SDL Real-Time Tasks – Concept, Implementation, and Evaluation

Dennis Christmann, Tobias Braun, and Reinhard Gotzhein

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

Abstract. *Real-time tasks* are a concept used in real-time systems to structure and schedule execution, in order to handle load situations, and to meet deadlines. In previous work, we have transferred this concept to the Specification and Description Language (SDL), by incorporating the notion of real-time task into SDL's formal syntax and semantics. More specifically, we have defined an SDL real-time task as a set of transition executions, which may span different SDL processes and are ordered by a strict partial order with a least element. In this paper, we extend this concept by the notion of *distributed* real-time task, which may span SDL processes of different SDL systems, thereby supporting tasks executed on several nodes. In addition, we introduce the notion of task types, which support task multiplexing in SDL processes. We then outline our implementation of real-time tasks in our SDL tool chain, consisting of the SDL transpiler ConTraST, the SDL Runtime Environment (SdlRE), and the SDL Environment Framework (SEnF). To evaluate the gain in real-time performance, we have devised an SDL specification of an Adaptive Cruise Controller taken from the automotive domain, and have executed it on an Imote2 hardware platform. The results clearly show that task-based scheduling outperforms ordinary and priority-based scheduling in terms of processing delays and reaction times to critical events.

1 Introduction

The Specification and Description Language (SDL) [1] has been devised as a formal design language for distributed systems. Yet, due to its notion of time (`now`) and its timer mechanism, it also provides expressiveness to specify certain aspects of real-time systems. To broaden this expressiveness, we have proposed, defined, implemented, and evaluated several language extensions, in particular SDL real-time signals [2] and SDL process priorities [3]. These extensions have proven valuable to enhance the predictability of networked control systems, which we have developed in a model-driven way with SDL as design language [4].

To further enhance the real-time capabilities of SDL, we have considered the concept of *real-time task* (or *task* for short), which is used in real-time systems to structure and schedule executions, in order to handle load situations and to meet deadlines. Tasks are code unit executions, and may be initiated dynamically when a significant change of state occurs (event-triggered) or at determined points in time (time-triggered). In the context of SDL, these code unit executions may be structured into several ordered SDL transition executions associated with one or more SDL processes. After some consideration, we came to the conclusion that our previous extensions, i.e., SDL real-time signals and SDL process priorities, were not sufficient to express real-time tasks in SDL. Therefore, we have devised further language extensions, which were necessary to enable real-time tasks in SDL [5]. Thus, we have established the notion of real-time task[1] in SDL's formal syntax and semantics.

In this paper, we continue our previous work conceptually and, in particular, by presenting the implementation and evaluation of SDL real-time tasks. Conceptually, we introduce the notion of *distributed* real-time task, which may span several SDL processes of *several* SDL systems (Sect. 2). In practical situations, this means that several nodes may be involved in the completion of a given real-time task. In addition, we propose *task types*, which can be used to naturally specify task multiplexing in SDL processes.

The focus of this paper, however, is on the implementation and evaluation of SDL real-time tasks and task scheduling. In Sect. 3, we outline the implementation in our SDL tool chain [4], which consists of the SDL transpiler ConTRaST, the SDL Runtime Environment SdlRE, and the SDL Environment Framework SEnF. In Sect. 4, we present extensive experimental results showing the gain of SDL real-time tasks and task scheduling w.r.t. the predictability of reaction times compared to existing scheduling approaches. To run these experiments, we have specified an Adaptive Cruise Controller (ACC) taken from the automative domain with SDL, and have executed it on an embedded hardware platform. From the experiments, it is obvious that real-time task scheduling outperforms existing scheduling strategies, in particular with increasing system load, and substantially improves the predictability of reaction times. The paper is completed by a survey of related work (Sect. 5) and conclusions (Sect. 6).

2 Distributed SDL Real-Time Task

In this section, we extend our previous work [5] by introducing the concept of distributed real-time tasks in SDL (see Sect. 2.1). With the extension, SDL tasks can not only be used to group functionality-related behavior of a single SDL system, but also to identify and prioritize behavior spanning several network nodes. To incorporate distributed real-time tasks in SDL, several language extensions are presented in Sect. 2.2.

2.1 Concept of SDL Real-Time Task

The formal definition of real-time task is based on a set of transition executions and a strict partial order with a least element, i.e., each real-time task starts

[1] Not to be confused with the existing notion of task in SDL, which is a sequence of statements.

with a single transition execution. Real-time tasks are dynamic in the sense that the set of transition executions is determined at run-time and may depend on the internal state of the system, that is, for instance, the current time or states of SDL processes. They terminate after all transition executions are finished. Let N be the set of network nodes. Then, a real-time task is defined as follows.

Definition 1. *A* ***real-time task*** *τ is a tuple (id_τ, $T_e(\tau)$, f_{prio}, f_{node}, $<_{eo}$), where id_τ is a globally unique task id, $T_e(\tau)$ is the set of transition executions, $f_{prio} : T_e(\tau) \rightarrow \mathbb{N}$ is a function assigning a priority to each transition execution, $f_{node} : T_e(\tau) \rightarrow N$ is a function to allocate each transition execution to a network node, and $<_{eo} \subsetneq T_e(\tau) \times T_e(\tau)$ is an execution order on $T_e(\tau)$ with following properties:*

- *$<_{eo}$ is a strict partial order, i.e., $<_{eo}$ is irreflexive, transitive, and antisymmetric*
- *$\exists t_e \in T_e(\tau).\forall t'_e \in T_e(\tau).(t'_e \neq t_e \Rightarrow t_e <_{eo} t'_e)$, i.e., there is a least element defining the starting point of the task, which is the first transition execution.*

We note that the definition allows concurrent transition executions within a real-time task, if they are not ordered by $<_{eo}$. A real-time task may be non-terminating, if its set of transition executions is infinite. Thereby, a real-time tasks τ may consist of cyclic executions of transitions, since executing the same transition multiple times results in different transition executions, i.e., different entries in $T_e(\tau)$. An example for such a task is the periodical calculation of control values. A real-time task itself is non-recurring, i.e., it is executed only once.

The definition of real-time tasks so far covers node-local and node-spanning tasks. Based on the general definition, we define distributed tasks as follows.

Definition 2. *A* ***distributed SDL real-time task*** *τ_{dist} is a real-time task, for which the image of f_{node} contains at least two distinct elements:*

$$\exists t_1, t_2 \in T_e(\tau_{dist}) : f_{node}(t_1) \neq f_{node}(t_2).$$

To fulfill this definition, there must be at least two SDL systems deployed on different network nodes, each running at least one transition execution of the task. Thereby, distributed behavior, such as an Request-to-Send/Clear-to-Send (RTS/CTS) handshake or a distributed leader election, can be expressed as a single task.

