Deriving Hard Real-Time Embedded Systems Implementations Directly from SDL Specifications

J.M. Alvarez, M. Diaz, L. Llopis, E. Pimentel, J.M.Troya
Dpto. Lenguajes y Ciencias de la Computacion
Complejo Politecnico. 29071
University of Malaga

{alvarezp,mdr,luisll,ernesto,troya}@lcc.uma.es

ABSTRACT

Object-Oriented methodologies together with Formal Description Techniques (FDT) are a promising way to deal with the increasing complexity of hard real-time embedded systems. However, FDTs do not take into account non-functional aspects as real-time constraints. Based on a new real-time execution model for FDT SDL proposed in previous works, a way to derive implementations of hard real-time embedded systems directly from SDL specifications is presented. In order to get it we propose a middleware that supports this model to organize the execution of the tasks generated from SDL system specification. Additionally, a worst case real-time analysis, including the middleware overhead, is presented. Finally, an example to generate the implementation from the SDL specification and a performance study is developed.

Keywords

SDL, real-time, scheduler, embedded system

1. INTRODUCTION

The requirements of real-time embedded systems are getting more and more complex and they are difficult to manage with the traditional methodologies used by developers. In this sense, object-oriented methodologies are a good alternative to develop these systems. In [1] we proposed a complete methodology for the design of embedded real-time systems. This methodology proposes to use Formal Description Techniques (FDT) in the design phase and, also, to include the real-time characteristics of the system in this phase. The usage of FDTs provides the basis for an automated design process, allowing simulation, validation and automatic code generation from the specification.

One of the most widely extended Formal Description Technique is SDL (Specification and Description Language). It is an ITU standard [2] and it is currently well supported by commercial tools like SDT [3]. SDL is based on Extended Finite State Communicating Machines. The standard execution model of SDL presents some important real-time anomalies that have to be solved in order to use it to design hard real-time embedded systems. In [4] we proposed solutions for these problems and also we presented a new real-time execution model for SDL. The definition of this new real-time model lets designers to take into account non-functional characteristics (priorities, periods, deadlines, jitter...) in the design stage before the implementation. This way, we increase the attention to the design stage to simplify the implementation stage and to approach both.

This approximation would not be complete if the process scheduling was carried out by the scheduler of the kernel supported by the embedded system that has not take into account the SDL real-time execution model used by the designer. Additionally, it is important to include a theoretical response time calculation based on the SDL execution model. It lets us to know the time that it takes to respond to each event of the system before the implementation and to make the necessary changes in the design in order to meet the timing requirements. This analysis includes the scheduler overhead.

In this paper we present the following contributions:

- A brief resume of the SDL execution model to be able to design hard real-time embedded systems presented in [1].
- A middleware (`Sched_SDL`) that controls the execution of the SDL processes and fulfils the proposed execution model.
- A worst case response time calculation taking into account the SDL real-time execution model and the scheduler overhead.
- An application example and, based on this example, a performance study for this middleware.

The paper is organized as follows: In section 2 we briefly introduce the SDL real-time execution model. In section 3 we present the scheduler design. In section 4 we present the worst case real-time analysis. In Section 5 we give an example and the performance analysis for the scheduler. Finally, some conclusions and future work are proposed.

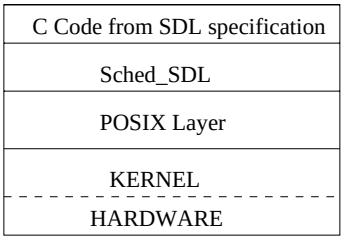

Figure 1: Sched_SDL. Layers

1.1 Related Work

In [6] SDL is used for the co-design of an ATM Network Interface Card. An important conclusion of this work is that SDL is clearly very suited for the specification of control and protocol systems. In [7] a fully specification in SDL of communication software is proposed and the final implementation is derived from there. An annotated concept for SDL is presented in [8] which allows for the specification of non-functional aspects.

