic ticks the sleep time or the fixed period if used. This encapsulates the system in a perfect world, where the execution is independent from the real physical time.

The other option is to pass the real time to the system. That has the effect that the deviation between the requested wake up time and deltaT that is passed to the system depends on the environment. Executing a tick *before* the requested sleep duration has passed does not affect the behavior since the clocks only increment by deltaT and the related time constraints do not trigger. However, when the tick is executed *after* the

requested sleep time, the additional delay time will affect all clocks, but when a clock is reset this time is lost.

For our traffic light example in Fig. 2.3c that means if the automaton is in state yellow and for some reason 70 sec, instead of 5 sec in eager execution, pass before the system is able to react. It enters state red, resets the clock to zero and starts waiting again. However, enough time passed that the system could have transitioned to red and then to green if more reactions were possible. Hence, one could argue that we want to process this additional time and try to "catch up" to the actual time. This can be done by resetting clocks not to zero but instead subtract the time that should have passed. In this example in combination with dynamic ticks the clock would be reduced by 5, leaving 65 on the clock, which is then consumed in the next reactions, triggered by a resulting sleep time of zero. However, in this use case of a traffic light it is quite inappropriate to do such "catch up" because there would be no actual time between switching from red to green, because this time already passed when yellow was lit. Nevertheless, there might be cases the additional passed time should stay on the clock.

5 Multi-Clock SCCharts

Timed automata naturally support multiple clocks and so does its SCCharts implementation. In synchronous languages, there is also the concept of multiclocking [11], as in Multiclock Esterel by Berry and Sentovich [5]. In that context the term "clock" does not relate to a real-valued time measurement but a hardware clock that drives a hardware circuit or similarly designed software. In multiclock systems, different parts of the program are activated by different clocks, which are additional inputs to the program and effectively refine the base clock. Our concept presented so far can be further adapted to allow such multiclocking.

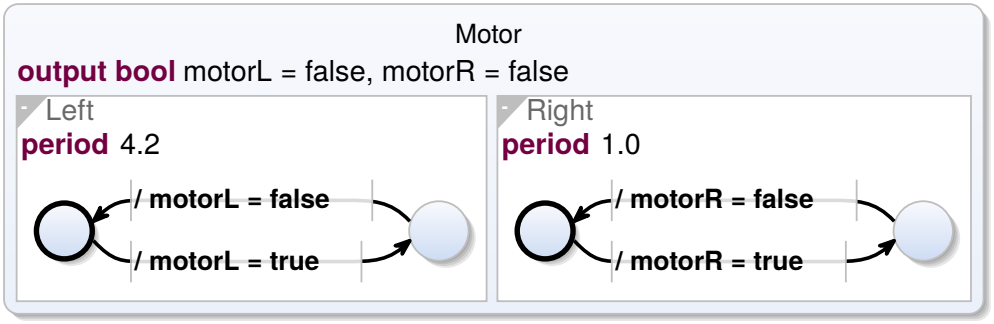
We have augmented SCCharts with an additional extended feature period, which controls the activation of states and regions based on a real-time clock. The period command ensures that the guarded state or region is only activated if the given amount of time has passed since the entering/start of the state/region or its last activation.

