Priority Scheduling in SDL*

Dennis Christmann, Philipp Becker, and Reinhard Gotzhein

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

Abstract. In real-time systems, the capability to achieve short or even predictable reaction times is essential. In this paper, we take a pragmatic approach by proposing priority-based scheduling in SDL combined with a mechanism to suspend and resume SDL agents. More specifically, we define adequate syntactical extensions of SDL and show that they are compliant with the formal SDL semantics. We have implemented all proposed extensions in our SDL tool chain, consisting of SDL compiler, SDL runtime environment, and environment interfacing routines, thereby being compatible with model-driven development processes with SDL. In a series of runtime experiments on sensor nodes, we show that compared to customary SDL scheduling policies, priority scheduling with suspension of SDL agents indeed achieves significantly shortened reaction times.

1 Introduction

In *real-time systems*, correctness does not only depend on functional correctness, but also on the points in time at which results are produced (see [1]). Timeliness is also crucial in *communication networks* applying TDMA for medium access, where nodes transmit frames in their assigned time slots, thereby avoiding frame collisions, or for efficient duty cycling. To support timeliness, the capability to achieve short or even predictable reaction times is essential.

SDL [2] is a formal description technique for distributed systems that is also being advocated for the design of real-time systems. To specify real-time aspects, SDL offers the notion of global system time, referred to by the function `now`, and the concept of SDL timers. However, timer expiry does not yield an instantly-processed interrupt; instead, a timer signal is produced and appended to the input queue of the local agent. In real implementations of SDL models, this timer semantics may cause substantial delays, thereby inhibiting the timely reaction of the SDL agent to this event. In particular, delays are caused by three sources: First, the SDL runtime environment implementing the SDL Virtual Machine (SVM) has to detect timer expiry, and to place a signal into the corresponding input queue. Secondly, the receiving agent has to be scheduled and dispatched. Especially in high load situations with many agents in the SDL system, the agent's scheduling delay may be high. Reasons for this potentially high delay are

* This work is supported by the Carl Zeiss Foundation.

I. Ober and I. Ober (Eds.): SDL 2011, LNCS 7083, pp. 202–217, 2011.

that neither does the SDL semantics stipulate a particular scheduling strategy nor does SDL provide any means to specify preference rules for agents. Thirdly, there may be further signals in the input queue to be consumed before the timer signal, which implies that the receiving agent may have to be scheduled more than once. Similar concerns apply to scenarios where no SDL timers but regular SDL signals are sent to trigger other SDL agents in a timely fashion.

In this paper, we take a pragmatic approach to improve the real-time capabilities of SDL. More specifically, we propose SDL extensions that can be used by the system designer to influence the agents' execution order. The first extension adds the possibility to assign static priorities to SDL agents, thereby giving privilege to time-critical tasks and shortening their reaction times even in situations with potentially high system load. The second extension is a mechanism to suspend and resume SDL agents, thereby reducing system load and thus reaction times in critical time intervals. This way, the problem that a transition of a low priority agent may be running when a transition of a high priority agent becomes fireable can be solved.

The work reported in this paper makes the following contributions. First, we propose SDL extensions to select among several scheduling strategies and to assign static priorities to SDL agents combined with a mechanism to suspend and resume agents. For compatibility with existing SDL tools, these extensions are defined by annotations (see Sect. 3). Also, we address their compliance with the formal SDL semantics. Secondly, we have implemented all proposed extensions in our SDL tool chain, consisting of the SDL compiler ConTraST [3], the SDL Runtime Environment (SdlRE), and the SDL Environment Framework (SEnF) to interface SDL systems with hardware platforms. Section 4 addresses several implementation aspects. Thirdly, we have performed runtime experiments in order to compare the performance of ordinary SDL scheduling policies to priority scheduling with and without the suspension of low priority agents (Sect. 5). The experiments provide sufficient evidence that the proposed SDL extensions are necessary and effective. The paper is completed by a survey of related work (Sect. 2), and conclusions with an outline of future work (Sect. 6).

2 Related Work

In this section, we survey related work. First, we look at approaches exploiting implementation choices to improve runtime efficiency of SDL. Then, we outline design and analysis techniques to increase predictability of SDL systems.

The improvement of runtime efficiency of SDL implementations is extensively treated in two standard works. In [4], Bræk and Haugen compare the conceptual SDL world to real hardware systems and discuss fundamental differences, e.g., limited hardware resources and processing delays. They introduce step-by-step guidelines to distribute SDL systems to several physical entities, and to dimension hardware resources. They also discuss the software design of SDL, i.e., alternatives how to implement SDL constructs, e.g., regarding communication, concurrency, data types, and the sequential behavior of SDL processes.

The second standard work by Mitschele-Thiel [5] focusses on performance engineering. His comprehensive treatment covers differences between SDL and real-world implementations, too, and introduces alternatives in the integration of SDL systems with an operating system (*tight*, *light*, *bare*). He also discusses a variety of implementation alternatives that influence runtime efficiency.

In addition to these standard works, the efficient implementation of SDL is addressed, e.g., in [6] and [7]. Topics in [6] are the realization of SDL signals by method calls and the mapping of SDL entities to physical objects. Since physical distribution cannot be specified in SDL, supporting tools have been developed, e.g., IBM's deployment editor [8], which is based on UML component diagrams.

