

New Functions, New Sensors, New Architectures – How to Cope with the Real-Time Requirements

T. Kramer, Dr. R. Münzenberger, INCHRON GmbH

Abstract

The detection of the omni-present event chains in embedded applications goes far beyond functional modelling and static analysis. Once identified the analysis of their dynamics reveals a lot of data about the system like stability, critical paths or load reserves for future extensions. By using Task-Models and a real-time simulation tool the detection and analysis of event chains is very easy. Especially in distributed and collaborative development environments this is very helpful in reaching not only functional perfect systems but also delivering a high level of real-time quality.

1 High Quality in Spite of Growing Complexity

The majority of the upcoming smart system technologies and Information and Communication Technologies (ICTs) in the automotive industry combine existing and new sensors, build new distributed functions and require networked architectures. They span multiple domains and break up the existing one-vendor-one-box-principle. But how can such a complex system be architected and built to reach high quality levels in short cycles while combining several technologies and contributing parties?

The aspired high quality covers different aspects. The dominant aspect is functional quality. This is well served by established methods and tools like model driven development. Often the process starts with requirements gathering, model creation, implementation in code. It can end at different levels of integration. Parallel to this a variety of quality tests are specified and executed to assure that the system matches the requirements at a high functional quality. But quality also demands reliability and robustness. The system shall generate the correct functional results at the desired point in time – reliably, repeatable and under all circumstances. These timing and robustness aspects do not share the same level of tool support so far. Nevertheless they do have a tremendous impact on the system's quality and on the development time.

This paper will describe how distributed embedded systems' soft- and hardware can be modelled to describe their dynamic real-time behaviour. These models can be executed and visualized on the real-time simulator chronSim or analyzed by the real-time validator chronVal. The execution timing, events and sequence of functions and communication is calculated and displayed in various diagrams. The worst case and distribution of execution times can be determined for single functions or complete systems.

In an example from the automotive industry a system will be discussed, where a function is implemented on multiple control units using FlexRay for inter processor communication. The application tasks generating and receiving data run asynchronous to the FlexRay bus on separate control units due to application specific cycles.

2 Event Chains and their Timing Aspects

Automotive electronic systems are built from various sensors, actuators, communication busses and controllers. They are highly integrated and distributed systems with real-time requirements performing discrete and continuous control tasks. To work properly the systems have to execute the functions in a logical correct order with the specified reaction to the stimulation. But not only the reaction has to be correct, it also has to occur at the right point in time.

A proper method to describe and test the system's reaction to stimuli is to follow the signals and functions along the various paths through the systems. The stimulus can be a pressed button or a sensor signal. But it can also be a group of signals or a defined system state that will lead to the reaction. And the reaction can be dependent on the history of these subsequently calculated system states. As a result there are many different paths through the system.