5.1 The Motor Example

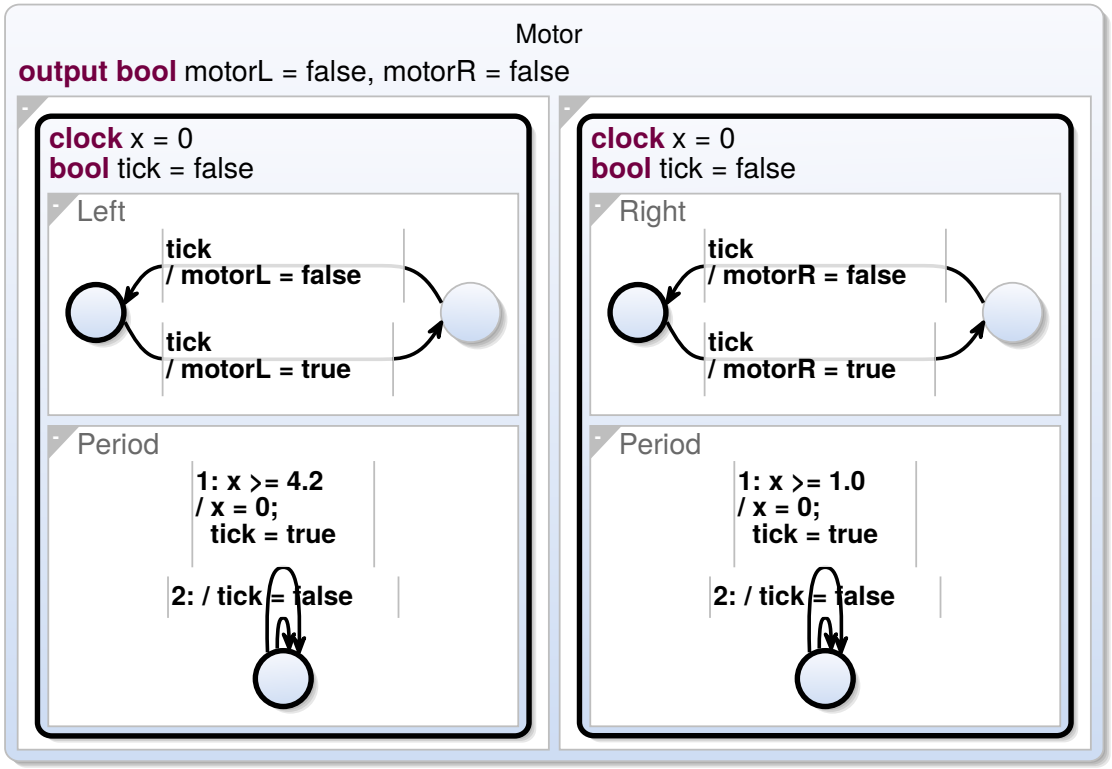
To illustrate the usage and effect of the period feature, Fig. 5.1 presents the SCCharts example Motor. This represents a controller for two rather simplified stepper motors, for example to drive a robot. There is a left (motorL) and right (motorR) motor, which are run by toggling the corresponding boolean output at a certain frequency. The SCChart has two concurrent regions, each controlling one motor with a simple state machine with two states. The transitions cycle between the states and toggle the motor variable. In our example, assuming time units of msec, the left motor must toggle every 4.2 msec, which is represented by the period annotation in the region. The right motor is run with a period of 1 msec.

To inspect the internal implementation of the extended period feature, Fig. 5.1b shows the compiled intermediate result of Motor. The periods are transformed into timed automata, as introduced in Chap. 2, to control the timing of the regions. In region Left, the inner states of the region are moved into a new super state that declares a new clock variable x and a boolean flag tick. The tick variable acts as guard for all reactions in the original state machine, now present in the inner region named Left. This prevents the inner SCChart from performing any action if the clock is false. If any transition or action has its own guard, it would be conjuncted with tick. Here tick is initialized to false, which means that no reaction takes place in the initial tick; however, we might also initialize tick to true, which would cause a reaction in the initial tick as well.

There is also a new region Period with a single-state timed automaton. At each tick when the clock x reaches the period's threshold, the tick variable is set to true and



(a) SCChart with period annotation



(b) Transformed SCChart

Figure 5.1: Motor example modeled in SCCharts with periodic regions.

enables the reaction in the other region. Otherwise, indicated by the transition with the lower priority (2:), the variable is set to false. Analogously, the Right region is affected by the period transformation. Note that one might also add the clock logic directly within the existing Left and Right regions. However, we decided to add the separate Period region and explicit tick guard, to reduce the number of timed guards in the model and to have a clear separation between timing and the original SCChart. In the process of compiling SCCharts, the next step would be to transform the clock feature as conceptually presented in Fig. 2.3.

