Real-Time Signaling in SDL*

Marc Krämer, Tobias Braun, Dennis Christmann, and Reinhard Gotzhein

Networked Systems Group,
University of Kaiserslautern, Germany
{kraemer,tbraun,christma,gotzhein}@cs.uni-kl.de

Abstract. SDL is a formal specification language for distributed systems, which provides significant, yet limited real-time expressiveness by its notion of time (`now`) and its timer mechanism. In our current work, we are investigating various ways to augment this expressiveness, by proposing language extensions and exploiting degrees of freedom offered by SDL's formal semantics. This paper presents some recent results of our work: a mechanism for real-time signaling, which can be roughly characterized as a generalization of SDL timers. More specifically, we propose to add the possibility of specifying a time interval for the reception of ordinary SDL signals, by stating their time of arrival and expiry. This extension can be used, for instance, to specify time-triggered scheduling, which is required in many real-time systems. In the paper, we present the concept of real-time signaling, propose a syntactical extension of SDL, define its formal semantics, outline our implementation, show excerpts of a control application, and report on measurement results.

1 Introduction

A real-time system is a reactive system in which the correctness of the system behavior depends not only on the correct ordering of events, but also on their occurrence in time (cf. [1]). Execution of a real-time system is usually decomposed into tasks (e.g., local computation units, message transfers), which are initiated when a significant change of state occurs (*event-triggered*), or at determined points in time (*time-triggered*), and which are to be completed before their deadlines.

SDL [2] is a formal description technique for distributed systems and communication systems, and has been promoted for real-time systems, too. By its notion of time (`now`) and its timer mechanism, SDL provides significant, yet limited real-time expressiveness. In SDL, a task can, for instance, be specified by a single transition (*simple task*) or by a sequence of related transitions (*complex task*).

In our current work, we are looking into various ways to augment SDL's real-time capabilities, by defining language extensions as well as by exploiting

* This work is supported by the Carl Zeiss Foundation and the German Research Foundation (DFG) within the priority program SPP 1305 "Control Theory of Digitally Networked Dynamical Systems" under grants GO 503/8-1 and LI 724/15-1.

degrees of semantic freedom (see, e.g. [3]). In this paper, we address the problem of specifying and executing *time-triggered task schedules* in distributed real-time systems, introducing and applying a mechanism called *real-time signaling*, with SDL as design language. In particular, we propose a syntactical extension of SDL, define its formal semantics, outline our implementation, and demonstrate *real-time signaling* in use.

Following this introduction are six sections. In Sect. 2, we present our concept of specifying and executing time-triggered schedules in SDL, and examine possible solutions. Based on our findings, we propose a syntactical extension of SDL for *real-time signaling* and present its formal semantics in Sect. 3. Section 4 gives an overview of the implementation of *real-time signaling*, including signaling to the SDL environment. In Sect. 5, we present excerpts of a networked control system and measurements that provide evidence for the effectiveness of our solution. Section 6 reports on related work. Section 7 contains conclusions and an outlook.

2 Conceptual Considerations

A time-triggered schedule is a list of tasks and their initiation points in time. To keep the list manageable, it can be restricted to tasks that are scheduled strictly periodically. In this case, it is sufficient to have one entry per task, defining period and time of first execution. For timely behavior, decomposition into tasks and their initiation points in time have to be carefully planned to ensure that required resources (e.g., CPU, shared medium) are available, and that tasks are finished before their deadline. In this paper, we assume that proper time-triggered schedules are available (see, e.g. [1]).

Given a time-triggered schedule, a design decision whether individual schedule entries are associated directly with corresponding tasks or whether they are collected in *schedulers* – dedicated components that manage a number of schedule entries and trigger tasks accordingly – is to be made. In the former case, scheduling information would be scattered over the entire system and thus be difficult to maintain. Therefore, we favor a more centralized solution with one or a few schedulers.

2.1 Realization of Time-Triggered Schedulers in SDL

In SDL, schedulers could be realized as components of the SDL Virtual Machine (SVM), which controls, inter alia, selection and firing of transitions. Here, the SVM would have to be configured with a time-triggered schedule before executing a given SDL system. Yet, it is not obvious how tasks consisting of one or more transitions could be identified and triggered by the SVM at runtime in an SDL semantics compliant way. Therefore, we favor another design choice, where schedulers are explicitly specified as regular SDL processes that are executed under the control of the SVM.

