Standard Software Components for X-by-Wire Networks

Thomas M. Galla
DECOMSYS GmbH
Stumpergasse 48/28
A-1060 Wien
Tel: +43 (1) 59983 15
FAX: +43 (1) 59983 715
galla@decomsys.com

Jochen Olig
3SOFT GmbH
Frauenweiherstrasse 14
D-91058 Erlangen
Tel: +49 (9131) 7701 120
FAX: +49 (9131) 7701 333
Jochen.Olig@3SOFT.de

1 Introduction

The use of real-time operating systems has become well established in the automotive industry within the last 10 years. The OSEK/VDX standard was the result of the endeavors of major car manufacturers and their suppliers to create a standardized software infrastructure for automotive electronics. The basic considerations were driven by the awareness that the continuously increasing complexity of the various control units can only be managed through well defined interfaces. With continuously shortened development cycles, especially in the electronics area, requirements arise concerning compatibility, reusability and increased test coverage that can only be fulfilled by setting standards for the various system levels. Reusability and standardization are seen as a basic step in the development of more reliable software, a fact that is demonstrated by the recent founding of the AUTOSAR consortium (AUTomotive Open System Architecture).

Today, OSEK/VDX conformant operating systems are widely used. This standard was designed for applications in the area of car body electronics or for the power train where autonomous control units build up a loose network. Currently, new and challenging applications are appearing to make driving safer by relieving the driver from routine jobs. These driver assistance systems need to have direct control over their related actuators. With these X-by-Wire applications, all mechanical and hydraulic couplings are replaced by fault-tolerant electronic components. Typical examples include electronic braking (Brake-by-Wire) and electronic steering (Drive-by-Wire). Due to their inherent safety-criticality, these systems require reliable and secure implementation methods since the failure of a single component may have fatal consequences.

Because these X-by-Wire systems are implemented in a distributed way, the various components in the resulting cluster must operate while being synchronized within tight time intervals. The event driven mechanisms provided by OSEK/VDX do not fulfill these properties since they do not provide a global time distributed over a network. Therefore, the simultaneous operation of different control units at a single point in time is not guaranteed. OSEKtime, available in Version 1.0 since July 2001, is the specification of a time-driven operating system including a communication layer that meets the requirements for real-time capability, synchronization, and fault tolerance. The new standard consists of two parts. The first part concerns the operating system (OSEKtime OS [3]) which provides features such as tasks and interrupt service routines and which controls their execution with respect to time. The second part concerns the middleware layer (OSEKtime FTCom [4]), which is responsible for the fault-tolerant communication between the single electronic control units (ECUs). OSEKtime FTCom also offers a global time-base that is used by all network partners to coordinate their activity amongst themselves. The full functionality required by X-by-Wire systems can only be achieved through the correct cooperation between the operating system and the communication system.

In their joint product *TimeCore* 3SOFT and DECOMSYS combined 3SOFT's OSEKtime implementation ProOSEK/time, DECOMSYS' development tools (e.g., DECOMSYS::GENERATOR for the generation of an OSEKtime FTCom layer), and a set of standard software components like transport layer [1] or network management [2] into an integrated platform for the development of such X-by-Wire systems.

2 OSEKtime OS

OSEKtime OS is designed as a time-driven single chip operating system for distributed embedded systems. It distinguishes between two logical parts, which can execute functions. These are *interrupt service routines* and *tasks*. Interrupt service routines (ISRs) execute interrupt related services. Tasks, however, are started at defined points in time and have one of three different states. In the *running* state, the CPU is assigned to the task and its commands are executed. Only one task can have this state at any point in time while all other states can be adopted simultaneously by several tasks. The *preempted* state is reached by a task that has been in the *running* state and which was preempted by another task that is to be activated. A task can leave the *preempted* state only if the preempting task changes into the *suspended* state. An inactive task that can be activated is in the *suspended* state. These state changes are depicted in Figure 1.