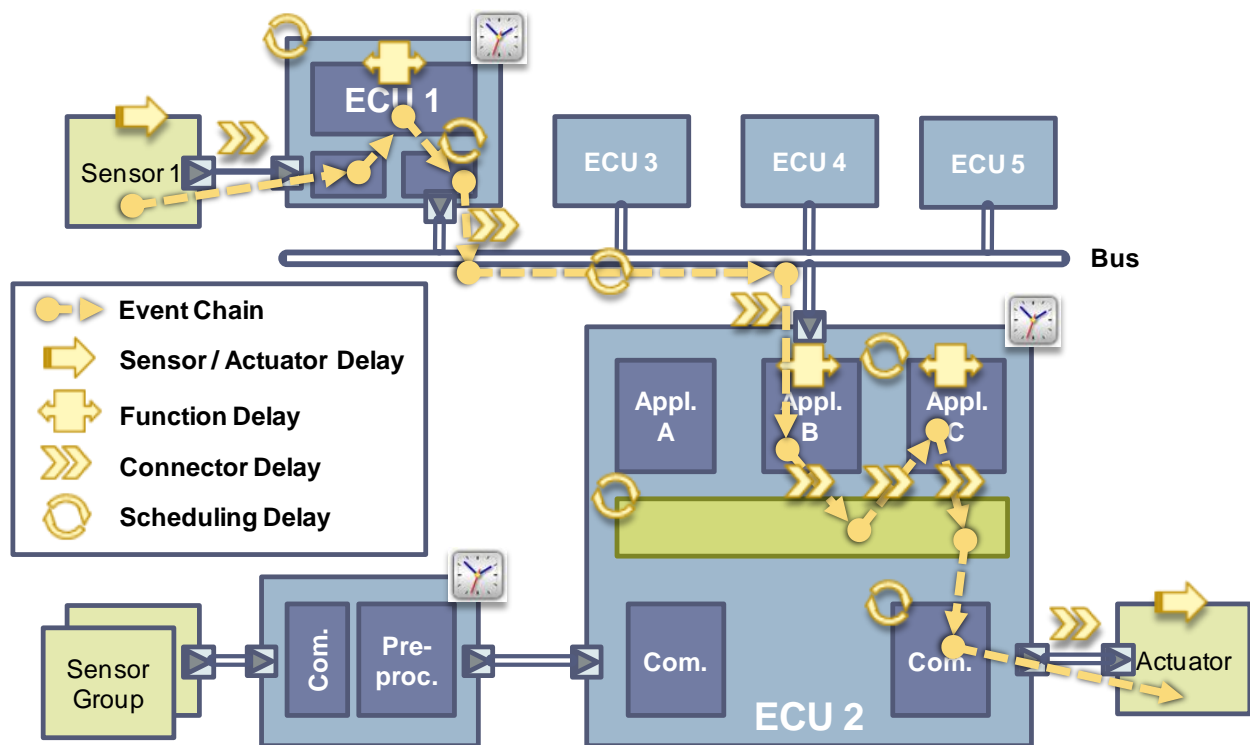


Fig. 1. Block diagram of a distributed system highlighting the delays along the complex event chain.

The system depicted in figure 1 highlights a path from the sensor 1, through ECU1, the bus, into ECU2 to the applications B and C. Finally the signal is sent over a communication interface to the actuator. But there are many more paths in that system. A stimulus from the sensor group will trigger the preprocessing that sends the combined signal data to ECU2. From here multiple reactions can be initiated and could stimulate ECU4, thus describing more paths.

2.1 Event Chains

Every step along the path through the system will create an event. If these events are ordered in causally relation one will get a chain of events. Event chains (aka effect chains; in German: ‚Wirkketten‘) are a commonly used term to describe the sequence of steps along the critical path performed to fulfill a certain functionality. The event chain has a timing requirement defining the end-to-end time from stimulus to reaction for the function to be correct. Having a common understanding what the event chains are in a system, all developers and architects can design, implement and test the functions properly.

Event chains are system immanent, but not necessarily well known and documented. In simple systems they are easier to identify and understand. In more complex and distributed systems the event chains often are unknown or only determined for fractions of the complete chain. That is because complex systems are developed by multiple groups and even companies. [2] The necessary information is as distributed as the system. The architects and developers often use function models and event chains only with the scope and detail level that is relevant for their own work. The outcome is that the complete information about the path is subdivided over all involved development groups.

Another cause for imprecisely identified event chains is the difficulty to determine the critical path in a system. The example in figure 1 has two sensor inputs, one with ECU1 in between sending the sensor data to ECU2 over a bus, the second sensor is a group of sensors with an additional preprocessing unit. Will the preprocessing or the bus communication add critical delays and jitter to the system timing, thus causing the path to be real-time critical? Especially when signals from different sensors are merged, when parallel processes are providing data for a joint calculation, it is not simple to determine that chain of events causing the critical timing.

Static methods and tools cannot answer these questions. With function models the consecutive steps along a path can be analyzed according their functionality and logic. It can be proved that the functions are called in correct order and a stimulus can produce the correct system reaction. But this implies that all signals are sent and received only

once and that the functions could be executed immediately. In contrast, when running the software on a dynamic system environment, the sensor could be triggered multiple times before the preprocessing has finished. The application can be blocked by higher priority tasks, never finishing the control loop calculations before the next cycle starts. With static analysis it is possible to see what happens to the data, but not when. That makes it very difficult to determine what dynamic timing behavior will cause one or another path to be real-time critical.

2.2 Time Durations of Event Chain Steps

Every component in a system has an execution time tied to it. Each sensor needs time from sensing the physical value to transforming it into an electrical signal. Every command will be executed on the microcontroller in a certain amount of time. So the execution time of every event chain (end-to-end timing) is the sum of all components' delays.