Another very important work is related to the C-Micro scheduler [3] created by Telelogic and very suited for embedded real-time systems. It is installed directly over the hardware of the embedded system. The priority assignment is by SDL process or by signal and the scheduling can be preemptive. However, it presents some anomalies to deal with hard real-time embedded systems. For example, the scheduler is enabled when there is a signal sending or the SDL transition reaches to another state. In this case, if a timer expires while a transition is executing then it will be met when this transition sends a signal or finishes. It causes an important delay because it is possible that the transition that is enabled when this timer expires is a higher priority signal. Also, if the priority is assigned to the signals, it is possible that the same signal takes part of different events and we need to assign different priorities for each event. With this kind of priority assignment it is not possible and it is necessary to define another signal.

2. REAL-TIME EXECUTION MODEL FOR SDL

In this section we summarize the SDL real-time execution model. It is formalised in [1]. The execution model is based on fixed priority preemptive scheduling, however we do not assign fixed priorities directly to processes but to process transitions. Process priorities can vary from one state to another depending on the transitions that it can carry out in the current state (taken into account the queued signals). Processes are scheduled according to these dynamic priorities, although the schedulability analysis is based on the transition priorities, which are fixed. Transitions can be preempted by higher priority ready transitions of other processes, but never by a transition of the same process, i.e. if a process transition with higher priority becomes ready while it is executing another transition, this transition is delayed until the current one has finished. This may cause an increment of the response time of events, but this constraint is necessary in order to maintain SDL process execution semantics. Assuming this, processes are preemptively scheduled according to its dynamic priority.

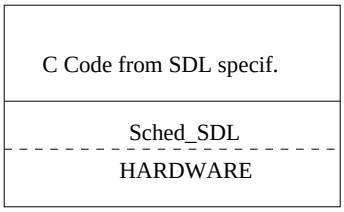

Figure 2: New Schema for Sched_SDL

3. SCHED_SDL: THE SCHEDULER

As we commented in the introduction, it is necessary a middleware that implements the execution model to derive the final implementation of the system directly from the SDL specification. The layer where `Sched_SDL` is built can be seen in the figure 1. It is implemented on the POSIX layer and it runs on the operating system of the embedded system. The generated code from the SDL specification is composed by a set of threads corresponding to each of the SDL transitions in the specification. In this sense, we talk about "transition" from the design point of view or "thread" from the implementation point of view.

Our first prototype has been implemented using POSIX 1003.4a on UNIX workstation. However, we are currently developing a new prototype that uses an implementation of the standardized POSIX Minimal Realtime System profile [10]. This implementation constitutes a kernel for high efficiency small embedded systems where we can include hard real-time requirements. This way, we have the new schema shown in figure 2.

`Sched_SDL` is composed by three important elements (see figure 3):

- Thread `th_sched` that selects the thread (transition) that has to execute.
- Thread `th_timer` that captures the interruptions in the system due to the timer expirations.
- A queue ordered by priority to store the threads ready to execute with less priority than the thread currently executing and the threads that have been suspended by higher priority threads.

When a timer expires an interruption is raised and it is handled by `th_timer`. It wakes up to `th_sched` that is the one that controls who is the thread that has to execute. This way, `th_shed` enables or suspends the threads, wakes up suspended threads and, also, it interacts with the queue to get into lower priority or suspended threads. Additionally, it actualizates the process state after the thread finishes its execution.

3.1 Thread th_sched

This thread is suspended waiting for one of these situations:

- A timer expiration occurs, that is, an external event is received and a thread wants to execute in the system.

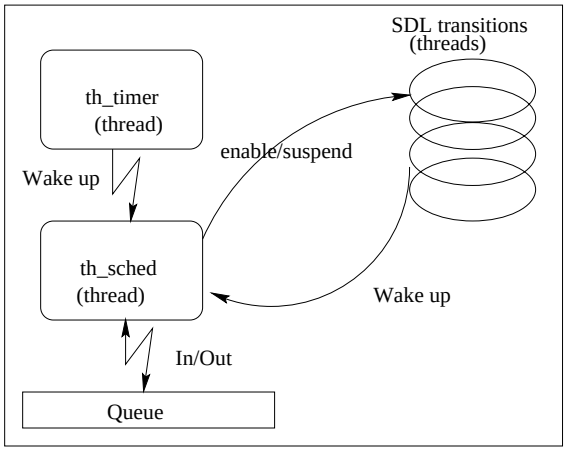

Figure 3: Sched_SDL: Main Elements

- The transition that is executing carries out some signal sending to activate other transition.