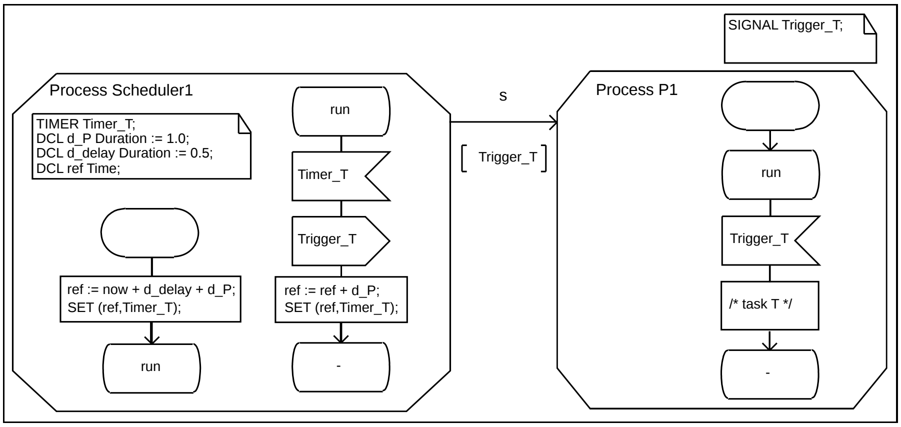


Fig. 1. `Timer_T` in scheduler triggers execution of task `T`

There are several ways of specifying schedulers in SDL. To keep examples short, we show two solutions for a single task `T` only, which is periodically triggered at times 1.5, 2.5, 3.5, ... One solution is to declare, for each task `T`, an SDL timer `Timer_T` in the scheduler, which is always set to the next initiation time of `T` (see Fig. 1)[1]. When `Timer_T` expires, a specific SDL signal `Trigger_T` identifying task `T` is sent by the scheduler to the SDL process `P1` hosting the task, and `Timer_T` is set again. On reception of `Trigger_T`, task `T` is eventually executed. While appearing to be straightforward, this solution creates substantial overhead, as a timer is required for each task. Also, there are several sources of delay[2], in particular, delays to create, forward, queue, and consume the signals `Timer_T` and `Trigger_T`, preventing `T` to be executed as scheduled.

The solution shown in Fig. 2 is an attempt to reduce delays. Here, we use one SDL timer `Timer_S` in the scheduler, which activates the scheduler periodically[3]. When activated, the scheduler triggers all tasks to be executed before its next activation. This is done by sending signals `Trigger_T` identifying task `T` to the SDL process `P2` hosting the task. However, since the trigger signals are now sent well before the actual initiation time, `P2` has to delay execution of task `T`. For this purpose, the signal `Trigger_T` carries a parameter of type `Time` with the task's next execution point in time `t`. When receiving `Trigger_T`, `P2` sets `Timer_T` to `t`, and executes `T` when `Timer_T` expires. We note that this solution eliminates several sources of delay, as the trigger signal is exchanged well ahead of the scheduled execution time of `T`. However, it produces roughly the same overhead in terms of SDL timers and signals.

[1] `d_delay` is the offset of the first executed timer.

[2] When referring to sources of delay, we assume that the SDL system is executed on a real hardware platform.

[3] The period of the scheduler and task `T` is set to same value `d_S`. `d_off` determines the relative starting time of the task relative to the execution time of the scheduler.

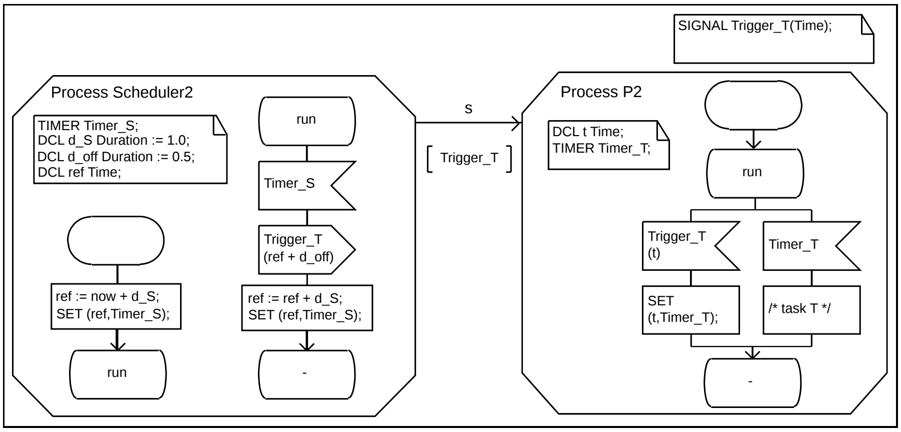


Fig. 2. External trigger and local task-activating timer

2.2 Drawbacks of SDL Solutions

As seen in Fig. 2, a normal *output* statement is used, sending a signal with the scheduled task execution time as parameter. The receiving process sets a local timer and executes the task on timer expiry.

This way of modeling time-triggered executions has several conceptual and execution time related drawbacks. Using an external signal to control a local SDL timer has impacts on the signal parameters and the design of the receiving process. Instead of just reacting and executing the task on signal reception, the process has to adopt the given pattern to be compliant with the scheduler. Thus, knowledge that should be encapsulated in the scheduler is scattered over all processes that host tasks. It follows that the reuse of these processes is limited and only possible in systems using the same type of schedulers. For instance, the use of the same task in a system with event-triggered scheduling requires an adaptation of the process specification because an immediate execution of the task is required.

Besides conceptual shortcomings, the standard SDL approach has also drawbacks concerning resources required at runtime. The process hosting the task has to be executed on reception of the trigger signal from the scheduler to set the local timer, and again when this timer expires to execute the task itself. In case of a temporary higher system load, the delivery of the trigger signal can be delayed. To cope with higher load, the process can be adapted to discard the received trigger signal and defer execution of the task to the next time a trigger signal is received (see Sect. 3.2). In SDL, this can be done by including an additional check before the task is executed. This results in higher overall system load, which is critical in the context of limited resources (e.g., embedded platforms) and has negative impacts on the timeliness of scheduled tasks.