Very often the time slices comprising the end-to-end timing vary because of the physical effects in the sensor or actuator. Other delays are dependent on the then active system state. If for example a communication bus is blocked by higher priority messages, the message carrying the sensor signal will be delayed until its message wins the arbitration. Furthermore the messages currently on the bus are the result of the states of all involved functions communicating over the bus. Thus the message delay is a very dynamic factor.

There are many possible delays in an event chain:

Sensor or Actuator Delay – Depending on the physical principle and the sampling method the reaction time will be constant (e.g. opening an electro-mechanical valve) or vary over a cycle time. For example when using image acquisition with a camera running at 50 frames per second, the acquisition can be just finished or may just have started. So the image can be accessed immediately or at worst after 20 milliseconds.

Function Delay – The software function commands executed on a microcontroller can be a constant set of cycles or may vary depending on the algorithm and values (e.g. division by 1 or by 0.9).

Connector Delay – The software functions or modules communicate over interfaces. These interfaces may interconnect directly with global variables or even use multiple abstraction layers to access different communication methods including in-vehicle busses or internet protocol. The connector delay can therefore depend on the scheduling of the communication layer, the bus scheduling and the current bus load.

Scheduling Delay – Sharing resources (μ C, busses, etc.) requires scheduling to grant or block their access. Requests a step in the event chain a shared resource, the delay depends on multiple factors like priority, time phase compared to other functions' cycles, mapping order within the tasks, and many more. And while a function is executed, the scheduler might suspended it before completion due to higher priority tasks or interrupts.

2.3 Impact of Multiple Clocks

On top of the above mentioned variable delays the fact that the system has multiple clocks adds another dynamic timing factor to the critical path. Clocks and periodic events play an important role in the timing of event chains. The components involved to perform the function often have their own cycles and time bases. That might be the revolutions of the crank shaft or wheels, the cycle times of the real-time bus FlexRay and not to forget the CPU clocks in the ECUs. The applications on the CPUs are often triggered periodically by the scheduler deriving its time base from the individual CPU clock. Every clock and cycle has a start time, cycle time and phase. All three parameters can vary showing drift, jitter and sometimes may drop out.

The examples in figure 2 and figure 6 show the effects, that drifting clocks produce. Event chain latency will vary significantly and even 'jump' from the best to the worst case in one cycle. Having passed the data to the next step right before the time slot, in the next cycle it can miss the time slot. The data transfer will have to wait for a complete cycle of the receiver to take place.

To reduce the variation of all cycles and the resulting large jitter in the end-to-end timing some of these clocks can be synchronized to a master clock. Or, what has been the choice in the past, the application is designed to run on one CPU only and is using an event-based communication. With the growing number of functions being distributed over a networked system, this design choice has become rare.