To improve predictability of SDL systems, adaptations of the SDL runtime model and pure analytical approaches that are based on assumptions on runtime environment and hardware platform have been proposed. In [9], Álvarez et al. present a more predictable but non-standard-compliant execution model for SDL to reduce non-determinism of the formal SDL model. Their solution involves the preemptive and priority-based scheduling of SDL agents. The priority of an agent is calculated dynamically and depends on signals in its queue and transitions in the agent's current state. In order to perform schedulability analysis, transitions are additionally labeled by their *worst case execution time* (WCET).

The assignment of process and signal priorities is also suggested in [4]. Some tools already support priorities, e.g., Cmicro in the IBM Rational SDL Suite [8]. Even though their approach provides an easy way to privilege agents and even transitions, it is not fully compliant to the SDL semantics. In addition, Cmicro suffers from several language restrictions.

There are also tools supporting tight integrations of an SDL system into an underlying real-time operating system (RTOS) (cf. Cadvanced integration strategies [8] or Pragmadev's Real Time Developer Studio [10]). Since such approaches introduce additional overhead compared to bare integrations, require an additional implementation step, and create a strong bonding between SDL system and RTOS, we do not go into further detail.

Bozga et al. propose a set of annotations, grouped into *assumptions* and *assertions* [11]. These annotations include a priori knowledge about the system's environment, such as execution delays and periodicity of external inputs. Additionally, they introduce cyclic and interruptive SDL timers as well as operators to access timer values. They also combine the concept of transition urgencies with SDL to influence the progress of time in simulations. In particular, they suggest to label transitions as *eager*, *lazy*, or *delayable*. Although this approach increases the expressiveness of SDL, it is more on the theoretical side, since progress of time is not controllable in real executions.

In addition, further analytical techniques have been proposed. One example is QSDL that is based on the foundations of *queuing theory* [12]. Further approaches include *timed automata* [13] and *tasked networks* [14]. Thereby, well-known analytical methods, e.g., known from model checking, are brought into the context of SDL. However, most analytical methods consider execution delays

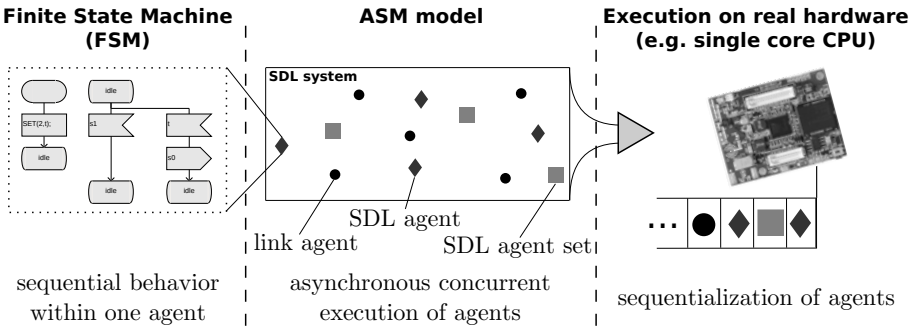


Fig. 1. Gap between SDL's formal semantics and the behavior on real hardware

within single transitions only and ignore overhead of the runtime system (e.g., the selection of fireable transitions).

To our knowledge, our approach is the first to provide priority scheduling combined with temporary suspension of low-priority agents. Different from the analytical approaches, it covers the explicit configuration of SDL runtime models and is fully implemented in an operational SDL tool chain. Unlike existing implementations and the standard works on implementing SDL, our focus is not on more efficient implementations of SDL, but on customized, transparent scheduling of SDL agents compliant with the SDL standard [2]. Moreover, our realization with low-power sensor nodes does not rely on an underlying (real-time) operating system, but states a bare integration on hardware, thereby being more efficient and controllable.

3 Scheduling Strategies in SDL

This section introduces annotation-based extensions to SDL to increase the language's expressiveness [15]. They are part of an ongoing research process with the objective to bring SDL further towards real-time systems. The presented annotations enrich system specifications with information configuring the system's scheduling strategy. By introducing priority scheduling, we establish a basis to privilege time-critical tasks. With the suspension of low priority agents, we provide additional measures to shorten reaction times further.

3.1 Problem Statement

The scheduling of SDL agents is a crucial step when implementing the concurrent runtime model of SDL, since it is critical with respect to reaction times and compliance with deadlines. Fig. 1 illustrates the gap between SDL's concurrent runtime model and its sequentialized execution on real hardware. The figure is subdivided into three parts: The left part shows the specification of a finite state machine (FSM), e.g., in the context of an SDL process. Transitions

within one FSM are executed sequentially and must not be interrupted by another transition of the same FSM (*run-to-completion*). The middle part of Fig. 1 shows the runtime model of SDL that is specified by Abstract State Machines (ASMs) in the dynamic semantics [2]. The formal semantics defines three types of ASM agents (`SDL AgentSets`, `SDL Agents`, and `Links`) that are executed concurrently and autonomously by the SDL Virtual Machine (SVM). Nevertheless, the concurrency has to be eliminated by a certain scheduling strategy when the SDL system is executed on real hardware, e.g., on a single-core platform (cf. right part in Fig. 1), which is common in many application areas such as wireless sensor networks. In general, asynchronous concurrency and the vague notion of execution time allow arbitrary scheduling strategies – preemptive as well as non-preemptive – for the sequentialization of SDL agents. However, SDL itself provides no means to specify a particular scheduling strategy.