3 Real-Time Signaling in SDL: Concept, Syntax, and Semantics

In this section, we propose the concept of *real-time signaling*, present a syntactical extension of SDL, and define its formal semantics. In particular, we add the possibility to specify the arrival time of signals (Sect. 3.1) and their expiry time (Sect. 3.2), and introduce timestamping of signals (Sect. 3.3) by adding new keywords (`at`, `expiry`, `sendtime`) to SDL. The required syntactic and semantic modifications of Z.100 [2] and Z.100 Annex F3 [4] are summarized in List. 1.1 and List. 1.2, and will be explained in the following subsections. With these preparations, we revisit the example of Sect. 2 and provide an improved solution in Sect. 3.4. In Sect. 2.2, we discuss and compare both solutions.

```
<output statement> ::= output <output body> <end>
<output body> ::= <signal identifier> [<actual parameters>] ...
                        <communication constraints>
<communication constraints> ::=
                        {to <destination> | timer <timer identifier> | <via path>
                        | at <time expression> | expiry <time expression>}*
<imperative expression> ::= <sendtime expression> | ...
<sendtime expression> ::= sendtime
```

Listing 1.1. Changes of the SDL syntax

```
shared atArg: PlainSignalInst → Time
shared expiryArg: PlainSignalInst → Time
shared sendTime: PlainSignalInst → Time
controlled sendTime: SdlAgent → Time

maxTime(a:Time,b:Time):Time =def
 if (b = undefined) ∨ (a ≥ b) then a else b endif

Output=def Signal× ValueLabel*× ValueLabel× ViaArg ×Time ×Time
      × ContinueLabel

EvalOutput(a:Output) ≡
  SignalOutput(a.s-Signal, values(a.s-ValueLabel-seq, Self), value(a.s-
        ValueLabel, Self), a.s-ViaArg, a.s-Time, a.s2-Time)
  Self.currentLabel :=a.s-ContinueLabel

SignalOutput(s:Signal, vSeq:Value*, toArg:ToArg, viaArg:ViaArg, atArg:
      Time, expiryArg:Time) ≡
    ...
     choose g: g ∈ Self.outgates ∧ Applicable(s, toArg, viaArg, g, undefined)
        extend PlainSignalInst with si
           si.plainSignalType:= s
           si.plainSignalValues :=vSeq
           si.toArg :=toArg
```

```
            $si.viaArg := viaArg$
            $si.atArg := atArg$
            $si.expiryArg := expiryArg$
            **if** $atArg = undefined$ **then** $si.sendTime := now$
            **else** $si.sendTime := atArg$ **endif**
            $si.plainSignalSender := Self.self$
            $INSERT(si, now, g)$
          **endextend**
          **endchoose**
      **endif**

DELIVERSIGNALS ≡
    **choose** $g$: $g \in Self.ingates \wedge g.queue \neq empty$
      **let** $si = g.queue.head$ **in**
        DELETE($si$,$g$)
        $si.arrival := maxTime(si.arrival, si.atArg)$
        ...
      **endlet**
    **endchoose**

SETTIMER($tm$:TIMER, $vSeq$:VALUE$^*$, $t$:TIME) ≡
  ...
    **endif**
    $si.sendTime := si.arrival$
**endlet**

SELECTTRANSITIONSTARTPHASE ≡
      ...
      **else**
            $Self.inport.schedule := cleanSchedule(Self.inport.schedule)$
            $Self.inputPortChecked := Self.inport.queue$
            ...
      **endif**

$cleanSchedule(siSeq$:SIGNALINST$^*$):SIGNALINST$^* =_{def}$
      **if** $siSeq = empty$ **then** $empty$
      **elseif** $siSeq.head.expiryArg = undefined \vee siSeq.head.expiryArg \geq now$ **then**
        $< siSeq.head > ^\frown cleanSchedule(siSeq.tail)$
      **else** $cleanSchedule(siSeq.tail)$
      **endif**
```

Listing 1.2. Changes of the SDL semantics

3.1 Specified Signal Arrival Time

To execute time-triggered schedules located in one or more dedicated schedulers, we propose the concept of *real-time signaling* in SDL. A *real-time signal* is an SDL signal for which an arrival time is specified when the signal is sent, with the effect that reception is postponed until this point in time. This is similar to setting a local SDL timer, which generates a timer signal on expiry. The main difference here is that real-time signals can be sent by other SDL processes. Another difference is that there is no reset operation, i.e. once a real-time signal is sent, it can not be deleted by the sender, preserving the signal character. Both normal signals and time-triggered signals are inserted in the same ordered signal queue. The ordering is based on the actual arrival times of normal signals and the specified arrival times of time-triggered signals.