The definition of distributed SDL real-time task does not model communication explicitly. Instead, consecutive transition executions of different nodes are ordered by the execution order only, and the required synchronization and value passing is left to the environment of the SDL system. A distributed SDL real-time task may run on several nodes in parallel, if there is suitable concurrency according to the execution order $<_{eo}$. Otherwise, the transitions of the distributed SDL real-time task are processed successively by the nodes.

The distinction between SDL transitions as code units and transition executions as execution units results in a very generic definition of real-time task. Some properties of this definition are:

- *Loose temporal ordering*: The definition does not make a statement on the time between transition executions.
- *Flexible activation paradigm*: Every transition execution can be event- as well as time-triggered. In particular, a real-time task can be temporarily suspended or wait for the activation of the next transition execution trigger.
- *Transition repetitions*: The same transition can be executed by one task several times, particularly with different priorities.
- *Priority-independent transition definitions*: A transition as code unit has no (static) priority assigned. Instead, the priority is (dynamically) associated with its execution, thereby allowing the same transition to be executed with different priorities.
- *Transition sharing*: Several real-time tasks may execute the same transition.

Though a real-time task is nonrecurring, it usually describes actions to execute a recurring system task like the response to a specific event that is observed by the environment of the node. To enable the association of a real-time task with the system task it fulfills, the notion of *task type* is introduced. During a transition execution, the information on the task type is, for instance, helpful for task multiplexing within the system or for changing the priorities of future transition executions. The relation between SDL task and task type has an analogy in object-oriented programming languages, because a real-time task τ (object) states the execution (instantiation) of a task type (class), and multiple executions of a task type result in multiple real-time tasks with distinct task (object) identifiers id_τ. Formally, the relation between real-time tasks and task types is as follows: Assuming τ is a real-time task and Γ is the set of all task types. Then, there is a function f_{type} with $f_{type}(\tau) \in \Gamma$ returning the task type of the real-time task.

2.2 Language Extensions to SDL

In [5], real-time tasks have been incorporated in SDL's syntax and semantics. By dynamically associating transition executions with task attributes, consisting of task id id_τ and a priority, a transition runs in the context of the real-time task τ. To accommodate task types, task attributes are now extended by $f_{type}(\tau)$. The task attributes are transported by SDL signals (so-called task signals), which are used to trigger task execution. Consuming a task signal transfers the task attribute to the triggered transition execution. The priorities given by the task attributes have implications on the transition selection order, such that task signals are consumed according to their priority and have additionally preference over plain SDL signals.[2] Thereby, transition executions of time-critical real-time tasks take precedence over all other transition executions.

[2] There is an exception for this rule, if the signal with highest priority is saved in the current process state.

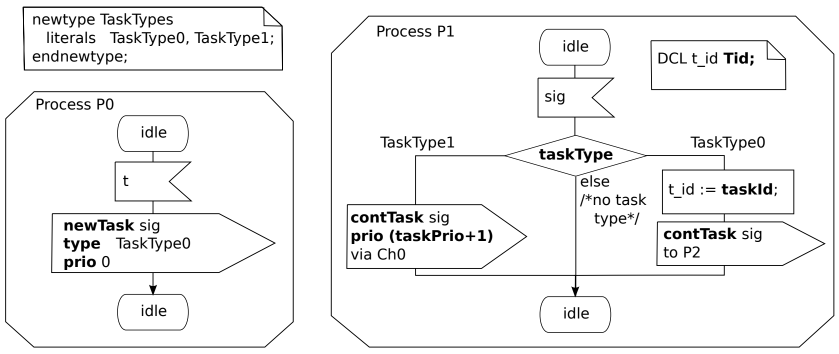


Fig. 1. Example showing the use of real-time tasks in SDL. Bold characters state new keywords/operators

Due to the consideration of priorities, transitions may be executed in a different order compared to standard SDL. Thus, existing tools – like tools performing reachability analysis to find deadlocks or implicit consumptions – must be extended to consider priorities. Since transition priorities are included in each signal's task attributes, all required information is, however, available in the SDL system and analysis of the system specification is still possible. This is even an additional benefit compared to many implementation methods introducing priorities in a separate implementation phase, because with such approaches, priorities are not available for system analysis on design level.

To control the execution of real-time tasks in SDL specifications, [5] presents several syntactical extensions, in particular, regarding task creation and forking (continuation of an existing task). Additionally, a new data type to store task ids (**Tid**) has been introduced together with a function returning the task id of the current task. To support task types and relative changes of priorities, the syntax has been extended with functions returning type and priority of the current task.

The example in Fig. 1 presents the use of real-time tasks in SDL: In process `P0`, a new task is created by the output of signal `sig`. This new real-time task is scheduled with task priority 0 and is of type `TaskType0`. In `P1`, task execution starts by consuming the task signal and by identifying the task type by means of the **taskType** operator. Because the task type is `TaskType0`, the right branch is taken, i.e., the transition stores the task id in the variable *t_ id* and triggers the next transition execution of this task by sending `sig` to process `P2` (not shown). Because no priority is provided, the task is continued with priority 0. The figure only shows excerpts of SDL processes highlighting language elements that have been extended to enable the control and processing of real-time tasks in SDL. In a real scenario, there are usually further application-related actions in the transitions' bodies (see also Sect. 4.1).

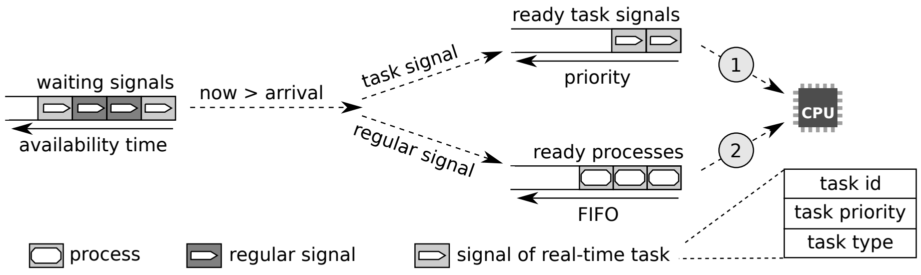


Fig. 2. Schematic outline of the task scheduler implementation

3 Implementation of Real-Time Tasks

The concept of real-time tasks has been implemented in our SDL tool chain, which supports an embedded ARM platform, Linux/PC, and various simulators. The tool chain consists of three main components: The code generator ConTraST, the SDL Runtime Environment (SdlRE), and the SDL Environment Framework (SEnF) providing interfaces and drivers of the SDL environment. To support real-time tasks, changes to all components were necessary.