This way, the scheduler wakes up when a new transition wants to execute. However, it can execute if it fulfils these three conditions defined in the execution model in section 2:

1. The priority of the executing transition is less than the priority of the enabled one.
2. The SDL current process state where the enabled transition is included coincides with the state where it can execute.
3. The transition that is executing does not belong to the same SDL process than the activated transition. It is due to SDL semantics.

It can be summarized in the following pseudocode:

```
Initiate Mutual Exclusion
Event (Timer expiration, signal sending)
IF Timer expiration (for example, the activated
    transition is A) THEN
    IF Some Transition executing THEN
        IF (SDL Process state is OK AND A is
            the highest priority AND
            SDL processes are different) THEN
            Suspend the executing transition
            Activate the associated thread to the
            highest transition (A)
        ELSE
            Insert in the queue
        END
    ELSE
        IF SDL Process state is OK THEN
            Activate the associated thread
        ELSE
            Insert in the queue
        END
    END
```

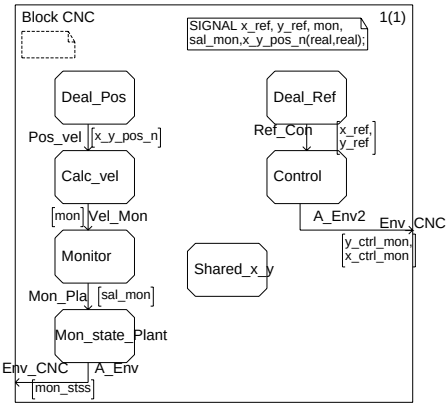

Figure 4: SDL processes of the CNC machine

```
ELSE (* A signal sending *)
    Actualizate the SDL Process state
    Insert in the queue
    Select the highest priority in the queue
    IF the selected transition is suspended THEN
        Wake up the selected transition
    ELSE
        Activate the selected transition
    END
END
End of Mutual Exclusion
```

3.2 Thread th_timer

Additionally, we need to handle the different periodic events that are produced as a consequence of timer expirations. For each timer expiration an interruption is raised and it is handled in the following way:

- The timer is set again.
- It analyses the SDL process and the associated thread that has to execute as a consequence of the expiration.
- `Th_sched` is waken up to analyze if the enabled transition has to execute or if it has to be got in the queue.

4. WORST CASE RESPONSE TIME

Once we have proposed the SDL execution model and a middleware that schedules the different SDL transitions carrying out this execution model, we can complete it presenting a theoretical study of the worst case response time to know, in the design stage, if the system meets the timing requirements. If it did not meet them then designers would be able to make some changes in the SDL design before the implementation.

To calculate the response time of a transition in a SDL system, we take into account the interference of the higher priority transitions and the blocking time of the lower priority transitions. We can add the following aspects to get accurate results:

- There exist precedence relations between the transitions that respond to an external event. It can reduce the number of transitions that participate in the interference calculation.
- Also, the SDL semantics influences in the response time.

Also, we take into account the following restrictions:

- We suppose that the deadlines are lower than the period of the events.
- We consider that all SDL processes of the system execute in the same processor.

In the following subsections we consider a set of external events $S_1, ..., S_n$ and $SEQ(S_i)$ as the sequence of transitions $t_{i1}, ..., t_{im_i}$ that respond to every external event S_i where $1 \leq i \leq n$. Every transition belongs to an SDL process in the system, and corresponds to the reception of a signal.

4.1 Worst case interference

To formally define the worst case interference, we need to introduce some definitions. Let us consider function $P_{sdl} : Signal \rightarrow Process$ which returns, for each signal (viewed as a transition), the SDL process that it belongs to.

Given an arbitrary event S, we define $SEQ_{p,q}(S)$ as the subsequence of transitions belonging to event S starting from transition pth and finishing in transition qth, that is, if $SEQ(S) =< t_1, ..., t_m >$ then $SEQ_{p,q}(S) =< t_p, ..., t_q >$, $1 \leq p \leq q \leq m$.

Given a transition t' of an event S' we define the sequence of transitions with higher priority than t' as:

$HP_{t'}(S) = \{SEQ_{p,q}(S) : \forall t \in SEQ_{p,q}(S)\ pri(t') \leq pri(t)$ and $p_{sdl}(t) \neq p_{sdl}(t')$ and $1 \leq p \leq q \leq n\}$.