To specify arrival times of signals, we add the communication constraint `at <time expression>` to the output action (see List. 1.1). In the formal SDL semantics (see List. 1.2), the specified signal arrival time is added to the subdomains of OUTPUT (TIME, line 9), and to the parameter list of the ASM macro SIGNALOUTPUT (*atArg*, line 15). When a signal is created (line 18), the specified signal arrival time is associated with the signal instance (line 23) using the shared function *atArg* (line 1). When the signal is eventually delivered to the receiving agent, as defined in the ASM macro DELIVERSIGNALS (line 33), the actual signal arrival time is the maximum of specified arrival time (*si.atArg*) and actual arrival time (*si.arrival*) as defined by the shared function *arrival*, which takes the effects of delaying channels into account (line 37). Thereby, signals are never enqueued in the past.

3.2 Signal Expiry Time

With *real-time signaling*, it can be expressed that signal delivery does not occur before the specified signal arrival time. However, there may still be an arbitrary delay until the signal is actually consumed, thereby triggering the scheduled task. On real hardware, this delay will increase during periods of high system load.

To control system load to some degree, we propose the concept of signal expiry in SDL. When a signal is sent, an expiry time can be specified, with the effect that the signal is discarded if it has not been consumed before this point in time. This is particularly useful in situations where a periodic value is delivered, where only the latest value is of interest. When used together with *real-time signaling*, a time interval where signal consumption is valid can be specified.

To specify signal expiry times, we add the communication constraint `expiry <time expression>` to the output action (see List. 1.1). In the formal SDL semantics (see List. 1.2), the specified signal expiry time is added to the subdomains of OUTPUT (TIME, line 9), and to the parameter list of the ASM macro SIGNALOUTPUT (*expiryArg*, line 15). When a signal is created (line 18), the specified signal expiry time is associated with the signal instance (line 24) using the shared function *expiryArg* (line 2). Signal expiry time is evaluated at the beginning of the transition selection phase, which is defined by the ASM macro SELECTTRANSITIONSTARTPHASE (line 48). Before the input port is frozen for transition selection (line 52), expired signal instances are removed (line 51). Removal of expired signals is formally defined by the new function *cleanSchedule* (line 56). Systems using expiry time must consider the fact of signal removal as further design aspect, since the removal of expired signals can lead to deadlocks.

3.3 Timestamping of Signals

In real-time systems, *timestamping* is used to refer to the time at which an event has been detected. If the event is reported by a signal, the timestamp can be associated with that signal. In [3], we have considered early timestamping for higher precision, adding the timestamp as signal parameter that was set when the signal was created.

Alvarez et. al. [5] proposed to associate, with each signal, timestamps recording sending and reception times, and to access them using predefined functions `time_sent` and `time_received`, respectively. We follow up on this idea and associate a timestamp with every SDL signal, denoting the time when the signal was sent. To access the timestamp, we introduce the anonymous variable `sendtime` (see List. 1.1), which yields the sending time of the last consumed input signal. In the formal SDL semantics (see List. 1.2), the sending time is defined to be the creation time of the signal (line 25), or, in case of SDL timers, the timer's specified time value (line 45), which is equal to their arrival time[4]. As stated in Sect. 3.1, *real-time signals* are considered as remote timers, hence the sending time is set to their specified arrival time ($si.atArg$, line 26).

When an ordinary SDL signal is processed in a transition, the expression `now - sendtime` can be used to determine the duration of the signal transfer and the waiting time in the signal queue. For real-time signals and local timers, it denotes the waiting time in queue only and can be evaluated at runtime.

3.4 Example of Real-Time Signaling

We now revisit the example of Sect. 2, and use the SDL extensions above. Figure 3 shows a solution that is derived from Fig. 2, where we have used only one SDL timer `Timer_S` for periodic scheduler activation. When activated, the scheduler triggers all tasks to be initiated before its next activation, by sending signals `Trigger_T` identifying task `T` to the SDL process hosting the task. Different from the solution in Fig. 2, `Trigger_T` now is a real-time signal, for which the time of arrival and expiry are specified using the communication constraints `at` and `expiry`, respectively. Compared to Fig. 2, the specification of process `P3` hosting task `T` is significantly shorter, as the timer `Timer_T` becomes obsolete – process `P3` is in fact the same as process `P1` in Fig. 1. In this solution, the SDL process hosting the task needs no adaptation.

Another improvement has been achieved by using the anonymous function `sendtime` to express a reference point in time to define arrival and expiry times (`OUTPUT Trigger_T`), and to set timers (see Fig. 3). This makes the use of time variables (variable `ref` in Fig. 2) to store reference points in time obsolete, thereby simplifying the specification of the scheduler further.

[4] Compared to the SDL standard, on timer expiry, a new signal with the same name is created and put in the input port of the agent.

We note that the solution in Fig. 3 indeed reduces the number of SDL timers, and thus runtime overhead. Compared to the previous solutions, almost all sources of delay have been eliminated; what remains is the delay of the consumption of the the real-time signal `Trigger_T`.