Though we would prefer the extension of SDL's concrete syntax to control real-time tasks (see Sect. 2), our implementation is based on annotations, thereby allowing the re-utilization of the graphical editor and analyzer of IBM's Rational SDL suite [6]. When generating C++ code, ConTraST analyzes the real-time task annotations in SDL/PR and generates relevant C++ instructions.

To support and schedule real-time tasks during system runtime, SdlRE has been extended by task signals and a non-preemptive scheduler realizing the task scheduling strategy (short: Priorities$_{tasks}$). Additionally, SdlRE provides an implementation of the **Tid** datatype and an interface to access the task id, priority, and task type of the transition that is currently executed.

Task signals are implemented by extending the existing SDL signal class with task attributes, i.e., task id, priority of the triggered transition execution, and task type. Further information of the real-time task, such as the node executing the transition, is implicitly available and not stored explicitly.

Different to Sect. 2, where task priorities affect the transition execution order of SDL processes[3] only locally, Priorities$_{tasks}$ enforces priorities system-wide. A schematic overview of Priorities$_{tasks}$ is presented in Fig. 2. In total, the scheduler operates on three global queues: A queue holding signals with future arrival times (e.g., timers), a queue with task signals sorted by task priorities, and a queue with runnable processes. When searching for the next transition to be executed,

[3] Due to their background of Abstract State Machines, the dynamic semantics of SDL-2000 is based on different types of agents [7]. Thus, to be precise, we would have to use the notion of agents when referring to the execution of an SDL system. Nevertheless, we use the term SDL process in the rest of the paper, because all schedulers of SdlRE affect the scheduling of agents that evolve from SDL processes.

the process holding the first consumable signal in the queue of task signals runs to fire the corresponding transition. If there is no such signal, the first process of the process queue is dispatched to execute one transition that is either a transition consuming a regular signal or a continuous signal.

Following our annotation-based approach, the runtime environment can be configured to use the task scheduling strategy Priorities$_{tasks}$ by annotations in the head symbol of the system. Besides Priorities$_{tasks}$, our tool chain supports the following non-preemptive strategies:

– *Signal-based First-Come-First-Served* strategy (short: FCFS$_{signals}$)
  The transition execution order of FCFS$_{signals}$ is determined by the arrival times of the triggering signals. For this purpose, a global First-In-First-Out (FIFO) queue of SDL signals is maintained. When searching for the next transition to be executed, the first consumable signal in the queue is taken.
– *Process-based First-Come-First-Served* strategy (short: FCFS$_{process}$)
  FCFS$_{process}$ is also based on a FIFO queue, but, different to FCFS$_{signals}$, the queue is filled with processes. Thus, a process with several signals in its inport, is scheduled only once. The process at the front of the FIFO queue is executed as long as it has firable transitions, thereby reducing the overhead of the scheduler compared to FCFS$_{signals}$.
– *Process Priority Scheduling* (short: Priorities$_{process}$)
  In [3], Priorities$_{process}$ has been introduced in order to privilege time-critical SDL processes. Static priorities are assigned to processes in the SDL specification, where lower values represent higher priorities. The scheduling strategy works on a queue of executable processes, which is sorted by their priorities.

Due to the non-preemptive character of all strategies, there is in general a non-zero waiting delay for all schedulers, increasing in particular reaction times in systems with long-running transitions. However, independent of the scheduling strategy, such transitions should already be avoided by design rules, because they always delay other transition executions of the same SDL process.

To support distributed SDL real-time tasks, the environment (implemented by SEnF) has been extended, such that task attributes are attached to outgoing data, e.g., to CAN messages, before leaving the node. By extracting task attributes from received data and adding them to generated SDL signals, the environment continues existing real-time tasks on the local node. Thereby, the SDL runtime environment can treat incoming signals according to their priority and task type, though the continuation of the real-time task is transparent to the system.

A special element of a task attribute is the task id. To guarantee its uniqueness also in case of distributed real-time tasks, further measures became necessary. In our implementation, the uniqueness is ensured by composing task ids of the node's id and a locally unique identifier, which is incremented each time a new task is generated.

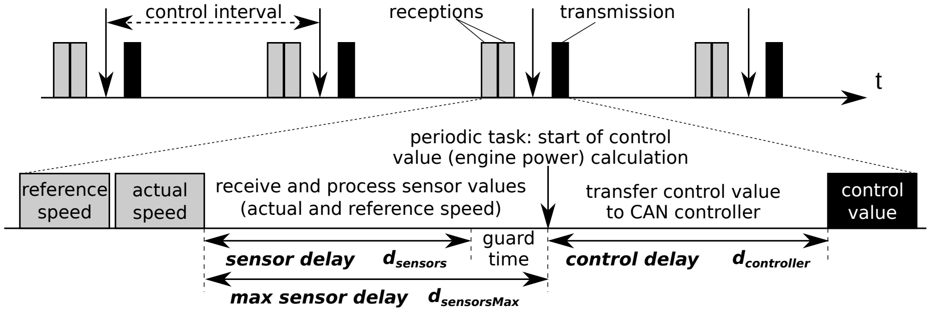

control interval
receptions
transmission
t
periodic task: start of control value (engine power) calculation
reference speed
actual speed
receive and process sensor values (actual and reference speed)
transfer control value to CAN controller
control value
sensor delay $d_{sensors}$
guard time
control delay $d_{controller}$
max sensor delay $d_{sensorsMax}$

Fig. 3. Message schedule of the node hosting the PID controller in the ACC scenario

4 Evaluation of Real-Time Tasks in a Control System

In this section, we present experimental results of our SDL task implementation. To assess its impact, we compare the task scheduling strategy with standard schedulers of SDL implementations as well as an SDL process priority scheduler. The scenario evaluates a network node that is connected to a Controller Area Network (CAN) bus and hosts an Adaptive Cruise Control (ACC), a realistic scenario from the automotive domain. An ACC is an enhanced cruise control system focused on retaining a reference speed against disturbance variables such as the current gradient or aerodynamic resistance. In contrast to a simple cruise control, a radar sensor is used to detect the distance to obstacles in front of the car. Depending on the speed of and distance to the obstacles, the reference speed is adjusted to keep a minimal safety distance or an emergency braking is initiated. Our realization of an ACC uses a Proportional-Integral-Derivative (PID) controller to minimize the difference between desired and actual speed.