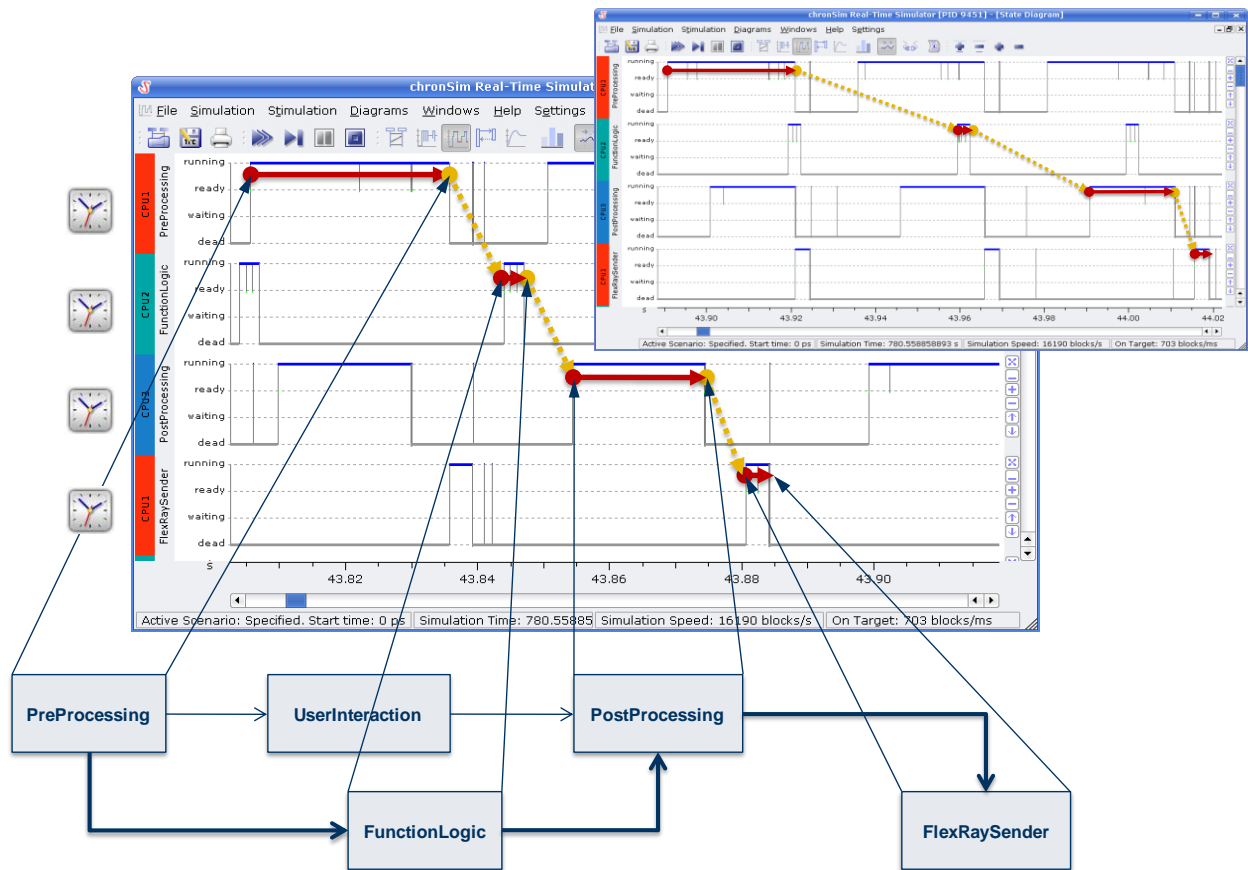


Fig. 2. Task state diagram with short and long event chain

Figure 2 depicts an example how drifting clocks can lead to very short and much longer event chain latencies. The screen shots from chronSim show the states of the tasks executing the functions referenced in the block diagram underneath. Since CPU1 (red; 1st and 4th plot), CPU2 (green; 2nd plot) and CPU3 (blue; 3rd plot) have own clock sources they may drift against each other. So the cycle phases of the tasks can be ideal for the event chain, the functions can be executed ‘back-to-back’ and the latency will be small. But as shown in the small screen shot, the adverse cycle phases will lead to large latencies. None of the situations represents a malfunction according to the designed behavior – but the event chain’s latency might violate a timing requirement.

3. Creating a Real-Time Simulation Model

One approach to develop an embedded system is to define a soft- and hardware architecture, write all code, build the hardware, integrate the system and test if it fulfills the requirements. If it does not pass the tests look for faults, correct them and start all over again. Obviously this trial and error method cannot be the best way to reach the goals. And for the functional requirements this has not been the chosen method. Instead functions are modeled, the models tested and refined until they could be used to generate the function code. The software then is tested once again using SiL (Software in the Loop) before it is deployed onto the target hardware.

But for the timing requirements the trial and error method sometimes must be chosen since the timing behavior cannot be measured until the code is executed on the target hardware. This leads to a hen and egg dilemma. The architect has to design a system with timing characteristics he only knows, when the system is built and the timing can be measured. So real-time optimized design processes will iterate over the complete development cycle.

The proposed method to overcome this dilemma is to use real-time modeling to describe the systems and timings. The models can be simulated and analyzed focusing the timing aspects without ignoring the functional dependencies of the event chains. The real-time simulator executes the real-time models and enables the architect to experience the timing of his designed system before it has been implemented. By applying additional stimuli and load to the model the designer can use the simulator to perform sensitivity analysis and test the robustness of his design.

The real-time requirements of the design can be tested with the same models using a real-time validator. The analysis of the system with the validator will show the critical situations leading to the requirement violations. Otherwise it reports the remaining system performance that is available for further expansions.