In our approach, we introduce extensions to SDL, enabling the selection and configuration of the scheduling strategy that is suitable for the concrete scenario. Thereby, scheduling of agents is not only more transparent, but also manipulable by the system developer. In addition, the approach provides the option to privilege certain agents and to temporarily suspend low priority agents in order to speed up time-critical reactions. Further objectives are compatibility with existing tools and the exploit of SDL's semantical freedom. Particularly, we leave the behavior of input ports, which is often manipulated when introducing priorities in SDL (e.g., in [9]), unchanged.

3.2 Supported Scheduling Strategies

When implementing SDL, there is in general no restriction on the scheduling strategy to be used. However, if the whole language is to be supported, offline time-triggered scheduling strategies are to be rejected and event-triggered strategies are the only remaining alternative to deal with SDL's actor model.

We have devised a scheduling framework within the *SDL Runtime Environment* (SdlRE), our implementation of the SVM, which currently supports three dynamic, non-idling, and non-preemptive scheduling strategies [16]. This means that scheduling decisions are made at runtime, that no idling periods are allowed if there are agents with pending tasks, and that an executed agent must return control to the runtime environment, since there is no enforced interruption of an agent's execution (*cooperative scheduling*). Although non-preemptive strategies can handle less scheduling problems, we do not support preemptive algorithms, since they require more overhead at runtime (coordination, restrictive synchronization of shared variables, ...), and are harder to implement and analyze than non-preemptive ones [17].

Non-preemptive Round Robin (NP-RR). With the *non-preemptive round robin* scheduling strategy (NP-RR), all agents of the system, i.e. `SDL Agents`, `SDL AgentSets`, and `Links`, are executed in a cyclic manner [18]. The order within the cycle is given by the agents' initialization order, which depends on the SDL tool chain and therefore cannot be prescribed during system design. NP-RR

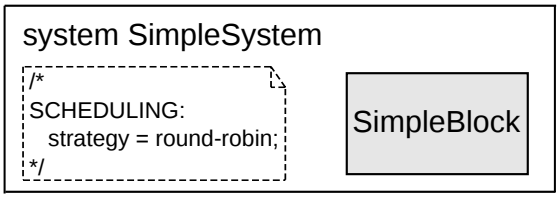


Fig. 2. Global scheduling policy in head symbol of the SDL system

reflects the SDL runtime model one-to-one, in the sense that every agent decides by itself if there is something to do, e.g., if there are SDL signals to process. Note that this is no necessary condition for compliance with SDL's semantics, since an agent without a fireable transition does not show any reaction from an outer point of view. Because many agents are executed without any pending signal, NP-RR usually suffers from low efficiency.

To select NP-RR as scheduling strategy in an SDL system, we specify a scheduling policy in the system's head symbol (see Fig. 2). The policy is specified as annotation, i.e., as formal SDL comment, thereby being compatible with existing SDL tools. Inside the annotation, the keyword `SCHEDULING` marks the beginning of a scheduling policy and is used in our tool chain as indicator to evaluate the annotation (see Sect. 4). Subsequent to this keyword, a set of parameters to choose and configure a desired scheduling strategy follows. In Fig. 2, there is only one parameter assigned, setting the scheduling strategy to NP-RR (`strategy = round-robin`).

Agent-based First Come First Served (FCFS). Compliance of the SVM's implementation with SDL's semantics does not require the one-to-one mapping of all types of ASM agents. Particularly, it is usual to omit the scheduling and dispatching of `SDL AgentSets` and `Links` and to flatten the system's hierarchy, i.e., to transport signals directly between `SDL Agents` holding the actual FSM that is specified by the developer. In these realizations, the functionality provided by `SDL AgentSets` and `Links` is moved into service primitives of the SDL runtime system. Regarding efficiency, it is even recommended to implement a more demand-driven scheduling strategy, i.e., to execute `SDL Agents` only when they hold fireable transitions.

The agent-based *first come first served* (FCFS) strategy realizes such a demand-driven scheduling in SdlRE [18]. It maintains a queue containing all agents with pending signals.[1] In steady state, there are only `SDL Agents` in the queue. `SDL AgentSets` and `Links` are only executed explicitly if new system structures are created (e.g., during system startup or dynamic process instantiation). If an SDL signal is put into the input port of an agent and the agent is not yet in the

[1] SdlRE also supports other types of transition triggers like `continuous signals`. However, it is often not easy to decide whether (and when) corresponding conditions yield true (e.g., if the condition contains `now`). Thus, such language features can hardly be treated in a pure demand-driven way.

queue, it is inserted at the end of the queue. On the other hand, if the agent is already enqueued, the signal is inserted in the input port of the agent without changing the agent's position in the queue. Thereby, the resulting execution order is fair on agent level. Compared to NP-RR, execution order of agents with FCFS is significantly more transparent. However, if two agents become fireable simultaneously, e.g., if two timers expire at the same point in time, the behavior of the SDL model also depends on the SDL tool chain. In particular, the system designer can not enforce a desired order in this case. Similar to NP-RR scheduling, FCFS is selected in the system's head symbol (`strategy = fcfs`). This is also the default scheduling strategy in our SVM implementation.