$HP_{t'}(S)$ includes the subsequences of transitions that respond to event S with higher priority than t' and that do not belong to the same SDL process. Since this set includes sequences that are included in longer sequences we define $HP^p_{t'}(S)$ to select the longest sequence that belongs to $HP_{t'}(S)$ and begins in p:

$$HP^p_{t'}(S) = \begin{cases} SEQ_{p,q}(S) & if SEQ_{p,q}(S) \in HP_{t'}(S)\ and \\ & SEQ_{p-1,q}(S) \notin HP_{t'}(S)\ and \\ & SEQ_{p,q+1}(S) \notin HP_{t'}(S) \\ <> & otherwise \end{cases} \quad (1)$$

where $<>$ represents the empty sequence.

In order to calculate the worst case interference time of a transition t' that belongs to event S' we do the following steps:

- Select the transition sequences that belong to $HP_{t'}(S)$ for each event S in the system. Several sequences can appear in every event.
- For each event S, we consider its longest initial sequence if it is in $HP_{t'}(S)$, that is $HP^1_{t'}(S)$, and we sum up all of them. We always select them because they have not precedence constraints.
- Also, we select the worst case interference of the rest of the subsequences of all the events that can interrupt to transition t'. Note that only one of them can interrupt transition t' in a monoprocessor system; for this reason we only consider the sequence with the worst case interference.

We formalize these ideas in the following expression:

$$WI(t') = \sum_{i=1}^{n} I(t', HP^1_{t'}(S_i)) + \max_{i=1..n}\{\max_{j=1...m_i}\{I(t', HP^j_{t'}(S_i))\}\} \quad (2)$$

where $WI(t')$ calculates the worst case interference of all the subsequences of the system that can preempt to transition t'. The above definition depends on the interference caused by a sequence of transitions over transition t', which is defined as:

$$I(t', SEQ_{p,q}(S)) = \sum_{t \in SEQ_{p,q}(S)} \left\lceil \frac{R_{t'}}{T_t} \right\rceil * c_t \quad (3)$$

where T_t is the period of transition t and c_t is the worst case execution time of transition t. $R_{t'}$ is the response time of t', which can be obtained upon equation 4.

4.2 Blocking time

In order to include the blocking time in the schedulability analysis we have to consider two possible blocking sources:

- The first blocking source is devoted to the access to shared resources. As we proposed in [4], we encapsulate these shared resources in passive processes accessed by means of remote procedure calls (RPC). Using the highest locker protocol, this blocking time is bounded. We call $B_{sh}(t)$ to this blocking source.
- The second possible source is due to the "run to completion blocking", $B_{rtc}(t)$. It is due to the SDL semantics, since no higher priority SDL transition can preeempt to a lower priority SDL transition if both belongs to the same SDL process.

The expression of B_{rtc} for transition t that belongs to S is the following:

$B_{rtc}(t) = max\{c_{t'} : p_{sdl}(t') = p_{sdl}(t)$ and $pri(t') < pri(t)$ and $t' \notin SEQ(S)\}$

The blocking time is the maximum worst execution time among the transitions that belong to the same SDL process and have a lower priority than the priority of the analysed transition t. We do not include transition t if it is not shared by more than one event.

If transition t takes part in the response to two external events, S and S', the same transition cannot execute concurrently answering both events. In this case the transition itself has to be considered like a possible blocking. In this case the blocking expression is:

$$B_{rtc}(t) = max\{c_{t'} : p_{sdl}(t') = p_{sdl}(t) \text{ and } pri(t') < pri(t)\}$$

The expression of the response time of transition t, R_t, that participates in the response of external event S is the following:

$$R_t = c_t + WI(t) + \max\{B_{sh}(t), B_{rtc}(t)\} \quad (4)$$

4.3 Including the Scheduler Overhead

In section 3 we talk about the details of how our scheduler is implemented. The scheduler adds an overhead in the threads (transitions) scheduling. In this section we complete the previous analysis in order to take into account this overhead introduced by the scheduler.

We add the following overhead sources:

- Worst time due to the context switch, c_{sw}, between the scheduler and the different threads of the designed system.
- Worst time spent to deal with a timer expiration, c_{timer}.
- Worst time due to the scheduler execution, c_{sched}.