3.1 Task-Models

Real-time properties and requirements covering all system levels have so far not been standardized or made widely available. For a complete system description there was no versatile tool or data format in use until now. Task-Models can close that gap and enable embedded system developers to cover also the timing aspects when specifying their system using modeling. [1]

A part of the Task-Model consists of model code in standard C or C++. Here the execution times of tasks, functions and software modules are entered. It can be derived by abstracting the target code and annotating it with the target execution time on the level of ISRs, tasks, functions or other software modules. Or it can be the starting point of a project having a skeleton code describing the software structure. Throughout the development it will become more detailed and finally may even be replaced by the target code itself.

```

ISR(ISR_Rotation)
{
    DELAY(30,unit_us);
    ActivateTask(Task_Rot);
}

TASK(Task_Rot)
{
    DELAY(150,unit_us);
    TerminateTask();
}

```

Example of the most simple implementation of a *Task-Model* for OSEK.

The *ISR_Rotation* is executed in 30µs and *Task_Rot* in 150µs.

```

TASK(TASK_TT_5ms) {
    DELAY(300, unit_us); ①
    Schedule(); ②
    while(!finished()) { ③
        DELAY(100, unit_us);
    }
    exectime = 10*data_size;
    DELAY(exectime,unit_us); ④
    DELAY(gaussian(500, 10),unit_us); ⑤
    TerminateTask();
}

```

Example of a more complex Task-Model

- ① : constant execution time
- ② : scheduling point
- ③ : control flow dependent execution time
- ④ : data dependent execution time
- ⑤ : probability distribution of execution time

Fig. 3: Simple and versatile Task-Models

Figure 3 lists on the left side a simple Task-Model for an OSEK OS. The macro DELAY() replaces the function code with its time budget. So the ISR will run for 30 microseconds and then use an OS function call to activate a task. This task will run for 150 microseconds as long as it is not suspended by a higher priority task. If the timing behavior of the modeled system has to be defined more in detail, there are all possibilities that C offers available. The right example lists several methods to vary the execution time depending on system states or other stimuli. This enables dynamic reactions of the simulation model showing the same timing as the real system.

Event chains are specified directly in the Task-Model. By using the EVENTCHAIN() macro the designer marks each step in a chain of causal dependent events. The current instance of the event chain execution as well as the current step within a specific instance can be specified by free variables. By passing their values between different nodes via a communication bus (e.g. within a CAN message) even causal dependencies across ECUs are possible.

```

TASK(PreProcessing){          /* CPU1 */
    Message msg = startProcessing();
    msg.instance = currentInstance++;
    EVENTCHAIN(CriticalPath, msg.instance, 0);
    DELAY(PreProcessingTime, unit_ms);
    sendMsgToFunctionLogic(&msg);
    TerminateTask();
}
...

TASK(FlexRaySender){        /* CPU1 */
    Message msg = recMsgFromPostProcessing();
    EVENTCHAIN(CriticalPath, msg.instance, 3);
    DELAY(FlexRaySenderTime, unit_ms);
    TerminateTask();
}

TASK(FunctionLogic){        /* CPU2 */
    Message msg = recMsgFromPreProcessing();
    EVENTCHAIN(CriticalPath, msg.instance, 1);
    DELAY(FunctionLogicTime, unit_ms);
    sendMsgToPostProcessing(&msg);
    TerminateTask();
}

TASK(PostProcessing){       /* CPU3 */
    Message msg = recMsgFromFunctionLogic();
    EVENTCHAIN(CriticalPath, msg.instance, 2);
    DELAY(PostProcessingTime, unit_ms);
    sendMsgToFlexRaySender(&msg);
    TerminateTask();
}

```

Sample Task-Model with event chain

EVENTCHAIN(ChainName, ChainInstance, ChainPosition)

Fig. 4. Sample code of the event chain in figure 2

Figure 4 lists an example Task-Model that generates the output shown in figure 2. The event chain starts with the task PreProcessing. The creation of a new message leads to the manual increase of an instance counter. The instance counter is used to mark step 0 of the event chain. The instance counter wrapped into the message is transferred to the second task FunctionLogic, then to the task PostProcessing and finally to the task FlexRaySender. In every task the value of the instance counter, the specific step and the actual point in time is recorded. The result is shown in the task state diagrams for the three different microcontrollers CPU1, CPU2 and CPU3. Only this or another analysis of the simulation results reveals the causal and temporal order of the event chain through the system.

To complete the system description the hardware, peripherals and stimuli are described using graphical user interfaces and libraries provided by the simulator tool. The scheduling method and software architecture with the definition of processes, priorities and e.g. stack sizes can be entered as well or be imported from standard OS configuration files like OIL (OSEK Implementation Language).