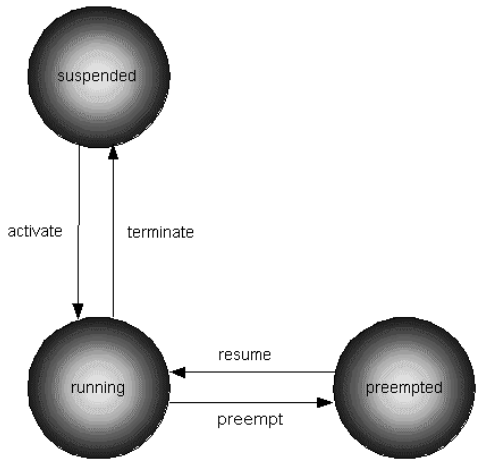

Figure 1: OSEKtime Task States

The activation times of the tasks are stored in a so-called dispatcher table before compile time. The dispatcher is the central component of the OSEKtime OS and is responsible for activating the tasks according to their activation times. The processing of the dispatcher table is done cyclically. A complete cycle of the dispatcher table is called a dispatcher round. If a time-driven task is still in the state *running* when the activation time of another time-driven task is reached, the first task passes over to the *preempted* state and stays there until the interrupting task is finished. This kind of scheduling is called stack-based scheduling; no priorities are assigned to the tasks - only the activation times define the precedence between different tasks. The later a task is started the less likely it is that another task can interrupt it while it is being executed.

Depending on the design of the application, a dispatcher round can include times during which there are no tasks or interrupt service routines active. In these idle times, a task named *ttIdleTask* is assigned to the CPU. This task is provided by the OS, has no entry in the dispatcher table and no deadline assigned. Non time-critical jobs can be executed in this task.

Usually the dispatcher is triggered from a non-maskable interrupt whose frequency is coupled to the local time-base. At the end of a dispatcher round, after the last time-driven task is finished, the local time is synchronized with the global time. This is achieved by prolonging or shortening the current dispatcher round in an adequate manner.

Figure 2 shows the functionality of the dispatcher and the principle of the stack-based scheduling. At the beginning of the dispatcher round, the OSEKtime dispatcher is activated and starts task TD1 (point in time t_1). At the next activation time (point in time t_2), the dispatcher preempts task TD1 to activate task TD2. Task TD1 changes into the *preempted* state. After TD2 is finished, the dispatcher once again becomes active and resumes the execution of the interrupted task TD1. After this task is finished, there is no time-driven task to be executed at this point of time and therefore the *ttIdleTask* is activated. As soon as t_3 is reached, ttIdleTask is interrupted and the execution of task TD3 is initiated.

In the dispatcher table a deadline can be defined for every task. The monitoring of the deadline is done by the dispatcher. If a task has not finished when its deadline is reached, the operating system will be shut

down. If there is no deadline for a specific task then it has to be finished by the end of the dispatcher round. If this is not the case, the system will shut down as well.