6 Extension with Clock Patterns

As we have introduced clocks and tick flags that represent activation conditions of regions, we discuss here some possible use of those ticks. In particular, we want to make explicit relationships between these ticks just as in polychronous systems [12] and multi-clock implementations [11]. The Clock Constraint Specification Language (CCSL) [3] has been defined as a language to handle clocks and to specify pure clock-related constraints independently of a specific programming language. CCSL sees clocks as infinite sequences of ticks and can define when a tick (therefore a region) should tick or cannot tick. We propose to annotate an SCChart with CCSL constraints that make explicit the rate relationships amongst the various regions and states. This can be done as a pure syntactic extension as long as such a specification can be compiled (internally) into a valid SCChart.

CCSL provides a concrete syntax to handle clocks, whether logical or physical, as first-class citizens. It provides patterns of classical clock constraints (like periodic, sporadic) that can be of three types: *synchronous* clocks are directly inspired from primitive constructs of synchronous languages [4]; *asynchronous* clocks rely on the relation "happens-before" from Lamport's logical clocks [16]; and *real-time* clocks represent physical time. Real-time constraints are usually a special case of the logical ones. For instance, CCSL defines both a real-time and logical notion of periodic behavior. A clock a is periodic on another clock b with period p if a ticks synchronously at every p^{th} tick of b. If b is a physical (real-time) clock (*e.g.,* s), then its a classical periodic behavior, otherwise it remains purely logical. The semantics of each CCSL constraint is an automaton and a CCSL specification is the synchronized parallel composition of those automata [19].

Synchronous constraints are encoded as pure finite-state automata. *Asynchronous constraints* rely on state machines with unbounded integer counters. In TimeSquare [9], real-time constraints are encoded as a composition of logical constraints. However, real-valued clocks can also be encoded as timed automata [23], and the dynamic tick mechanism provides an efficient way to encode them in SCCharts. The goal here is to annotate a SCChart with CCSL constraints. This relies on the explicit tick flag introduced in Fig. 5.1b. CCSL annotations can either force the tick to occur (and therefore the region to execute) or observe unexpected behaviors and raise alarms. Both examples are illustrated in this section.

Figure 5.1a illustrated real-time clocks. In that model, the periodic behaviors of both regions are relative to an absolute real-time reference, assumed to be msec in that example. Alternatively, we can define the relative periodicity of the regions in CCSL as some rational period p, as in **repeat** left **every** 4 right, where left is a clock associated with the left region and right is a clock associated with the right region. The semantics of this constraint is given as a simple finite-state automaton that can be encoded as a SCChart

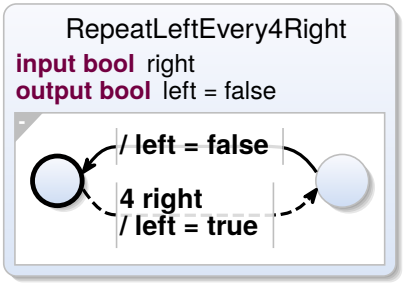


Figure 6.1: Expansion of logical periodic constraint

in a straight-forward way, as illustrated in Fig.6.1. There the guard "4 right" is a *count delay* that becomes enabled after four occurrences of right.