Task-Models can be adapted to the abstraction level, the development phase, the focused timing detail of all involved development partners. A network designer can model the communication across multiple nodes without having to have detailed models of the nodes' internals. Accordingly the developer of an application part within one node, can model his parts very detailed. But he reduces the communication details and peripheral timing behavior models to simple stimuli sources as far as they influence his applications timing.

3.2 Gathering the Timing Information

For the correct real-time simulation and validation the net execution times of the software modules are required. Depending on the abstraction level these can be the execution times of the tasks and ISRs as fixed values. At a higher detail level the times can be split down to each function or even command. Having mentioned the hen and the egg dilemma, this timing data is not available until the system is developed and integrated.

When designing a system from the scratch the architect has to set certain cornerstones defining the general functionality and timing of the system. At this point he chooses what will be executed in parallel or in sequence. For every function or task the architect sets timing quotas. Throughout the implementation more details become available and can be added to refine the real-time model.

If prototypes or legacy systems are available the timing data can be derived from measurements [3]. By adding instrumentation code at critical points (e.g. start and end of software modules) the system behavior can be traced with timestamps to identify what system state occurred when. The drawback of this monitoring technique is twofold: – the instrumentation commands add code to be executed and therefore change the real-time behavior of the system. – and the trace will cover only short time periods since the memory is limited showing only single

system situations. Nevertheless the trace will provide realistic gross execution times of software modules in different system states. The net execution times can be derived utilizing additional scheduling information.

The real-time simulator also has an optional estimator module with processor models to estimate the execution times of target C code. When the development has progressed so far that target code and compilers are available, the estimator module will use the C code, the processor model and the compiler output to determine static execution time budgets. During the simulation the dynamic effects of pipelines and caches are applied to the static time budgets. This option has its beauty, but requires the code to be available and adds a detail level that can distract from the core problems and questions regarding the real-time behavior.

4 Event Chain Simulation and Analysis

Describing the dynamic behavior of the system using Task-Models all fractioned information can be combined in a single real-time simulation model. The event chain segments from different development teams can be completed and highlighted in the model to be analyzed regarding their timing as a whole. Simulating and analyzing the model will help to understand the timing of each chain and show what the critical path and real-time relevant situations are.

The simulator executes the model and records all system events. In different graphs the task states and process communication is shown (see figure 2). The event chains can be plotted over time to see when and where large delays occur or where oversampling might cause data integrity problems.