Priority Scheduling. In many systems, privilege of critical tasks has to be provided instead of fairness. Particularly, in event-triggered real-time systems, priority-based scheduling is an indispensable precondition for the timely execution of critical tasks and the shortening of their reaction times.

With *priority scheduling*, SdlRE provides a scheduling strategy with fixed priorities that are assigned on agent level. In particular, only `SDL Agents` can be configured with explicit priorities. `SDL AgentSets` and `Links` always run with system-wide highest priority, since they are executed only when then system structure changes (cf. FCFS strategy). For the same reason, the initiation of `SDL Agents` is privileged with highest priority as well. Thus, the final priority of an `SDL Agent` is assigned after the execution of its start transition.

Even with priorities, there are still situations in which system behavior is nondeterministic. In particular, FCFS is used within each priority class, i.e. if two timers in two agents of same priority expire simultaneously, the execution order is not prescribed by the specification. However, such situations can be resolved by the designer by assigning different priorities.

In contrast to signal priorities in Cmicro [8], priority scheduling leaves the consumption order of signals in an input port unchanged. Particularly, the semantics of `priority inputs` is untouched [2]. Thereby, SDL's implementation freedom is exploited while remaining compliant to its formal semantics.[2]

Similar to NP-RR and FCFS, the system developer selects priority scheduling by setting `strategy = static-priorities` in the SDL system's head symbol (see Fig. 3). In addition, priority scheduling provides parameters that the developer has to configure during system design, namely the number of priority levels (`levels = 5`) and the default priority of `SDL Agents` (`priority = 1`). By setting `levels` to a concrete natural number, the range of valid priorities is determined, which is from 0 (highest priority) to `levels-1` (lowest priority). Default priorities are used for `SDL Agents` without explicit priority assignment.

The assignment of priorities to `SDL Agents` is also defined by the developer in terms of annotations. For this purpose, several possibilities exist on various levels of the system hierarchy to allow priority assignments in a flexible way. In general, a priority is assigned in an SDL comment symbol that is connected

[2] We assume that the system is not overloaded permanently. Otherwise, there is no guarantee that a low-priority agent is eventually executed.

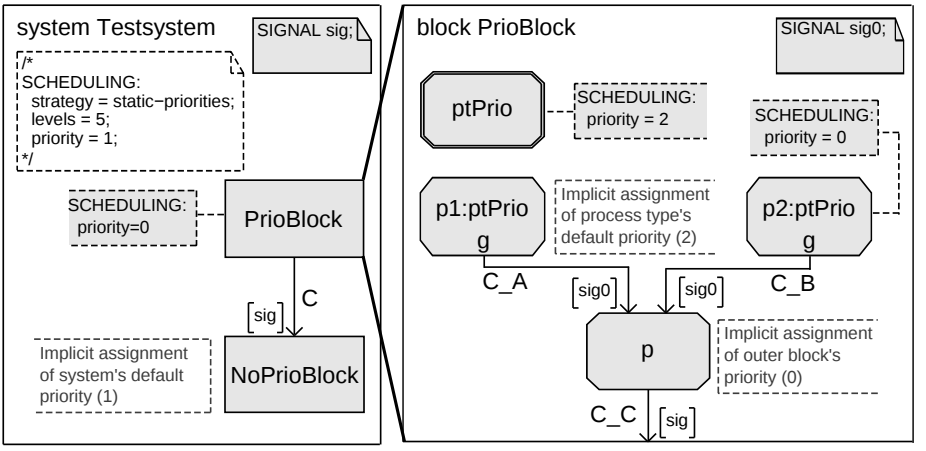


Fig. 3. Priority scheduling: Strategy is selected and configured in the head symbol of the SDL system. Priorities are directly assigned to SDL components.

to a corresponding SDL component (see Fig. 3). Again, annotations within the comment symbol start with the keyword `SCHEDULING`, followed by `priority = <priority>` (e.g., `PrioBlock` in Fig. 3). They can be associated with services,[3] processes, and blocks, in each case with instantiations as well as type definitions. If a priority is assigned to both type definition and instantiation, the instantiation's priority overrides the type definition's priority, resulting in an unambiguous priority assignment (cf. `ptPrio` and `p2:ptPrio` in Fig. 3). Otherwise, the instantiation gets the priority of the type definition (cf. `p1:ptPrio` in Fig. 3).