If a timer expires, it always interrupts the executing thread and it spends the time to handle the timer expiration and the scheduling. Additionally, three context switches are added. One from the executing thread to thread `th_timer`, another one from thread `th_timer` to thread `th_sched` and the last one from `th_sched` to the thread selected to execute. This way, we add to equation 2 expression $over(S_i)$:

$$over(S_i) = \left\lceil \frac{R_{t'}}{T_t} \right\rceil * (c_t + 3 * c_{sw} + c_{sched} + c_{timer}) \quad (5)$$

where $t \in SEQ_{1,1}(S_i)$.

Also, every preemption of one thread by another thread could results in two context switches: One of them from the lower priority thread to the preempting thread (it is included in the previous expression) and one context switch back again when the preempting thread completes. This way, equation 3 is modified:

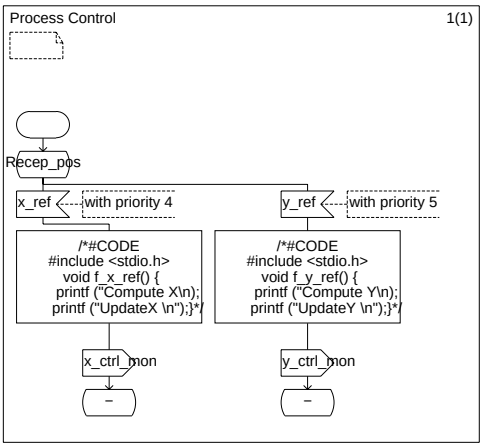

Figure 5: SDL process Control. Specification

Event	*SDL Proc*	*Threads*
Calc. cutter X pos.	Deal_Ref, Control	2
Calc. cutter Y pos.	Deal_Ref, Control	2
Calc. disturbance	Deal_Pos, Calc_Vel, Monitor, Mon_state	4

Table 1: SDL Processes and threads

$$I(t', SEQ_{p,q}(S)) = \sum_{t \in SEQ_{p,q}(S)} \left\lceil \frac{R_{t'}}{T_t} \right\rceil * (c_t + c_{sw}) \quad (6)$$

Finally, we have to add two context switches (in/out of the analysed thread) to equation 4.

$$R_t = c_t + 2 * c_{sw} + WI(t) + \max\{B_{sh}(t), B_{rtc}(t)\} \quad (7)$$

5. EXAMPLE

In this section we present a computerized numerical control (CNC) machine that has been designed with SDL [11]. A CNC machine is an automatic machining tool that is used to produce workpieces designed by users. This controller should be able to position the cutter precisely and automatically along the reference trajectory of the machined workpiece. In figure 4 we show the SDL process design and in figure 5 we show the specification of one SDL process of the system. There are three periodic events: The first one calculates the `X` position of the cutter and the second one calculates the `Y` position. Both of them have a period of 400 ms. The last one estimates the disturbance and monitors the plant with a period of 600 ms. In table 1 we show the SDL processes that take part in the different events if we take into account the SDL design and the number of threads that take part in the event execution if we take into account the generated code from SDL (Figure 5). The number of threads generated from the SDL specification is eight. We take the SDL processes to generate a set of threads that execute the `C` code of each transition together with the middleware proposed. It is executed on the operating system of the real-time embedded system.

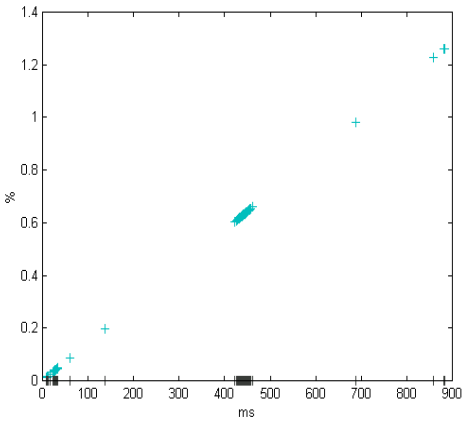

Figure 6: Performance of th_sched

Event	*WorstExecutionTime*
th_sched (Scheduling)	0.460 ms
th_timer (Timer Handling)	0.607 ms
context switch	0.04 ms

Table 2: Measures of Time

5.1 Performance

In this subsection we evaluate the scheduler performance with respect to the previous example. We calculate the scheduler overhead of the CPU time corresponding to the execution of the CNC machine. Moreover, we compare the thread theoretical response time, including interrupt delay, context switch time and delay scheduling, and the real response time. As we can see in figure 6 the higher overheads (around 0.7%) are when the scheduler has to suspend lower priority threads. The average overhead is 0.4%. Additionally, we show the time spent for threads `th_sched` and `th_timer` and for the context switches in table 2.