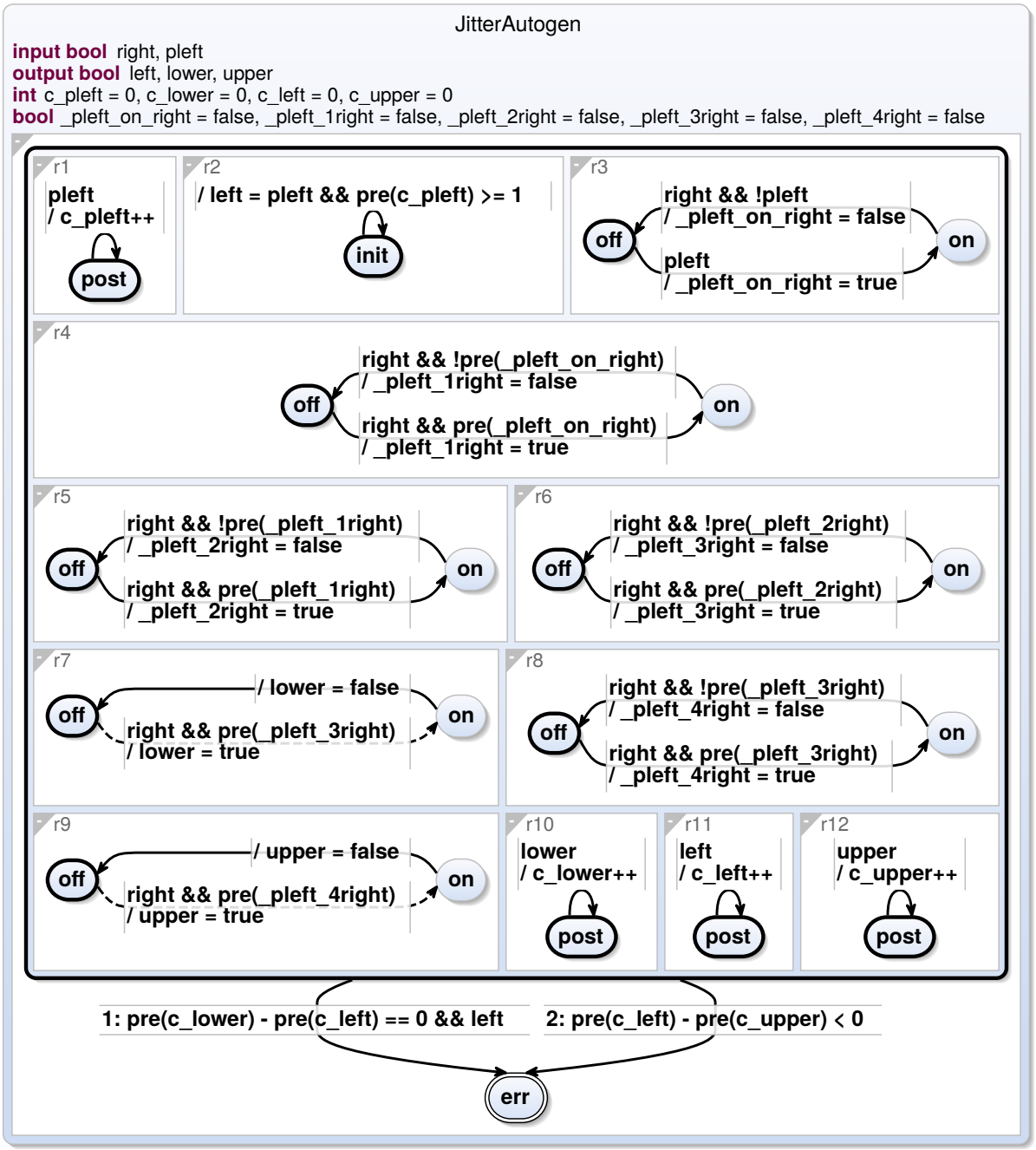
Such synchronous constraints specify a fully determined behavior. When using asynchronous constraints, we may get a partially undetermined behavior. Consider, e. g., a periodic behavior with jitter as in **repeat** left **every** [4,5] right. This constraint expands as the following primitive CCSL constraints:

1 lower = pLeft **delayedBy** 4 right
2 upper = pLeft **delayedBy** 5 right
3 lower $<$ left $\leq$ upper