An abstract schedule of the ACC is shown in Fig. 3. The ACC periodically (every 20 ms) calculates new control quantities, which are sent via CAN bus to the engine control unit. The duration to calculate and transfer the control value to the CAN controller is given with $d_{controller}$ in the figure. For correct operation of the controller, control values must be calculated and transferred on schedule with low delay, and sensor values of the reference and actual speed must arrive at the PID controller on time, taking the processing delay of the system into account. In Fig. 3, the sensor delay in the system, i.e., the duration between reception of the last sensor value at the CAN controller and the updating of the values at the PID controller process, is denoted by $d_{sensors}$. Because this delay may vary, a maximal sensor delay $d_{sensorsMax}$ has to be considered in the schedule. The best quality of control is achieved, if the sensor values are as new as possible, i.e., if the processing delay of the node generating the sensor value, the communication delay, and $d_{sensors}$ is small and almost constant, allowing a small $d_{sensorsMax}$ before the periodic control task. In addition to the periodic speed values, the node also receives sporadic radar messages that are used to keep an adequate distance to other objects by correcting the controlled speed and/or by enforcing to brake. To test the system under different workloads, additional sporadic load messages are sent to the system.

Since the scenario is based on a networked system, a comprehensive analysis must consider delays that are introduced by all nodes as well as the network itself. Because the system designer must consider the worst case when planning the global schedule, all delays must not only be low but also free of large jitter. This, in particular, demands high requirements to scheduling decisions in cases of secondary system load, which must be deferred on behalf of relevant system tasks. In a first step, this requires the assessment of delays at each single network node. This evaluation focuses on the impact of SDL schedulers on the behavior of the node hosting the PID controller.

4.1 Evaluation Setup

Hardware. To obtain reliable and reproducible results, all experiments ran on an Imote2 node, an embedded hardware platform that can be linked to various peripherals and communication technologies. E.g., in [8], a FlexRay [9] communication controller is connected to the Imote2 via Serial Peripheral Interface (SPI). The Imote2 is equipped with 256 kB SRAM, 32 MB SDRAM, and 32 MB flash ROM. Its processor is based on an ARM architecture providing up to 416 Mhz. Due to energy aspects, the processor frequency was fixed to 104 Mhz in all experiments. Since our implementation on the Imote2 is a bare implementation (without further operating system), SEnF and SdlRE have full control over the system's execution and interrupts. Therefore, all measured times can be attributed to the execution of the SDL system and its runtime environment.

Because the experiment's objective is not the evaluation of the communication technology, we did not use a real CAN bus. Instead, we simulated all CAN events taking the minimal interarrival time of CAN messages into account. As a side effect, this approach avoids distortion of results due to communication errors. To additionally avoid faulty measurements, results of experimental runs were stored in the local memory of the node and transferred to a PC via UART after the end of the run. Thus, the measurement overhead is minimized and uniform for all evaluated scheduling strategies.

System Under Test. The SDL system used in the evaluations is shown in Fig. 4 and consists of four blocks. The `CAN` block is the interface to the environment and contains two processes. The `CANMac` process converts between CAN identifiers and internal event expressions. `ConcatCoder`, on the other hand, encodes and decodes the data of CAN messages into SDL types. On top of the `CAN` block are two blocks (`Speed` and `Distance`) that are part of the cruise control. The third block `Load` processes background load that is stimulated by messages from the environment. I.e., load messages are forwarded by `CANMac` and `ConcatCoder` before arriving at the `Load` block, in which they trigger further transition executions. Because the origin of load is in the system's environment, the generation of additional load is independent of the SDL system. By changing the average frequency of load messages, different load situations are emulated.

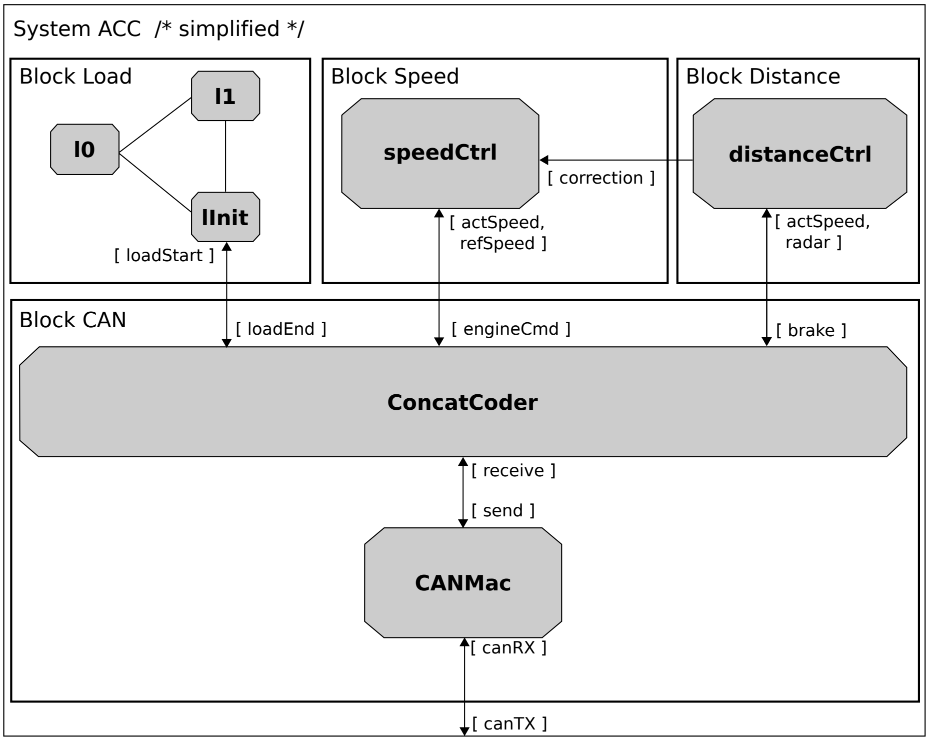


Fig. 4. SDL specification of the evaluated Adaptive Cruise Control system

In the evaluations, the system receives four different types of CAN messages, which are sent as `canRX` signals by the environment to process `CANMac`. After forwarding them through the `CAN` block, they are delivered to their responsible SDL processes. In the case of task scheduling, task types can be used in `ConcatCoder` to determine the target process of a signal. The CAN messages with the actual and reference speed are sent to the processes `speedCtrl` and `distanceCtrl`, in which they are received as SDL signals `actSpeed` or `refSpeed`. On the other hand, the radar and load messages are delivered as SDL signals `radar` or `loadStart` to process `distanceCtrl` and `lInit` respectively. CAN messages sent by the system are received by the SDL environment as `canTX` signals. They are either engine and brake control values, or load information. The engine control values are periodically calculated by `speedCtrl` and initially sent as SDL signal `engineCmd`. Brake control values and load information are generated reactively as responses to `radar` or `loadStart` signals and have their origins in `distanceCtrl` and `lInit`. `distanceCtrl` also creates a `correction` signal, which is considered by `speedCtrl` to calculate the engine control values.