As it is known, the theoretical results are an upper bound of the response time. We include two theoretical response time calculations: The first one calculates the event worst case response time [9] and the second one is based on the equations proposed in section 4. This way, theoretical and real results are closer. We show in figure 7 a comparison with the real response time for event `Y position` and theoretical results. As we can see in the figure, theoretical results that do no take into account the SDL execution model are not an upper bound.

6. CONCLUSIONS AND FUTURE WORK

In this paper we propose to use FDT as a good alternative in the design of hard embedded real-time systems. We present a way to implement this kind of system directly from the SDL specification attending to the real-time SDL execution model. Moreover, we present a middleware to control the execution of the tasks in the system and a real-time analysis.

Currently, we generate code from a basic SDL set and, as future work, we want to complete the code generation. Additionally, we will try to improve the scheduler performance reducing, for example, the number of context switches. Finally, we want to prove the prototypes in a target platform.

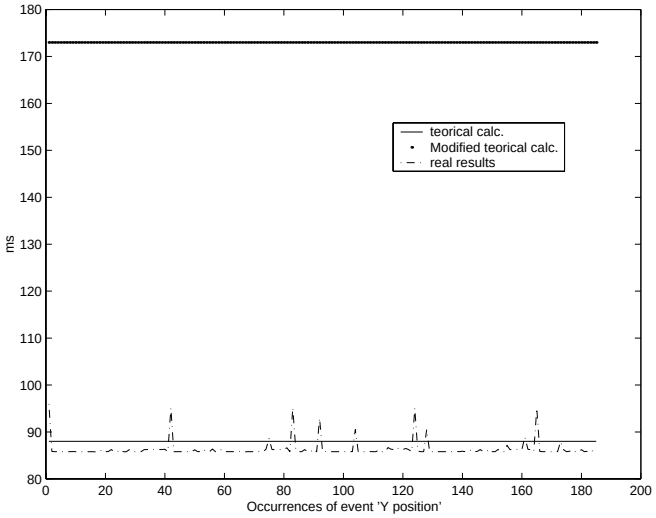

Figure 7: theoretical and real comparison

7. REFERENCES

[1] J.M. Alvarez et al. *Integrating Schedulability Analysis and SDL in an Object-Oriented Methodology* 9th SDL Forum. Elsevier. 1999

[2] *ITU recommendation Z. 100. Specification and Description Language (SDL)* 1994

[3] *SDT 3.5 Manuals* 1998

[4] J.M. Alvarez et al. *An Analysable Execution Model for SDL for Embedded Real-Time Systems* Workshop on Real-Time Programming (WRTP99). Elsevier 1999

[5] J.M. Alvarez et al. *Schedulability Analysis in Real-Time Embedded Systems Specified in SDL* Workshop on Real-Time Programming (WRTP00). Elsevier 2000

[6] N.E. Zergainoh et al. *Using SDL for Hardware/Software Co.Design of an ATM Network Interface Card* 2nd Workshop on SDL and MSC (SAM98). 1998

[7] W. Dulz et al. *Early Performance Prediction of SDL/MSC Specified Systems by Automatic Synthetic Code Generation* 9th. SDL Forum. Elsevier. 1999

[8] S. Spitzet al. *SDL*- An Annotated Specification Language for Engineering Multimedia Communication Systems*

[9] Klein, M.H. et al. *A Practitioner's Handbook for Real-time Analysis* Kluwer Academic Publishers 1993

[10] M. Aldea and M. Gonzalez *Early Experience with an Implementation of the POSIX.13 Minimal Real-Time Operating System for Embedded Applications* 25th IFAC Workshop on Real-Time Programming. 2000

[11] N. Kim, M. Ryu, S. Hong and H. Shin *Experimental Assesment of the Period Calibration Method: A Case Study.* Real-Time Systems Journal , 1-26(1999). Kluwer Academic Publishers. 1999.