Although priorities are assigned at design time, the priorities of the resulting `SDL Agents` must be determined at runtime, since they are in general optional and depend on the agents' positions in the system hierarchy. In particular, the priority of an `SDL Agent` is determined according to the following constraints of the SDL specification, where the evaluation is applied in descending order:

1. If a priority is assigned to the SDL process or it contains services with assigned priorities, then the resulting `SDL Agent` gets the highest priority of all these assigned priorities.
2. If a process inherits from another process with assigned priority, the resulting `SDL Agent` gets the priority of the supertype. If the inheritance affects multiple levels, the priority is evaluated from bottom to top.
3. If a priority is assigned to the block specification containing the process, the resulting `SDL Agent` gets the block's priority (cf. process `p` in Fig. 3). If there is a hierarchical nesting of blocks, the process priority is evaluated from bottom to top.
4. If there is no priority assigned at all, the default priority is used as defined in the system's head symbol.

[3] Since SDL 2000, services are replaced by state aggregations [2]. Due to missing tool support, we still refer to services.

3.3 Suspension of Agents

All scheduling strategies supported by SdlRE are non-preemptive, leading to an efficient implementation with less runtime overhead. However, the non-preemptive scheduling delays the execution of high-priority agents until the currently running transition is finished.

To solve this problem and to speed up reaction times of high-priority agents further, priority scheduling in SdlRE provides the option to temporarily suspend agents according to their priority level. By this approach, system load can be reduced prior to an expected time-critical task, e.g., before time resynchronization starts. For this purpose, two SDL procedures, `Suspend` and `Resume`, have been specified, building a wrapper to corresponding scheduler functions in the SdlRE. The `Suspend` procedure provides a priority level as parameter, which states the borderline between agents that are dispatched further and agents that are (temporarily) prevented from execution. By calling the `Resume` procedure, all suspended agents are released.

In general, the decision to suspend particular agents from execution should be taken from some central components with scenario-specific knowledge. By enabling the suspension of agents by means of SDL procedures, components taking this decision can be specified in SDL during system design, e.g., in a centralized SDL process. Thereby, application knowledge and the system's current state can be taken into account to decide which agents are to be suspended.

3.4 Dealing with the SDL Environment

According to SDL's semantics, it is assumed that the SDL environment has one or more agent instances [2]. In our implementation of the environment, the *SDL Environment Framework* (SEnF), there is exactly one agent. Different from ordinary agents within the SDL system, the environment agent requires a special treatment, since it can be triggered by external events, e.g., a hardware interrupt. Therefore, when searching for an appropriate scheduling strategy for SDL systems, the question arises how to schedule the environment.

A naive solution might be the execution of the environment agent in *idle* times of the system, i.e., when no other agent is executable. This solution corresponds to a priority-based solution, at which the environment agent has lowest priority. However, in cases of temporary intra-system load, system input/output would be delayed significantly, which is not acceptable for many scenarios. Another solution would be the immediate execution of the environment agent each time a signal is pending to/from the environment, which corresponds to the assignment of the highest priority in a priority-based solution. But this solution may cause problems in case of polling hardware drivers and can also lead to overload within the system, since adding additional load from the environment to the system would be privileged. In sum, there is no generic solution to handle the environment, since the best strategy depends on the specific scenario as well as the used hardware platform. We propose again an annotation-based solution to let the system developer decide how to deal with the environment during load

situations. In particular, we extend the presented scheduling annotations with three optional parameters that are specified in the system's head symbol:

- `env-signal_in-threshold`: Number of signals in the environment's input port that enforces the execution of the environment agent.
- `env-signal_out-threshold`: Threshold on the number of signals generated by the environment that are to be sent to the SDL system. Exceeding this value enforces the execution of the environment agent.
- `env-agent-threshold`: Number of non-environment agent executions that enforces the execution of the environment agent. This parameter defines a maximal interval (in terms of number of agent executions) between subsequent executions of the environment agent.

In general, smaller numbers lead to faster reactions. However, too small numbers may increase system load in situations in which load is already high. Therefore, adequate planning and load estimations are required to determine a suitable configuration. Compared to a scheduling of the SDL environment with one static priority, the solution based on thresholds is stronger and more flexible.

4 Implementation Aspects

In this section, we outline the implementation of the proposed extensions of SDL. Components and dependencies of our tool chain are illustrated in Fig. 4. The figure shows an excerpt of SDL-MDD (SDL-Model Driven Development), a holistic and domain-specific development process with SDL [19]. The excerpt starts with the *Platform-Independent Model (PIM)*, a functionally complete SDL specification. Next, the PIM is transformed to the *Platform-Specific Model (PSM)*, by adding hardware-specific aspects, e.g., to interface with specific communication technologies, and by adding scheduling aspects as introduced in Sect. 3. This transformation is partially guided by a set of heuristics, and supported by reuse approaches, e.g., SDL design patterns and SDL micro protocols. To generate code, the PSM is first transformed to SDL-PR format, using the IBM Rational SDL Suite, which accepts and preserves the annotations specifying scheduling aspects as (formal) comments. With the code transpiler ConTraST [3], the SDL-PR format is then automatically transformed to *Runtime-Independent Code (RIC)*, the C++ representation of the SDL system.