The presented SDL system is executed with four different scheduling strategies (see Sect. 3). The priorities used in experiments with Priorities$_{process}$ are given in Tab. 1. They are assigned such that the environment obtains highest priority and processes in the `Load` block have lowest priority. For the task scheduling strategy Priorities$_{tasks}$, Tab. 2 summarizes the task types of the system, their priorities, and the affected signals. Tasks with sources or destinations in italics are distributed tasks and include communication via CAN bus. By deriving task types and task ids from received CAN messages, the environment continues existing tasks in the evaluated system, considering their privileges as well. On the other hand, before transmitting CAN messages, task attributes are appended to the messages.

Table 1. Process priorities used by Priorities$_{process}$

processes	**priority**
CANMac, ConcatCoder	3
speedCtrl	2
distanceCtrl	1
lInit, l0, l1	4
environment	0

The control of real-time tasks in the ACC system is illustrated by means of an exemplary excerpt of process `speedCtrl` in Fig. 5. The figure shows the start transition and four transitions that are executed in the context of real-time tasks. As discussed in Sect. 3, the control of real-time tasks is specified by annotations to be compatible with the graphical tool and analyzer of IBM's Rational SDL suite [6]. In the process, a periodical real-time task computing new engine control values is created with priority 3 in the start transition. This task is processed by executing the transition consuming `controlTimer`. In this transition, a further real-time task is created propagating the new engine control value. The calculation of engine control values takes the reference speed, the current speed, and a correction value given by the distance controller into account. Each of these values is received in a separated transition. Though the executions of the transitions receiving these values are part of a corresponding real-time task, no task control actions are specified, because the real-time tasks end after receiving the values.

Table 2. Task types of the system. Sources and destinations in italics state processes on other nodes.

task type	**source**	**destination**	**priority**	**task signals**
reference speed	*refSpeedInput*	speedCtrl	4	canRX, receive, refSpeed
actual speed	*actSpeedSensor*	ConcatCoder	4	canRX, receive
	ConcatCoder	speedCtrl	4	actSpeed
	ConcatCoder	distanceCtrl	5	actSpeed
speed control value	speedCtrl	speedCtrl	3	controlTimer
engine regulation	speedCtrl	*engineCtrl*	3	engineCmd, send, canTX
radar	*radarSensor*	distanceCtrl	2	canRX, receive, radar
collision avoidance	distanceCtrl	*brake*	1	brake, send, canTX
	distanceCtrl	speedCtrl	4	correction
load	*loadSimulator*	*loadSimulator*	8	canRX, receive, loadStart loadEnd, send, canTX

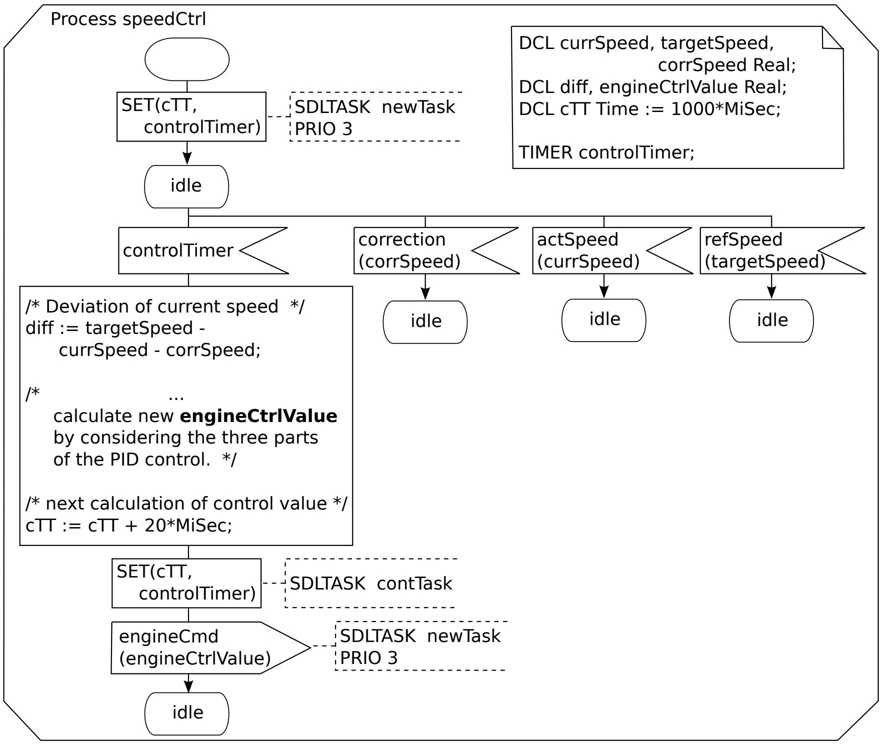
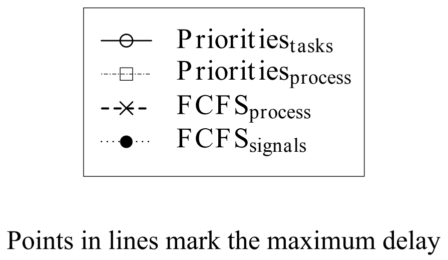

Process speedCtrl
DCL currSpeed, targetSpeed, corrSpeed Real;
DCL diff, engineCtrlValue Real;
DCL cTT Time := 1000*MiSec;
TIMER controlTimer;
SET(cTT, controlTimer)
SDLTASK newTask PRIO 3
idle
controlTimer
correction (corrSpeed)
actSpeed (currSpeed)
refSpeed (targetSpeed)
idle
idle
idle
/* Deviation of current speed */
diff := targetSpeed - currSpeed - corrSpeed;
/* ... calculate new engineCtrlValue by considering the three parts of the PID control. */
/* next calculation of control value */
cTT := cTT + 20*MiSec;
SET(cTT, controlTimer)
SDLTASK contTask
engineCmd (engineCtrlValue)
SDLTASK newTask PRIO 3
idle

Fig. 5. Excerpt of `speedCtrl` showing the usage of real-time tasks exemplarily

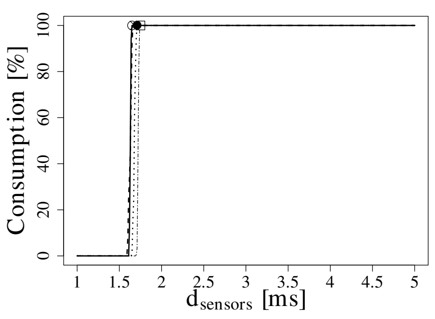
4.2 Evaluation Results

In two series of experiments, the evaluations focus on three delays: Sensor delay $d_{sensors}$, control delay $d_{controller}$, and reaction delay $d_{reaction}$ to radar messages. The sensor and control delays are evaluated in the first series consisting of 125 runs for each scheduling strategy. In each run of this series, in which no radar messages are used, 200 sensor values (reference and actual speed) are received by the system and 100 new control values are calculated and sent. The second series comprises 200 runs and includes additionally 50 radar messages per run that are sent to the system sporadically with a minimal interarrival time of 35 ms.