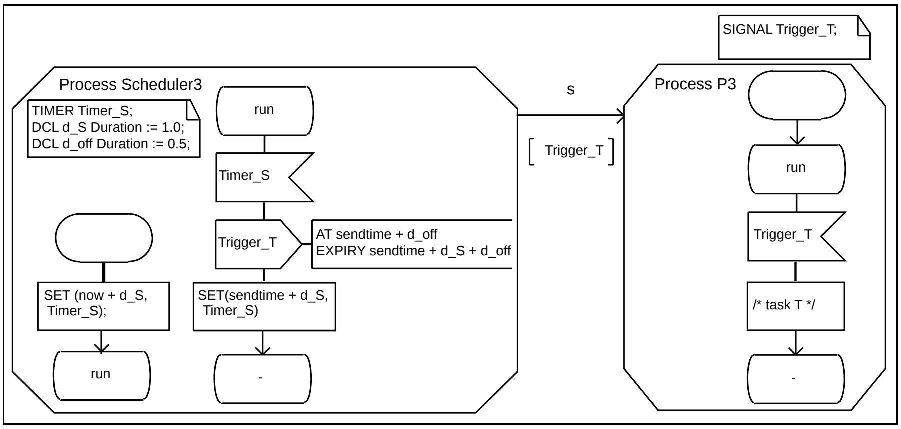


Fig. 3. Real-time signaling

4 SDL Environment Framework (SEnF)

Particularly on embedded platforms, the SDL system under execution needs to interact with the peripherals of the platform, for instance, to communicate with other nodes or to take measurements using integrated or connected sensors. With our tool chain and model-driven development approach SDL-MDD [6], we are able to automatically transform an SDL specification into runtime-independent C++-code using our code generator ConTraST [7]. Next, a platform-dependent compiler is used to obtain executable machine code for the target platform. The generated code is executed under the control of the SDL Runtime Environment (SdlRE), our implementation of the SDL Virtual Maschine (SVM) as defined in Z.100 Annex F3. Currently, SdlRE supports the PC platform[5] and the ARM-based wireless sensor node Imote2 [8] by Memsic.

To interact with peripherals, SDL processes exchange SDL signals with the environment. These SDL signals are processed by hardware-specific drivers, which are provided by our SDL Environment Framework (SEnF) [9]. The usage of signals to the environment to interface with hardware drivers avoids manual coding entirely and enables an holistic model-driven development approach for embedded platforms with SDL.

SdlRE treats the environment SEnF like any other SDL agent: SDL signals to the environment are delivered to its SDL agent and stored in its signal queue. On execution of the environment agent, all pending signals are delivered to the

[5] The PC platform uses Linux as operating system.

responsible hardware drivers by calling their input procedure. After processing a delivered SDL signal, execution control is returned to the environment agent. When all signals are processed, execution control is passed back to SdlRE. Regarding embedded platforms, most hardware drivers are based on interrupts. This means that the integrated or connected hardware components generate an interrupt, which is handled by the driver in the responsive interrupt service routine (ISR). Thereby, interrupts allow almost an immediate reaction of the driver to time-critical hardware requests, e.g., to read buffers of a transceiver early enough to prevent overflows.

To implement *real-time signaling* with SEnF accurately, we added a centralized *real-time* queue for all hardware drivers, which contains the SDL signals transmitted to the environment using *output at*. This queue is isolated from the regular signal queue, storing all SDL signals sent to the environment using standard SDL mechanisms. The *real-time* queue is sorted by the specified *real-time signaling* timestamps. A hardware timer is set to the timestamp of the earliest signal in the queue, generating an interrupt on expiration. In the ISR the signal is then removed from the queue and delivered to the responsible hardware driver by calling its input procedure. Thus the input procedure runs in the context of the ISR and therefore yields very high precision. Finally, the hardware timer is updated to the timestamp of the next signal in the *real-time* queue.

On insertions of new signals in the *real-time* queue, the configuration of the hardware timer is checked, and updated if the new signal is earlier than the current timer setting. Before a new signal is enqueued, an optional hardware driver specific callback handler is called, which can adjust the *real-time signaling* timestamp of the signal. Hence, driver-specific delays can be considered, e.g. if the driver needs additional time to communicate with a sensor and to start a measurement, the *real-time signaling* timestamp of the signal is decreased by this constant delay. Accurate *real-time* queues are currently supported on the Imote2 platform, which provides the required hardware timers and gives full control over all system resources. In our implementatin, the SdlRE acts as operating system of the platform.

The centralized *real-time* queue was chosen to achieve high precision of the *real-time signaling* mechanism for all hardware drivers. Furthermore, this yields an encapsulation and clear separation between SDL on the one hand, and driver- and hardware-related timing aspects on the other hand.

5 Evaluation

We illustrate the application of *real-time signaling* by our inverted pendulum system, a wireless networked control system (WNCS), where controllers, sensors, and actuators exchange information over a wireless digital communication network. The main challenge of a WNCS is to achieve predictable performance and stability in all possible dynamic situations. The WNCS Communication