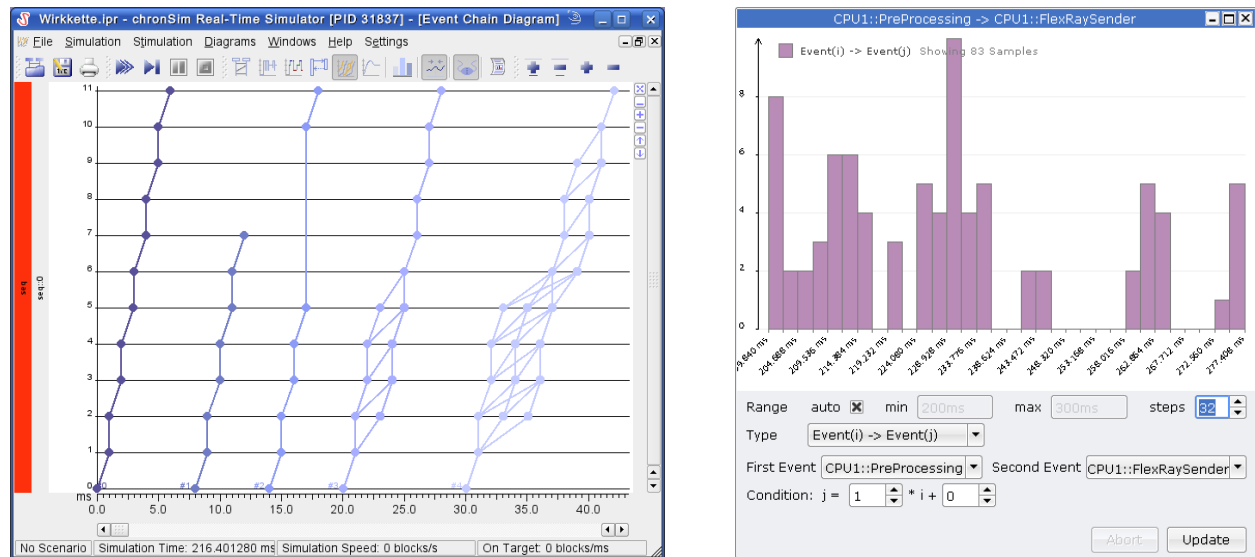


Fig. 5. Event chain diagram and a histogram of the time between two chain events

On the Y-axis of the event chain diagram in figure 5 the steps along the path are listed (parameter ChainPosition in figure 4). The five plotted event chain instances show some possible timing effects. The second iteration ends uncompleted. In the third iteration some steps are never passed. The fourth and fifth iteration show multiple activations beginning with step 2. Here the event chain may continue using older or younger data since the according functions processing the data are called multiple times with the same data from step 1.

The histogram in figure 5 depicts the distribution of time passed between the PreProcessing and the FlexRaySender event.

5 Example Application with FlexRay

In this example two ECUs communicate over a bus to perform a control loop exchanging set values and actual values. That could be e.g. a brake control unit sensing the slippage of the wheels and requesting lower torque from the power train. The power train controller will reduce the fuel injection and reply with the actual torque value. This combination of brake and power train controllers is more efficient than just applying the brakes to prevent the car from sliding.

The event chain in this distributed control loop is spanning two controllers and a bus with an own scheduling and time base. The architects and function developers have to handle this very complex system, where parts of the event chain are defined by engineers from a different development team. Several parameters shall be identified. How long it takes from sending the torque request to the engine until the engine reports back the updated torque value. How much jitter is added through the bus communication. When one ECU fails to send the messages, how fast the system will detect the fault and switch to fail safe mode. How old the data is when it is read by the consumer application. How often messages will be lost without having a system failure.