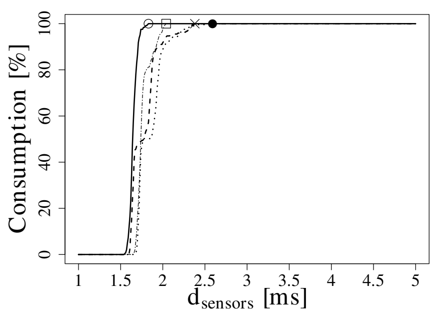
The runs of each series are divided into 25 different load situations, ranging from no additional load up to approximately 80% additional load. Since the regular system load is about 15%, the heaviest load situation takes the system almost to its limits. The load is processed by the processes within the `Load` block after the reception of special sporadic CAN messages and regulated by changing the average frequency of these messages.



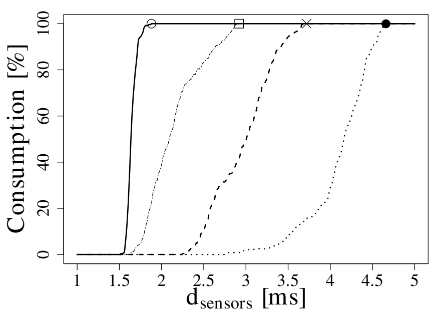
(a) Plot legend



(b) No additional load



(c) 40% additional load



(d) 80% additional load

Fig. 6. Ratio of sensor value consumptions at `speedCtrl` as a function of the sensor delay $d_{sensors}$ in three different load situations

Accuracy of Sensor Values. To enable correct operation of the PID controller, delays between taking the sensor values and running the control algorithm must be as low and as constant as possible. In particular, this implies high requirements on the processing delay $d_{sensors}$ at the controller node.

In Fig. 6, the percentage of consumed `actSpeed` and `refSpeed` signals is plotted against $d_{sensors}$ for three different amounts of load. The lines in each plot show how many sensor values have been received by the `speedCtrl` process after a given delay $d_{sensors}$. On each line, there is a point marking the maximal delay, i.e., the time after which the latest sensor value was updated in `speedCtrl`.

In case of no additional load (Fig. 6(b)), only sensor values are sent to the system and all schedulers perform almost similar. In detail, the *best* scheduler (Priorities$_{tasks}$) delivers the latest sensor signal after 1.67 ms and the *worst* scheduler (Priorities$_{process}$) requires 1.76 ms. These small differences are basically due to two reasons: First, `ConcatCoder` forwards `actSpeed`, one of the sensor signals, to `distanceCtrl` and to `speedCtrl`, thereby introducing a serialization delay. The second reason is due to different overhead of the scheduling algorithms.

However, in situations with load, the sensor delays increase and the differences between the scheduling strategies become observable. In Fig. 6(c), about

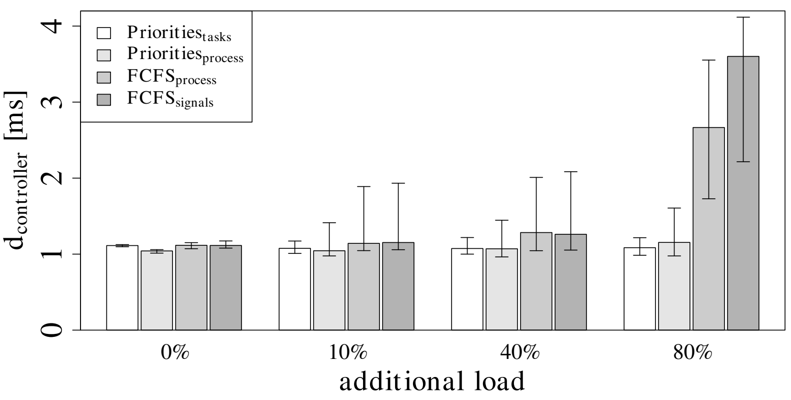


Fig. 7. Delay of calculating the control value and transfer to the SDL environment

40% additional load is added to the system, resulting in a maximal sensor delay of 1.83 ms for the best scheduler Priorities$_{tasks}$ and 2.59 ms for the worst scheduler FCFS$_{signals}$. Here, the increased delay with Priorities$_{tasks}$ is due to its lack of preemption, i.e., before the SDL environment is executed to transfer an SDL signal with sensor values into the system, the running transition must finish. A second reason is attributed to software and hardware caches performing more replacements in case of load. But all other strategies additionally suffer from an inadequate transition execution order. Though the second best strategy Priorities$_{process}$ benefits from the rejection of processes in the `Load` block (maximal delay 2.13 ms), sensor delays are increased due to the execution of transitions in `ConcatCoder` and `CANMac` that are triggered by signals belonging to the background load.

In the high load situation (Fig. 6(d)), the differences become even larger and only Priorities$_{tasks}$ is almost insusceptible against the load. As result, the maximal sensor delay with task scheduling is 41% lower than with the next best scheduling strategy Priorities$_{process}$. Another big advantage is the low sensor delay jitter with Priorities$_{tasks}$ that is only 370 μs. In contrast, the second best scheduling strategies suffers from a jitter of 1560 μs.

These results clearly show that a schedule as shown in Fig. 3 can be realized very accurately with task scheduling. All other scheduling strategies require a more pessimistic value for $d_{sensorsMax}$, thereby decreasing the quality of control.

Control Delay. This section assesses the four scheduling strategies w.r.t. control delay, because the best quality of control is achieved if the periodical computation of new control values is on time and if the new control values are transferred to the actuators rapidly.

Figure 7 depicts the measured control delays in terms of a bar diagram. The plot shows average, minimum, and maximum delays for each scheduling strategy in four different load situations. The four bars in the left part of Fig. 7 present the results in case of no load. Similar to the sensor delays, control delays differ only slightly in this case. However, when load is added to the system, task scheduling again outperforms the other scheduling strategies.