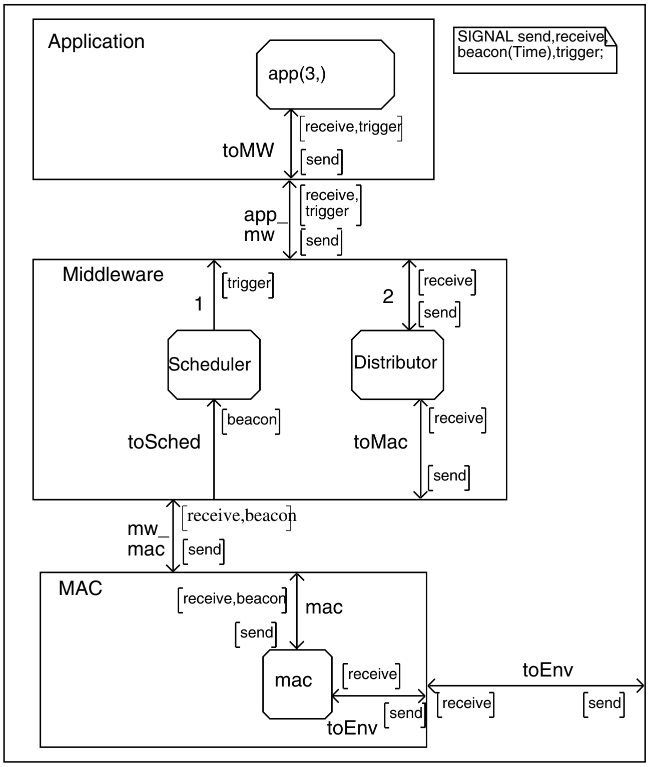


Fig. 4. Excerpt of the WNCS_CoM

Middleware (WNCS_CoM), which is part of the inverted pendulum system, follows a service-oriented approach. Sensor and actuator nodes announce their services and the minimum service time interval. Controllers subscribe to announced services, specifying a periodic update interval, so the service provider is responsible to send the data. In Fig. 4, we show an excerpt of our SDL model, consisting of `MAC`, `Middleware`, and `Application` layer.

The `MAC` layer provides medium-wide tick synchronization and TDMA-based medium access. Reference ticks are used for medium arbitration and time adjustment of the distributed scheduler in `Middleware`. The scheduler is responsible for the activation of tasks, based on current service subscriptions and the constant delay of the control application to provide measurement data or to apply actuator values.

In this inverted pendulum system, there are several places where *real-time signaling* is applied. First, there is the MAC layer, which is responsible for periodic tick synchronization and medium slotting (see Fig. 5). For resynchronization, a master node transmits a tick frame every d_{resync}, which is repeated by further network nodes. Synchronization accuracy and therefore guard times within each

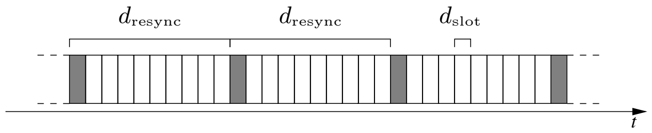


Fig. 5. MAC with TDMA

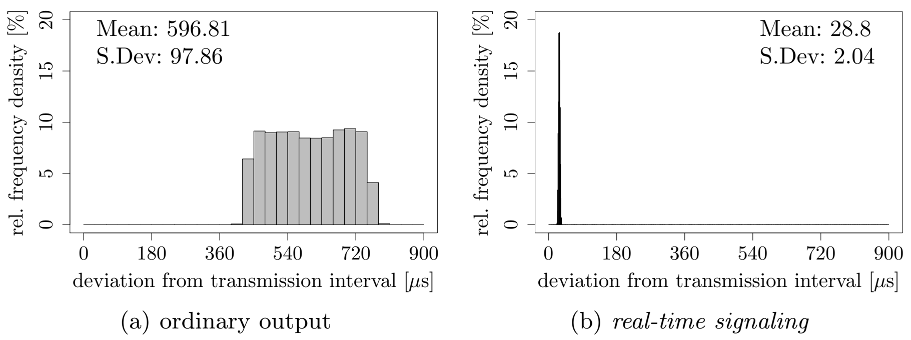


Fig. 6. Tick synchronisation using regular output and output **at** via SEnF

time slot of duration d_{slot} depend on the precise timing of tick frames. Therefore, we use real-time signals to schedule tick frame transmissions, and measure the deviation of actual transmissions from the specified transmission interval. For comparison, we repeat these measurements, using regular SDL timers and signals. The results are summarized in Fig. 6.

All measurements are performed on an Imote2 sensor node [8], using ConTraST [7]. Each experiment consists of 6000 measured values, where d_{resync} is set to 100 ms. In Fig. 6(a), the difference between the planned transmission time (the time when the SDL timer expires) and the actual transmission time is shown. We measure delays from $\sim 400\,\mu$s up to $\sim 800\,\mu$s. In comparison, we show the measurements of the same experiment, now using *real-time signaling* (cf. Fig. 6(b)). Since the signal is transferred early to the destination process (environment), the signal is already in the input queue when it has to be processed. Therefore, it can be processed almost instantly at the specified arrival time (see Sect. 4), which yields a significantly smaller average value of only $29\,\mu$s. The measured maximum delay is only slightly higher at about $35\,\mu$s, due the use of a hardware interrupt routine and other non-interruptible routines. Compared to the use of regular SDL signals, there is an improvement by a factor of 22.