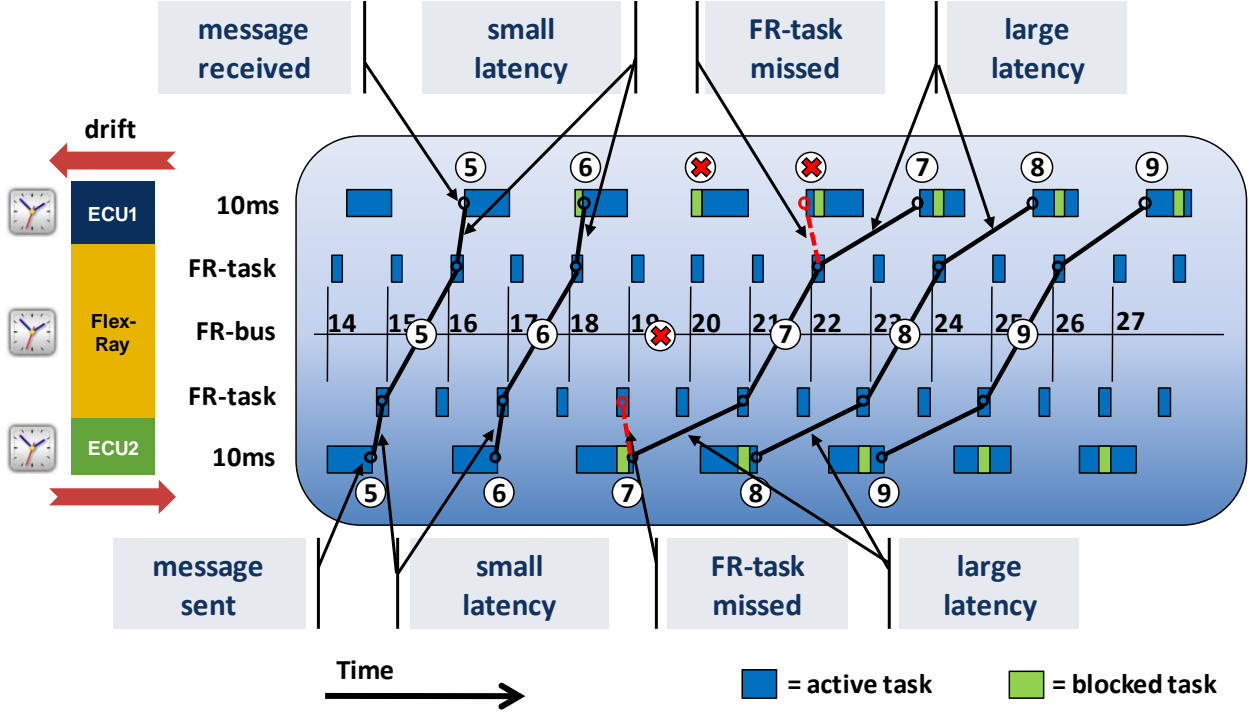


Fig. 6. Two ECUs with drifting clocks show large message latency

Each ECU has a time triggered task architecture that runs the base software, the application and its I/O. From these tasks only the 10ms-tasks are shown in the diagram in figure 6. The deterministic FlexRay has its own time base running unsynchronized to the ECU's clocks. But the FlexRay communication software is running synchronized to the bus. Therefore the application tasks can drift against the FR-task in each ECU. In this case the ECU2 clock runs slower than the FlexRay time base. The high priority FR-tasks on the ECU start to block the lower priority 10ms-tasks (green blocks; starting with FlexRay cycle 19). For the receiving ECU1 the drift has the opposite direction. The ECU clock runs faster than the FlexRay clock and the blocking starts already at cycle 18.

The event chains in figure 6 are graphed as black lines with small circles. Instance 5 and 6 of the event chain have a small latency since the producing tasks (10ms-task from ECU2) and consuming tasks (10ms task from ECU1) send and receive shortly before and after their FR-tasks. Beginning with instance 7 the producer ECU2 cannot send the data to the FR-task in time and misses the sending in FlexRay cycle 19. 'Unfortunately' at the same time the receiving FR-task on ECU1 misses the 10ms-task because of the drift. As a result the event chain (instance 7) has a large latency and the consumer receives the data missing two iterations of the 10ms-task.

The effect of one 10ms-task missing its FR-task will occur frequently due to the drift. But having it happen on the sending and receiving side in the same event chain instance is rare and very hard to reproduce on the test bench or test drive. Nevertheless it can occur and has to be considered e.g. when a message counter would detect this as an error subsequently turning off the system.

In the under laying customer project the ECUs came from different suppliers. Each supplier could not test against this effect since he never had all components of the system. Only when both ECUs were integrated into a test system at the OEM, the complete event chain could be executed and tested – quite late in the development process. Using the Task-Models both suppliers and the OEM could discuss the effects based on a common understanding. The improved collaboration led to discussions about ways to synchronize parts of the application to the FlexRay. Different ideas could be modeled and evaluated using the simulation much faster with fewer resources.

6 Conclusion

When complex embedded systems are developed by multiple parties, defining and fulfilling the real-time requirements is one of the most challenging tasks. Task-Models for real-time simulation and analysis can provide the versatile basis to identify, describe and test the real-time critical paths throughout the systems. By marking the event chains in the simulation model their dynamic behavior can be observed and optimized very efficiently. The quality of real-time embedded systems regarding reliability and robustness will increase significantly due to timing optimized architectures and thorough real-time testing.

The collaboration support that Task-Models offer becomes mandatory when AUTOSAR comes true. Software will be developed by various development teams but integrated by others. Having a model based solution to describe, architect and integrate real-time critical software from multiple parties will complete the currently available toolsets for AUTOSAR mainly focusing the functional aspects only.

References

- [1] Komarek T., Dörfel M., Münzenberger R., Developing Real-Time Constrained Embedded Software Using Task Models, in proceedings of the Advanced Automotive Electronics (AAE 2007) Gaydon www.aae-show.co.uk, January 2007
- [2] Münzenberger R., Kramer T., Collaboration Support to Master Real-Time Challenges, in proceedings to the embedded world conference 2009, March 2009
- [3] Kramer T., Münzenberger R., With Measurements to Real-Time Simulation, in proceedings to the embedded world conference 2009, March 2009

Dipl.-Ing. Tapio Kramer

INCHRON GmbH
Lichtenbergstr. 8
85748 Garching b. M.
Germany
Tel.:+49 89 5484-2960
Kramer@INCHRON.com

Dr. Ralf Münzenberger

INCHRON GmbH
August-Bebel-Str. 88
14482 Potsdam
Germany
Tel.:+49 331 97992-232
Muenzenberger@INCHRON.com

Keywords: real-time, simulation, validation, embedded systems, event chain, collaboration, requirements