If the additional system load is about 10%, the average delay remains almost unchanged, but the maximal delay increases for all scheduling strategies. However, the increase with Priorities$_{tasks}$ is much lower than with the other scheduling strategies. In detail, the maximal control delay with Priorities$_{tasks}$ is about 240 μs less than with the second best scheduling strategy Priorities$_{process}$. This difference is basically due the shared transitions in `ConcatCoder` and `CANMac` for which Priorities$_{process}$ can not distinguish between load and control signals. If the number of signals or shared processes would be higher, reaction times with Priorities$_{process}$ would even get worse.

With increasing system load, both FCFS strategies suffer more and more from an inadequate transition execution order, whereas the priority-based strategies are less prone to the load. Thus, comparing situations with 10% and 80% additional load, there are only small differences of control delay with Priorities$_{tasks}$ as well as Priorities$_{process}$. Maximum and average control delays with FCFS$_{signals}$ and FCFS$_{process}$, however, are more than 2.5 times higher than with Priorities$_{tasks}$ in case of 80% additional load, thereby demonstrating that fair strategies are not adequate if predictability is required.

Similar to the sensor delays, task scheduling does not only achieve a lower control delay but is also attended by a lower jitter, thereby increasing predictability significantly and resulting in a better quality of cruise control.

Reaction Times to Time-Critical Events. Though the fair scheduling strategies are prone to load, reading sensors and calculating control values are periodic events, thereby allowing to determine an off-line event schedule as shown in Fig. 3 by taking the maximal delays for each scheduling strategy into account. However, with sporadic events, such a schedule is often not possible or requires very pessimistic assumptions on the events' interval. In the following scenario, we add sporadic time-critical radar messages to the system that require fast reactions and measure the reaction times between the reception of the radar messages and the sending of brake commands for each scheduling strategy.

The average and maximal reaction times are presented in Fig. 8, each with 25 different load situations. In case of no load, the average reaction times are almost equal. However, increasing load results in increased average reaction times with FCFS$_{signals}$, FCFS$_{process}$, and Priorities$_{process}$, whereas the reaction times with Priorities$_{tasks}$ stay constant. For instance, the average delay with Priorities$_{tasks}$ is only 60% of the average delay with the next best scheduler Priorities$_{process}$ at 80% additional load.

The maximal delay already differs without load (see Fig. 8(b)), because only task scheduling can entirely prefer radar and brake events, which are more time-critical than the actual and reference speed or engine control values. Thus, the maximal reaction time with Priorities$_{tasks}$ is about 38% less than with

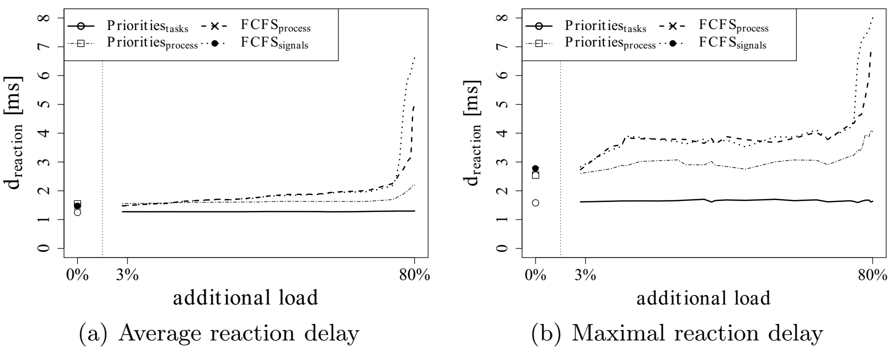


Fig. 8. Reaction delays: Time between radar messages and brake messages

Priorities$_{process}$, though there is no additional system load. The differences become even larger if the system is stressed with load, and with 80% additional load, task scheduling is actually 2.4 times faster than process priority scheduling.

Discussion. The results of all experiments demonstrate the benefits of task scheduling regarding both shorter and less variable reaction times of time-critical system tasks. They also point out that existing language elements of SDL are not sufficient to develop applications with real-time requirements if the hardware platform is predetermined and severely limited. These shortcomings can, in particular, be ascribed to the gap between the concurrent execution of all processes according to the SDL semantics and the required serialization of transition executions on embedded hardware. Though there is language support in SDL to prefer transition executions within single SDL processes (e.g., priority inputs)[4], the scheduling non-determinism of transition executions in different SDL processes is not addressed by SDL. Thus, the preference of specific system tasks distributed across several SDL processes can not be expressed in SDL.

To serialize transition executions, several scheduling strategies have been proposed. In our evaluation, we compare task scheduling with three common SDL scheduling strategies. In summary, the results demonstrate that existing approaches have drawbacks: FCFS strategies are in general inadequate, because they can not prefer time-critical system tasks at all. Strategies with static priorities are more adequate but suffer from their dependence on the static system specification. With task scheduling, these limitations are removed by dynamically adding information about the context of a transition execution, i.e., the system task it contributes to. Though these extensions improve delays significantly, the amount of additional information and overhead is only very low.

[4] We note that these language elements are limited, because they are based on static elements in the system's specification. In contrast, real-time tasks are created at run-time and assign priorities to transition executions dynamically.

5 Related Work

SDL real-time tasks enrich SDL in two respects: First, they improve the language expressiveness by enabling the specification and identification of SDL process- and node-spanning functionalities. Thus, there is a similarity to Message Sequence Charts (MSCs) [10], which is a common technique to describe the communication within and between nodes. The second impact of real-time tasks is on the scheduling of SDL systems, in which task priorities determine transition execution orders, thereby improving the predictability of reaction times. Hence, this section also outlines related work regarding SDL schedulers.

Because MSCs are a high-level way to specify distributed behavior, there are several proposals to transform MSCs to SDL [11,12]. In [11], Dulz et al. present the transformation of MSCs to synthetic SDL specifications used for early performance predictions. The intention of the MSC to SDL transformation proposed by Khendek et al. [12] is to achieve consistency between both specifications. For this purpose, the authors present a tool called MSC2SDL using an MSC and a target SDL architecture as input. Since MSCs are not suitable for describing complete systems, the influence of such approaches on the run-time behavior is limited. Yet, MSCs are a useful method to visualize and identify SDL task types.

Due to the scheduling non-determinism of the SDL semantics, there are many proposals dealing with the implementation of SDL schedulers. These proposals can be divided into two categories: The activity thread model, mapping SDL signal transfer to procedure calls [13,14], and the scheduling of the SDL system by means of priorities [6,15,16]. An overview of alternatives of SDL implementations can be found in [17,18].

The activity thread model differs from the SDL semantics, because it is synchronous and dissolves the distinction between communication and transition execution [17]. Yet, it is an efficient and often standard-compliant way of implementing SDL. However, different from SDL tasks, the activity thread model is not able to prefer specific transitions. Additionally, due to its synchronous execution model, deadlocks may occur in systems with cyclic signal flows [19]. To overcome these limitations, several measures, e.g., the reordering of signal outputs, have been proposed [13,14]. Similar to SDL tasks, the system execution with activity threads is driven by SDL signals and not by SDL processes.