Another place to apply *real-time signaling* in the inverted pendulum system is the scheduling of the control application `Application`. For a given schedule (cf. Fig. 7), the scheduler can send the activation signal to the application in advance, which reduces the delay upon task execution. As before, we perform measurements to compare the processing delays of regular SDL signals to real-time signals. We define a schedule for 3 applications (as shown in Fig. 7) on the same node, with an expiry time of 100 ms. The upper schedule was specified

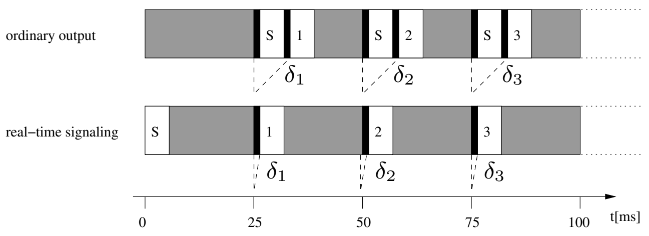
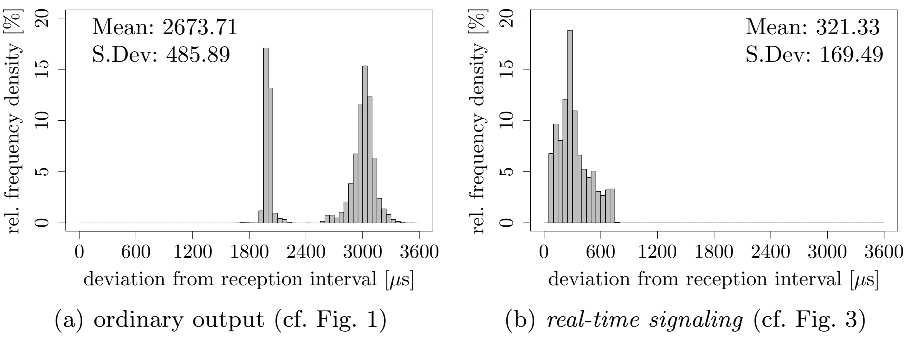


Fig. 7. Control application schedules

using regular SDL signals as shown in Fig. 1. Here, the `Scheduler` (S) has to be activated just before the task `Application` (1,2,3) to be scheduled. In case of *real-time signaling* (below), the scheduler is only activated once per period as shown in Fig. 3. For each application i, the difference δ_i between the planned and actual execution time is measured.



(a) ordinary output (cf. Fig. 1)

(b) *real-time signaling* (cf. Fig. 3)

Fig. 8. Scheduler using output and output **at**

In Fig. 8(a), we show the deviation of the applications between the planned and the actual execution time. The deviation for regular SDL signals ranges from 1600 μs to 3600 μs. In comparison, *real-time signaling* shown in Fig. 8(b) has only a deviation between 0 μs and 800 μs – this is only $\frac{1}{4}$th of the time of the ordinary mechanism.

6 Related Work

SDL's limited real-time expressiveness and its high degree of implementational freedom have been addressed by quite a number of authors over the past two decades. Although common objectives of all proposals are the improvement of predictability and the handling of overload situations, various approaches can be found, reaching from extensions to SDL on design level (like [5,10,11,12]) to implementation-specific realizations of the SDL semantics (like [13,14,15]). However, to our knowledge, there is no approach based on time-triggered execution of SDL systems. In the following, we restrict the survey on approaches extending the notion of SDL timers and signals, and proposals covering the scheduling of agents.

In [10], the authors introduce extensions to SDL's notion of time progress on specification level. The presented approach targets at simulations and is motivated by timed automata with urgencies. In particular, it is suggested to classify transitions as eager (time must not progress), lazy (time may progress with indefinite amount), and delayable (time may progress within given bounds). Although delayable timers seem to be similar to our approach of signal arrival and expiry, the implication is completely different, since our objective is not the control of simulation time, but the control of system load on real hardware. In [11], the authors extend their approach by introducing cyclic and interruptive timers, and present further annotations such as periodicity of timers or execution delay of tasks, also for simulative purpose only.

In [12], Gotzhein et al. present the concept of input port bounds, an extension to SDL to avoid queue overflows that can cause illegal behavior on concrete limited hardware platforms. Similar to the signal expiry, their approach makes SDL systems more robust against system overload at the cost of less reliable signal transfer within SDL systems. In contrast to the approach in this paper, the number of signals (not an expiry time) is the criterion for removal of signal instances. In addition, they distinguish between three perceivable solutions of treating incoming signals in case of input port overflows (discard, replace, delete/append).

Besides many topics regarding the efficient implementation of SDL, Mitschele-Thiel discusses the handling of limited input queues on implementation side in [13]. He presents four alternatives to handle input queue overflows: Prevention (with hard assumptions on the system's environment), removal of signals (similar to [12]), blocking of the signal's sender (with the risk of deadlocks), and dynamic control (detecting overload situations and throttling the load). With dynamic control, Mitschele-Thiel introduces also a design-based solution to handle overload situations, but in contrast to expiry times of signals, his solution generates additional system load in situations in which high load already exists.

Similar to the predefined variable `sendtime` (see Sect. 3.3), Álvarez et al. [5] introduce a function called *time_ sent*, holding the point in time when an SDL signal has been created. In addition, they propose a function *time_ received*, recording the time when the signal is consumed in a transition. Since our SVM provides non-preemptive scheduling, we have no need for such a function. Álvarez et al. also discuss the problem of timely transition activations. However, their approach is based on local SDL timers and transition priorities, and not compliant to the SDL semantics regarding to the order of signal receptions.