To incorporate the proposed SDL annotations into the generated code, we have modified and extended ConTraST. Before generating the C++ representation, ConTraST checks the correctness of the annotations' syntax and applies additional semantic checks, e.g., if an assigned priority is in the range of allowed priority levels. Afterwards, annotations regarding scheduling are grouped into runtime-independent and runtime-dependent annotations. Runtime-independent annotations describe static configuration parameters (e.g., the selected scheduling strategy) and are transformed into C macros. Thereby, they are already considered at compile time, decreasing the overhead at runtime. Runtime-dependent annotations are more dynamic in the sense that the corresponding configuration

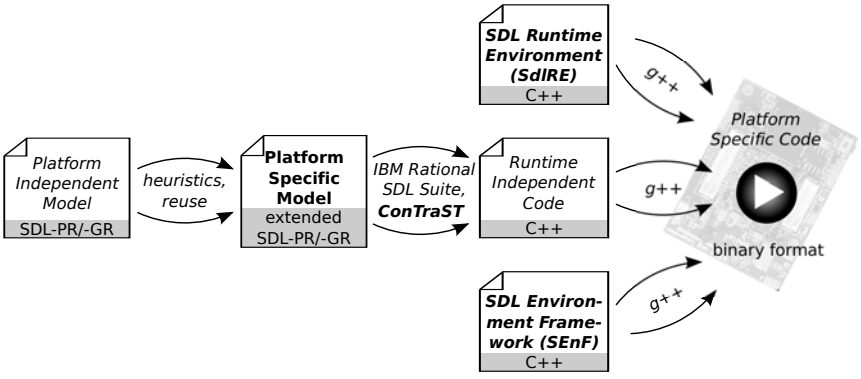


Fig. 4. Model-driven development with SDL and its annotation-based extensions

parameter must be determined at runtime. An example of a runtime-dependent annotation is the priority of an agent.

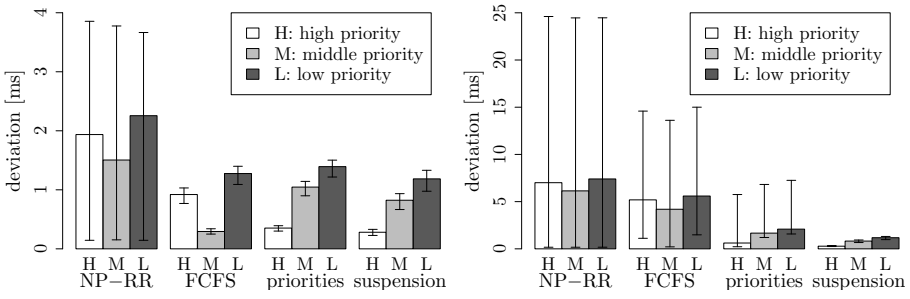
The RIC is complemented by the SDL Runtime Environment (SdlRE) and the SDL Environment Framework (SEnF). SdlRE, our implementation of the SVM, coordinates the execution of agents and the delivery of SDL signals. SEnF provides environment interfacing routines specific to the selected hardware platform, for instance for communication devices. To realize the scheduling annotations, we have modified and extended SdlRE and SEnF. In particular, an extensible scheduling framework with a well-defined interface was incorporated into SdlRE, in which, up to now, three scheduling strategies have been integrated (see Sect. 3).

To obtain *Platform-Specific Code (PSC)*, SdlRE, SEnF, and the RIC are compiled by a platform-specific compiler, and linked. The generated file is then deployed on the hardware platform, and executed.

5 Experimental Evaluation

In this section, we present quantitative evaluations of priority scheduling with and without suspension of agents, and demonstrate the benefits regarding response times and predictability of critical tasks in comparison to NP-RR and FCFS. In a first series of experiments, we measure the accuracy of SDL timers, i.e., the amount of time the consumption of SDL timer signals is delayed. In a second series of experiments, we investigate the precedence of signal chains. In both cases, we use SDL benchmark specifications for better comparison, and run the experiments on sensor nodes that are fully controlled by SEnF and SdlRE. The results show that our implementation is fully operational, and that the proposed extensions are necessary and effective.

All evaluations are done by benchmark experiments on a customary Imote2 sensor node [20] (on the right-hand side of Fig. 1). The Imote2 is equipped with 32 kB SRAM, 32 MB flash memory, additional 32 MB SDRAM, and a single-core



(a) System without load (b) System with load

Fig. 5. Deviations of timer signals' actual consumption times from nominal values. Notice that (a) and (b) use different time scales.

XScale CPU, providing clock rates up to 416 MHz. The measured data values are sent to a PC via serial interface. To obtain reproducible results, further external interfaces are disabled. The evaluated SDL systems are deployed on the sensor node without further underlying operating system (bare integration). Thereby, all influences on the measured values, like interrupt handling and execution order of agents, are fully controlled by SEnF and SdlRE.

5.1 Accuracy of SDL Timers

This benchmark examines the deviation of SDL timers from their nominal values. The evaluated SDL system includes three SDL process instances of the same type, each instance setting a periodical timer to absolute and identical points in time. To each process instance, a different priority is assigned, defining an order of precedence between the agents at runtime. In addition, further processes with lower priority are added to the system to quantify the impact of load.