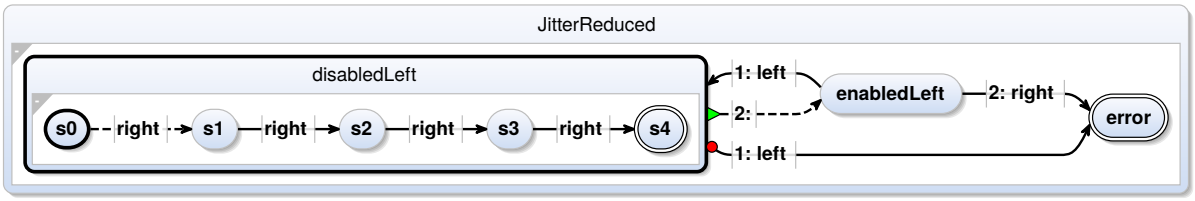
where pLeft is defined by the constraint left = pLeft $ 1. The $ is used for unit-delay as in Signal, it defines pLeft as the same clock as left preceded by one more tick, like a pre operator.

lower represents the lower bound for left to tick while upper is the upper bound. The last equation forces left to tick strictly after lower and before upper. Each of these constraints can be encoded as concurrent SCCharts (see Fig. 6.2a). TimeSquare builds the synchronized product of these automata to compute a finite state automaton that can be encoded as a simpler SCChart, see Fig. 6.2b; note that the dashed transition leaving s0 is *immediate*, meaning that it could be taken immediately in the tick when s0 is present.

The behavior exposed in Fig. 6.2 describes clock relations between the two regions left and right. At the same time, it observes whether or not the regions behave as expected. Clocks left and right are inputs and a wrong sequence of inputs would lead into the error state, like an assertion. It also enables or disables the code in regions. In state enabled_left, the region left is enabled and its code is executed as expected. In other states, the region is disabled and its code should be ignored.



(a) Automatically generated SCChart for jitter



(b) Simplified SCChart for jitter

Figure 6.2: Periodic behavior with jitter.

7 Related Work

Timed automata [2] introduce real-valued clocks to describe the temporal behavior of systems using a continuous notion of time. Usually some progress conditions are required [14] to avoid time divergence and to guarantee that the system does not remain idle forever. While timed automata and their multiple extensions are originally defined with an asynchronous semantics, we here propose to harness them in the synchronous settings of SCCharts.

As presented here, the clock feature models single-rate clocks, as initially proposed by Alur and Dill [2], since it relieves the modeler of explicitly handling time. However, note that the clock type is only a convenience feature, and a user can always model SCCharts directly as presented in Fig. 2.3b and implement multirate clocks by scaling the change of x in the during actions. As discussed by Sifakis et al., multirate timed automata can be mapped to timed automata [20], and in the traffic light example, that transformation is rather straightforward as the only clock x always moves at the same speed.

Harel [13] also proposes time extensions to statecharts where a time t is associated with every transition and t refers to a global notion of discrete time steps. We consider here both discrete and real-valued models of time.

Zelus [6] is a synchronous language that mixes both discrete-time and continuous-time behaviors. Continuous behaviours are described through ordinary differential equations. We are not decribing continuous behaviours here but provide an extension to SCCharts to make explicit the activation conditions of regions under the form of clocks that serve to express both real-valued and logical constraints.

As explained, this work builds on the concept of dynamic ticks proposed by von Hanxleden et al. [24]. Thus most of the related work discussed by them is also relevant for this work. This includes for example the work by Jourdan et al. on extending ARGOS with timing constructs [15], or PTIDES (Programming Temporally Integrated Distributed Embedded Systems), which addresses the design and implementation of distributed real-time embedded systems [10].

The dynamic ticks is akin to the agenda of timed events used in discrete event system specification (DEVS) [**?**] to always pick the most urgent event without relying on a timed-triggered strategy. However, the sequentially constructive semantics of SCCharts, which permits instantaneous modifications of variables under consideration of data dependencies, reduces the need for so-called delta-cycles.