Priority-based scheduling solutions operate either on process [6,15] or signal priorities [6,16]. E.g., C-micro [6], which is part of IBM's Rational SDL suite, supports static signal priorities as well as static process priorities. An extension of SDL's execution model with dynamic process priorities is introduced in [15], where priorities are derived from fixed transition priorities, forming the basis of a preemptive scheduler and schedulability analysis. Other than task priorities, prioritization based on processes or static signal priorities is not well-suited if transitions of a process are used to fulfill both time-critical and non-time-critical functionalitites. In [16], a scheduling approach with dynamic signal priorities, called Message Earliest Deadline First (MEDF), is proposed, sorting transition executions by means of message deadlines. For this purpose, several language extensions, e.g., annotations to specify timing constraints, are presented. Similar to SDL tasks, the proposed execution model can pass priorities on to signal outputs during transition executions. A drawback of MEDF compared to SDL real-time tasks is the limitation of SDL's language constructs. Additionally, EDF scheduling is in general more costly.

6 Conclusions

In this paper, we have continued our previous work on SDL real-time time tasks both conceptually, and, in particular, with a strong focus on implementation and evaluation. Conceptually, we have introduced *distributed SDL real-time tasks*, which may now span transition executions of SDL processes of several SDL systems. This enables distribution of real-time functionality across nodes while preserving tight control of global scheduling decisions and transition execution priorities. Furthermore, we have added *SDL real-time task types*, which can be used to naturally specify task multiplexing in SDL processes.

We have then presented an overview of the implementation of SDL real-time tasks in our tool chain, with emphasis on transition scheduling strategies. Based on this implementation, we have conducted extensive experiments to provide evidence of the benefit of SDL real-time tasks. In particular, we have measured the gains of SDL real-time task scheduling w.r.t. standard SDL scheduling strategies and SDL process priority scheduling. In summary, the delays of time-critical transition executions grouped into SDL real-time tasks was considerably lower when using task scheduling, in particular, in situations with increasing CPU load. In the evaluated ACC system, it has, for instance, been shown that the worst case reaction delay with task scheduling is 2.4 times less than with SDL process priority scheduling, which suffers from processes executing time-critical and none-time-critical transitions. In addition, we have observed a dramatic drop of jitter with task scheduling, which accounts for far better predictability of reaction times. Our experiments also show that these real-time performance gains have been achieved without creating additional overhead during transition selection.

As future work, we are considering applications of SDL real-time tasks in real control systems to assess its benefit to the quality of control. Additionally, we plan to extend our simulation framework by Imote2 nodes as hardware-in-the-loop in order to evaluate distributed real-time tasks in large networked systems.

Acknowledgments. This work is supported by the Carl Zeiss Foundation.

References

1. International Telecommunication Union: ITU-T Recommendation Z.100 (12/11) - Specification and Description Language - Overview of SDL 2010 (2012), `http://www.itu.int/rec/T-REC-Z.100-201112-I`
2. Krämer, M., Braun, T., Christmann, D., Gotzhein, R.: Real-Time Signaling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 186–201. Springer, Heidelberg (2011)

3. Christmann, D., Becker, P., Gotzhein, R.: Priority Scheduling in SDL. In: Ober, I., Ober, I. (eds.) SDL 2011. LNCS, vol. 7083, pp. 202–217. Springer, Heidelberg (2011)
4. Gotzhein, R.: Model-driven by SDL - Improving the Quality of Networked Systems Development (Invited Paper). In: Proceedings of the 7th International Conference on New Technologies of Distributed Systems (NOTERE 2007), pp. 31–46 (2007), http://vs.cs.uni-kl.de/publications/2007/Go07/Go07.pdf
5. Christmann, D., Gotzhein, R.: Real-time Tasks in SDL. In: Haugen, Ø., Reed, R., Gotzhein, R. (eds.) SAM 2012. LNCS, vol. 7744, pp. 53–71. Springer, Heidelberg (2013)
6. IBM Corporation: Rational SDL Suite, http://www-142.ibm.com/software/products/us/en/ratisdlsuit
7. International Telecommunication Union: ITU-T Recommendation Z.100 Annex F: Formal Semantics Definition (2000), http://www.itu.int/rec/T-REC-Z.100
8. Braun, T., Gotzhein, R., Wiebel, M.: Integration of FlexRay into the SDL-Model-Driven Development Approach. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 56–71. Springer, Heidelberg (2011)
9. FlexRay Consortium: FlexRay Communication System Protocol Specification Version 3.0.1 (2010)
10. International Telecommunication Union: ITU-T Recommendation Z.120 (02/2011) Message Sequence Charts (MSC) (2011), http://www.itu.int/rec/T-REC-Z.120-201102-I/en
11. Dulz, W., Gruhl, S., Lambert, L., Söllner, M.: Early Performance Prediction of SDL/MSC Specified Systems by Automated Synthetic Code Generation. In: SDL 1999 –The Next Millennium, pp. 457–471. Elsevier (1999), http://dx.doi.org/10.1016/B978-044450228-5/50030-8
12. Khendek, F., Zhang, X.J.: From MSC to SDL: Overview and an Application to the Autonomous Shuttle Transport System. In: Leue, S., Systä, T.J. (eds.) Scenarios. LNCS, vol. 3466, pp. 228–254. Springer, Heidelberg (2005)
13. Langendörfer, P., König, H.: Automated Protocol Implementations Based on Activity Threads. In: 7th International Conference on Network Protocols, ICNP (1999)
14. König, H., Langendörfer, P., Krumm, H.: Improving the Efficiency of Automated Protocol Implementations using a Configurable FDT Compiler. Journal of Computer Communications 23(12), 1179–1195 (2000)
15. Álvarez, J.M., Díaz, M., Llopis, L., Pimentel, E., Troya, J.M.: Integrating Schedulability Analysis and Design Techniques in SDL. Journal of Real-Time Systems 24(3), 267–302 (2003)
16. Kolloch, T.: Scheduling with Message Deadlines for Hard Real-Time SDL Systems. PhD thesis, Technische Universität München (2002), http://tumb1.biblio.tu-muenchen.de/publ/diss/ei/2002/kolloch.pdf
17. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. Prentice Hall (1993)
18. Mitschele-Thiel, A.: Engineering with SDL – Developing Performance-Critical Communication Systems. John Wiley & Sons (2000)
19. Sanders, R.: Implementing from SDL. In: Telektronikk 4.2000, Languages for Telecommunication Applications. Telenor (2000)