The results of this scenario are shown in Fig. 5 for all supported scheduling strategies. For priority scheduling, the plots distinguish between scheduling without suspension (named `priorities`) and with suspension (named `suspension`). In Fig. 5a, timer deviations from their nominal values are shown without further system load, where in Fig. 5b, random load was added to the system. In case of priority scheduling with suspension, load-generating agents are suspended from execution. Both plots illustrate, for every SDL timer, the average, minimum, and maximum deviation of timer signal consumption times from their nominal values. Each bar is based on 6000 measured samples.

In both plots, deviations are always greater than $0\,ms$ due to the overhead of selecting agent and transition. Both plots also show that average deviations and jitter are significantly higher when NP-RR is used. This is due to the scheduling of all agents with NP-RR, independent of signals in their input queue.

In case of no load (Fig. 5a), deviations with FCFS are in general comparable to deviations with priority scheduling. With FCFS, the precedence order between agents does not fit to the developer's requirements, since priorities can not be

considered. However, there is also a clearly visible order when using FCFS (`H` – middle deviations, `M` – smallest deviations, `L` – largest deviations). This can be traced back to the process specification order in the SDL-PR file, a tool-specific property that is hardly ascertainable and influenceable during the creation of the SDL system. Since the system is free of background load, there is no necessity to suspend low-priority agents, thus, the results of priority scheduling without and with suspension are similar.

When load is added to the system (Fig. 5b), priority scheduling shows its full potential. Different from NP-RR and FCFS, average deviations and jitter increase significantly slower, since load-generating agents have lower priority, resulting in less influence in the consumption time of the considered agents' timer signals. In particular, average deviations increase by a factor of about 0.7 with priority scheduling (without suspension), where deviations with NP-RR and FCFS increase by factors of 3 up to 13. Comparing the average deviations with priority scheduling without suspension against FCFS, the high priority agent is executed 4.6 ms earlier (0.6 ms vs. 5.2 ms). This impressive number illustrates the need for priority-based scheduling in SDL. Although the degradation of timer accuracy is significantly smaller when using priority scheduling without suspension, the jitter is higher in comparison to the experiments without load, since load-generating transitions cannot be interrupted. This problem is solved by priority scheduling with suspension of low-priority agents. In particular, scheduling with suspension does not show any difference when load is added to the system, resulting in substantially more accurate and load-independent consumption times of timer signals, and thus in more predictable system behavior.

5.2 Prioritized Signal Exchange

In this scenario, the main focus is on the functional evaluation of NP-RR, FCFS, and priority scheduling by evaluating timing behavior of signal exchange within the SDL system. Therefore, we introduce the term *signal chain* to be a sequence $C = \langle s_1, s_2, \ldots, s_n \rangle$ of signals, started by signal s_1, such that consumption of signal s_i triggers the output of s_{i+1}, for $1 \leq i \leq n-1$. The evaluated SDL system processes three identical signal chains $C_j = \langle s_{j,1}, \ldots, s_{j,n} \rangle$, $1 \leq j \leq 3$, where $s_{1,i}$, $s_{2,i}$, and $s_{3,i}$ (with $1 \leq i \leq n$) are processed by identical components (SDL processes and services) with different assigned priorities. Each signal chain triggers a total of 14 transitions, allocated into 5 SDL services and 9 SDL processes. By using inheritance and priority annotations on several hierarchy levels (blocks $\leftrightarrow$ processes $\leftrightarrow$ services), the correct implementation of priority scheduling as described in Sect. 3.2 is validated.

Figure 6 shows completion times of the three signal chains for FCFS, NP-RR, and priority scheduling without suspension. For each scheduling strategy, every signal chain is completed 4000 times. Because no random load was added to the system, completion times only have a small jitter. As expected, completion times with NP-RR are in general significantly higher due to the (non-required) execution of idling agents. In case of NP-RR and FCFS scheduling, there are only marginal differences in completion times of the three chains, where the

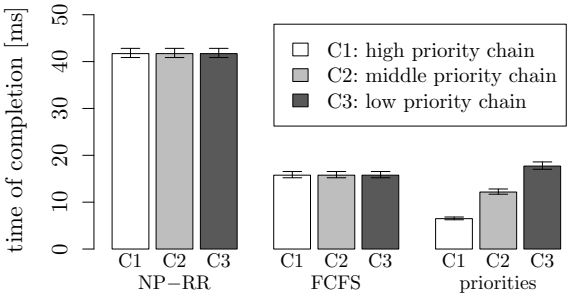
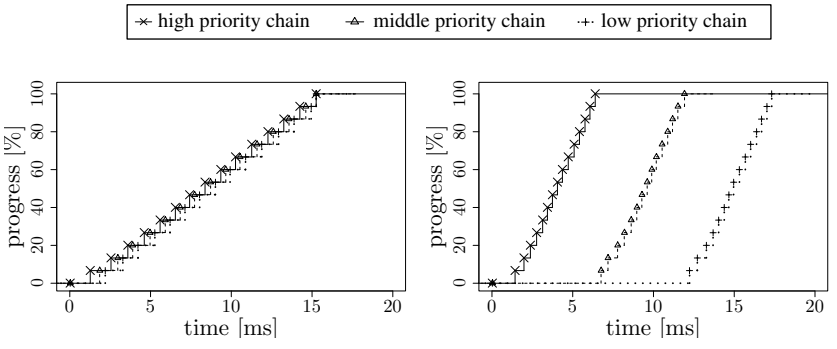


Fig. 6. Time of completion (avg/min/max) of all signal chains



(a) FCFS scheduling (b) Priority scheduling

Fig. 7. Comparison of example signal chain progresses

precedence order between tasks is clearly observable in case of priority scheduling. Particularly, the completion time of the high-priority chain is 6.4 times (2.5 times) shorter than the completion time with NP-RR (FCFS) scheduling. Thus, reaction times on critical events are significantly shorter.