8 Conclusions and Outlook

We have investigated how to incorporate physical time into the synchronous model of execution. As it turns out, timed automata can be mapped naturally to the synchronous setting, requiring only minimal support from the environment. However, to achieve a concrete implementation also requires to settle for a concrete, unambiguous semantics that specifies not only when a system *may* react but also when it actually *does* react; to that end, we have settled for the eager semantics, as also suggested by Lee and Seshia [18].

We have proposed two extensions to SCCharts, namely clocks and periods, that can be mapped directly to standard SCCharts. We have implemented these extensions as part of an open-source compiler[1]. We expect that similar extensions could be implemented in other synchronous languages, such as Esterel, Lustre or also SCADE, as they for example also facilitate the "during actions" required for tracking clocks.

We have cast the concept of clocks and time in the context of physical time and durations. However, for us the only practical requirements on clocks are the ones that timed automata cast on clocks, namely monotonicity and progress. Thus, one might also consider other (at least conceptually) continuous entities as clocks, such as distance travelled. In other words, the multiform notion of time could also be applied to time as proposed here, all within a synchronous setting.

There are several directions to proceed from here. First, we would like to get more practical experience with the language constructs proposed here. The clock and period extensions already promise to be quite useful, but other, more high-level language extensions would be feasible as well, as suggested for example by the features already present in CCSL. Then, while the way these features are mapped to standard SCCharts seems natural and straightforward, more efficient mappings might be possible. Similarly, in the context of dynamic ticks, we currently have a rather simple heuristics to compute sleep times from timing constraints; more powerful static analyses might again lead to a more efficient implementation.

[1] `http://rtsys.informatik.uni-kiel.de/kieler`

Bibliography

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995.

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

[3] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[4] 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.

[5] Gérard Berry and Ellen Sentovich. Multiclock Esterel. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 110–125, London, UK, 2001. Springer-Verlag.

[6] 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.

[7] 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.

[8] 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 2017, Sophia Antipolis, France, September 13-15, 2017*, pages 1–11, 2017.

[9] Julien Deantoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In *50th Int. Conf. on Objects, Models, Components, Patterns (TOOLS)*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer, 2012.

[10] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE*, 100(1):45–59, January 2012.

[11] Abdoulaye Gamatié and Thierry Gautier. The Signal synchronous multiclock approach to the design of distributed embedded systems. *IEEE Trans. Parallel Distrib. Syst.*, 21(5):641–657, 2010.

[12] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. POLYCHRONY for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193 – 244, 1994.

[15] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *Proceedings of Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 347–358, June/July 1993.

[16] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[17] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[18] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition*. MIT Press, 2017.

[19] Frédéric Mallet and Robert de Simone. Correctness issues on MARTE/CCSL constraints. *Sci. Comput. Program.*, 106:78–92, 2015.

[20] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *Proceedings of the 6th Annual Conference on Computer-Aided Verification, Lecture Notes in Computer Science 818*, pages 81–94. Springer-Verlag, 1994.

[21] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. On reconciling concurrency, sequentiality and determinacy for reactive systems — a sequentially constructive circuit semantics for Esterel. In *2018 18th International Conference on Application of Concurrency to System Design (ACSD)*, pages 95–104, June 2018.

[22] Alexander Schulz-Rosengarten, Steven Smyth, Reinhard von Hanxleden, and Michael Mendler. A sequentially constructive circuit semantics for Esterel. Technical Report 1801, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2018. ISSN 2192-6247.

[23] Jagadish Suryadevara, Cristina Cerschi Seceleanu, Frédéric Mallet, and Paul Pettersson. Verifying MARTE/CCSL mode behaviors using UPPAAL. In *Software Engineering and Formal Methods*, volume 8137 of *Lecture Notes in Computer Science*, pages 1–15. Springer, September 2013.

[24] 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.

[25] 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)*, Edinburgh, UK, June 2014. ACM.