The idea of a centralized scheduler in SDL was realized in [16] in terms of a CPU scheduler simulator for educational purposes. The presented scheduler supports a plenty of scheduling strategies (such as FCFS, SJF, and EDF); however, it simulates operating system processes only and is not intended for the scheduling of SDL systems.

To test schedulability of SDL systems, several approaches with off-line analysis have been proposed. Many of those approaches apply a transformation of the SDL specification to a different formal representation (such as analyzable task networks [17], methods from queueing theory [18], and timed automata [19]). Since the mentioned proposals perform analyses without enforcing an appropriate scheduling of agents, we will not go into further detail.

7 Conclusions and Future Work

In this paper, we have introduced *real-time signaling* in SDL, by extending SDL to specify time intervals for the reception and expiry of ordinary signals, and by associating send times with signals. This extension can be used, for instance, to specify timer-triggered scheduling, which is required in many real-time systems. Our contribution covers syntactical extensions of SDL, their formal semantics, their implementation in the SDL Runtime Environment (SdlRE) and the SDL Environment Framework (SEnF), and their application in case studies. Finally, we have reported on measurement results which have revealed improvements by a factor of up to 22. We have successfully applied the introduced concepts to a realtime communication system for an inverted pendulum [20].

In future work, we will investigate various ways to further improve the real-time expressiveness of SDL. In particular, we will exploit the degrees of freedom offered by the SDL Virtual Machine (SVM) as part of the formal SDL semantics. For instance, to achieve predictability, we need more control of the timing behaviour including the load situation. This requires a priori knowledge on worst-case execution times, and the possibility to suspend non-real-time processes from execution.

References

1. Kopetz, H.: Real-Time Systems – Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers, Dordrecht (1997)
2. International Telecommunication Union (ITU): ITU-T Recommendation Z.100 (11/2007): Specification and Description Language (SDL) (2007), `http://www.itu.int/rec/T-REC-Z.100-200711-I`
3. Becker, P., Christmann, D., Gotzhein, R.: Model-Driven Development of Time-Critical Protocols with SDL-MDD. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009. LNCS, vol. 5719, pp. 34–52. Springer, Heidelberg (2009)
4. International Telecommunication Union (ITU): ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition (2000)
5. Álvarez, J.M., Díaz, M., Llopis, L., Pimentel, E., Troya, J.M.: Integrating Schedulability Analysis and Design Techniques in SDL. Real-Time Systems 24(3), 267–302 (2003)
6. Gotzhein, R.: Model-driven with SDL – Improving the Quality of Networked Systems Development. In: Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco, pp. 31–46 (2007) (invited paper)

7. Fliege, I., Grammes, R., Weber, C.: ConTraST – A Configurable SDL Transpiler and Runtime Environment. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 216–228. Springer, Heidelberg (2006)
8. Memsic: Imote 2 datasheet, `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134`
9. Fliege, I., Geraldy, A., Jung, S., Kuhn, T., Webel, C., Weber, C.: Konzept und Struktur des SDL Environment Framework (SEnF). Technical Report 341/05, TU Kaiserslautern (2005)
10. Bozga, M., Graf, S., Kerbrat, A., Mounier, L., Ober, I., Vincent, D.: SDL for Real-Time: What is Missing? In: Sherratt, E. (ed.) SAM, VERIMAG, IRISA, SDL Forum, pp. 108–121 (2000)
11. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.-L., Vincent, D.: Timed Extensions for SDL. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001)
12. Gotzhein, R., Grammes, R., Kuhn, T.: Specifying Input Port Bounds in SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 101–116. Springer, Heidelberg (2007)
13. Mitschele-Thiel, A.: Engineering with SDL – Developing Performance-Critical Communication Systems. John Wiley & Sons, Chichester (2000)
14. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. Prentice Hall, Englewood Cliffs (1993)
15. Sanders, R.: Implementing from SDL. In: Telektronikk 4.2000, Languages for Telecommunication Applications. Telenor (2000)
16. Rodríguez-Cayetano, M.: Design and Development of a CPU Scheduler Simulator for Educational Purpose Using SDL. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 72–90. Springer, Heidelberg (2011)
17. Kolloch, T., Färber, G.: Mapping an Embedded Hard Real-Time Systems SDL Specification to an Analyzable Task Network - A Case Study. In: Müller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 156–165. Springer, Heidelberg (1998)
18. Diefenbruch, M., Hintelmann, J., Müller-Clostermann, B.: QUEST Performance Evalution of SDL System. In: Irmscher, K., Mittasch, C., Richter, K. (eds.) MMB (Kurzbeiträge), TU Bergakademie Freiberg, pp. 126–132 (1997)
19. Ober, I., Kerbrat, A.: Verification of Quantitative Temporal Properties of SDL Specifications. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 182–202. Springer, Heidelberg (2001)
20. Chamaken, A., Litz, L., Krämer, M., Gotzhein, R.: Cross-layer design of wireless networked control systems with energy limitations. In: European Control Conference 2009, ECC 2009 (2009)