To demonstrate the differences in scheduling with FCFS and priority-based scheduling, Fig. 7 depicts showcase signal progresses per signal chain. With FCFS, signal progress is nearly in lockstep, since the scheduling of agents follows their activation order. In contrast, priority scheduling executes agents with fireable transitions according to their priority. Thereby, the high priority chain is finished even before signal forwarding of other chains starts.

6 Conclusions

When executing the concurrent runtime model of SDL on a concrete hardware platform, scheduling of the system's agents becomes necessary. Often, this step is applied in an unpredictable and tool-specific way. In this paper, we have introduced annotation-based extensions to SDL, which enable the developer to better

control the execution order of agents by choosing and configuring appropriate scheduling strategies. In particular, we have introduced priority-based scheduling of SDL agents combined with a mechanism for their temporary suspension. The presented approach exploits the implementation freedom of SDL while still being compliant with its semantics. Furthermore, the approach is compatible with existing tools and incorporated into SDL-MDD, our model-driven development approach with SDL [19].

By the realization of a priority-based scheduling strategy, measures were introduced to privilege time-critical tasks and to suspend low-priority agents. Thereby, average as well as worst case reaction times of critical tasks can be reduced enormously, bringing SDL an important step closer to real-time system development. In experimental evaluations on customary sensor nodes, the benefits of priority scheduling with and without suspension were demonstrated. For instance, on a system with random load, consumption delays of expired critical SDL timers were more than 10 times smaller than consumption delays with FCFS scheduling.

During our experiments, it turned out that the priority of the SDL environment agent has a large impact on the predictability of reaction times. Here, pure priority scheduling with a single, static priority of the environment agent is not sufficient. With environment signal thresholds, we have presented an approach that distinguishes between input and output signals to and from the environment. This approach can be seen as a generalization of static priority scheduling: usually, the environment agent has the lowest priority, but obtains the highest priority if the threshold is exceeded - a step towards dynamic priority scheduling.

The SDL extensions proposed in this paper are an important, but not yet sufficient step towards hard real-time system development with SDL. First, tasks often consist of several related transitions, which may be associated with different SDL agents. If these SDL agents have different priorities, task execution can not be performed in a timely manner. Secondly, to design hard real-time systems, Worst-Case Execution Times (WCETs) must be known. While it is already difficult to determine WCETs for single transition bodies, it is even more difficult if the assumption that agent scheduling takes no time is dropped. Thirdly, agent priorities have to be assigned dynamically, if they are time-dependent, or if they depend on the current system state. For instance, an SDL process performing network synchronization should be privileged during resynchronization phases only. In summary, these points call for dynamic priorities, a notion of task in SDL, and WCETs of tasks. We will address these topics in our future work.

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)
3. 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)

4. Bræk, R., Haugen, Ø.: Engineering Real Time Systems. Prentice Hall, Englewood Cliffs (1993)
5. Mitschele-Thiel, A.: Engineering with SDL – Developing Performance-Critical Communication Systems. John Wiley & Sons, Chichester (2000)
6. Sanders, R.: Implementing from SDL. In: Telektronikk 4.2000, Languages for Telecommunication Applications, Telenor (2000)
7. Leblanc, P., Ek, A., Hjelm, T.: Telelogic SDL and MSC tool families. In: Telektronikk 4.2000, Languages for Telecommunication Applications, Telenor (2000)
8. IBM: Rational SDL Suite (2011), http://www-01.ibm.com/software/awdtools/sdlsuite/
9. 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)
10. Pragmadev: Real time developer studio (2011), http://www.pragmadev.com/
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. 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)
13. 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)
14. 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)
15. Christmann, D.: Spezifikation und automatisierte Implementierung zeitkritischer Systeme mit TC-SDL. Master's thesis, TU Kaiserslautern (2010)
16. Stankovic, J.A., Ramamritham, K.: Hard Real-Time Systems, Tutorial. IEEE Computer Society Press, Los Alamitos (1988)
17. Jeffay, K., Stanat, D.F., Martel, C.U.: On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In: IEEE Real-Time Systems Symposium, pp. 129–139 (1991)
18. Fliege, I.: Component-based Development of Communication Systems. PhD thesis, University of Kaiserslautern (2009)
19. Gotzhein, R.: Model-driven with SDL – Improving the Quality of Networked Systems Development. In: Proc. of the 7th Int. Conf. on New Technologies of Distributed Systems (NOTERE 2007), Marrakesh, Morocco, pp. 31–46 (2007)
20. Memsic: Imote 2 datasheet (2011), http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134