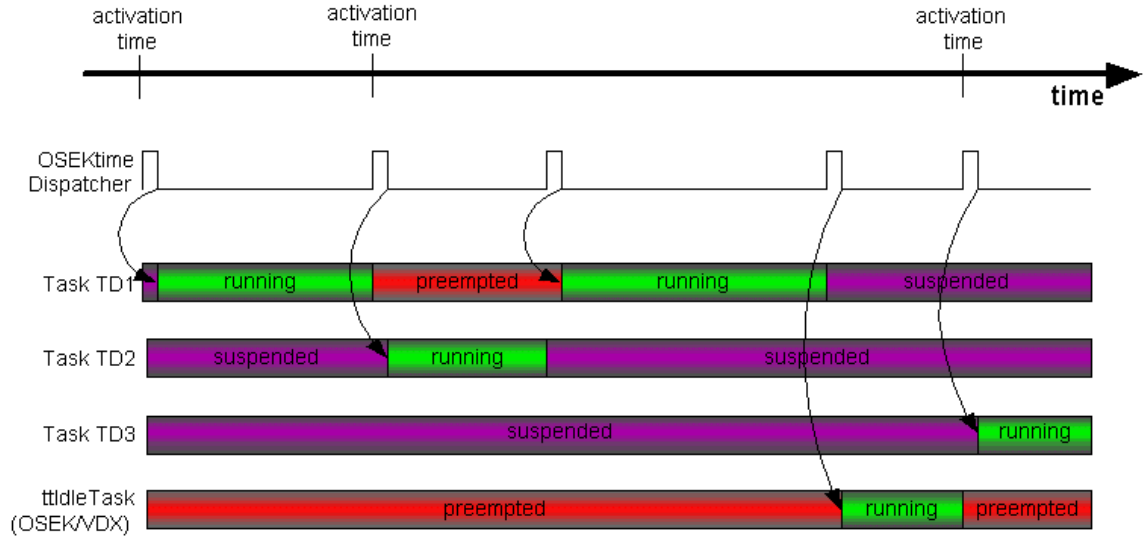

Figure 2: Operation of the Stack Based OSEKtime Dispatcher

The occurrence of interrupts is also supervised and controlled by the operating system. An interrupt source is switched on only through an entry in the dispatcher table. After an interrupt is raised, its interrupt service routine will be executed. Then the operating system switches off the interrupt source and it can only be switched on again by a new entry in the dispatcher table. In this way, the whole dispatcher round can be divided into disjoint time intervals where a particular interrupt can occur at most once within one of these intervals.

Since the duty of the dispatcher during runtime is restricted to the cyclic execution of the dispatcher table, the configuration process of the distributed system with highly synchronized job execution has to be done with care. The knowledge of the worst case execution time (WCET) [10, 11, 12, 13, 14] of every task and interrupt service routine is essential to plan the course of a dispatcher round.

2.1 OSEK/VDX Subsystem

If an application consists of a time-critical part and a more complex, non time-critical part, then OSEKtime offers an interesting solution for it: Instead of the ttIdleTask described above, an OSEK/VDX subsystem can be integrated which offers an extensive set of functionality to the user. These are described briefly in the following paragraphs.

2.1.1 Tasks

There are two types of tasks in the OSEK/VDX subsystem. These are *basic* and *extended* tasks. Both can have the states *ready*, *running,* and *suspended*. Additionally, extended tasks can assume the state *waiting* to wait for an event. In contrast to the time-driven tasks described earlier, OSEK/VDX tasks have priorities starting at 0 (low) and increasing to larger numbers. OSEK/VDX tasks support different scheduling strategies: a fully preemptive OSEK/VDX task in the *running* state can be interrupted and moved to the *ready* state by another OSEK/VDX task which reaches the *ready* state and has a higher priority. However, a non-preemptive OSEK/VDX task does not release the CPU during its execution when a higher priority task reaches the *ready* state. A task in the *suspended* state is passive and can be activated.

2.1.2 Interrupt Service Routines

The OSEK/VDX subsystem offers two categories of interrupt service routines. Category 1 ISRs do not use any OSEK/VDX functionality. After the termination of such a routine, the normal program execution is continued at the point where the corresponding interrupt occurred. In ISRs of category 2, OSEK/VDX

functions can be called which can initiate a rescheduling. Rescheduling takes place after termination of the ISR if the interrupted task is fully preemptive and there is no other pending interrupt.

2.1.3 Events

Events are used to synchronize OSEK/VDX tasks and interrupt service routines. Extended tasks can wait on one or more events while any OSEK/VDX task, ISRs of category 2, and some other mechanisms can set an event. In doing so an event driven mechanism can be built up allowing extended tasks to wait for a certain event occurrence without requiring CPU time.

2.1.4 Counters and Alarms

Counters serve for counting any kind of event and can, for example, be associated with a cyclically occurring timer interrupt. An alarm represents an action, which will be executed by reaching a certain counter value. This action can either be the activation of an OSEK/VDX task or the setting of an event.

2.1.5 Resources

During the time in which an OSEK/VDX task occupies a resource, it cannot be interrupted by another OSEK/VDX task that also requires the same resource. By this mechanism, the access to critical sections, for example memory or ports, is limited and mutual exclusion constraints are enforced.

2.2 Mixed System

If combining OSEKtime and OSEK/VDX in a so-called “mixed system” one has to take care for obvious reasons that all OSEKtime interrupts have precedence over the OSEK/VDX interrupts. Because of this, OSEKtime can delay the execution of OSEK/VDX IRQs, but all time critical activity should be processed in the OSEKtime context anyway. The same is true for OSEK/VDX IRQs and OSEKtime tasks: If OSEK/VDX IRQs have a higher priority than OSEKtime tasks, the OSEK/VDX IRQs can delay the processing of OSEKtime tasks which is a contradiction to the behavior required from the OSEKtime specification.

3 OSEKtime FTCom

For the determinism of the entire distributed system, it is required that not only the operating system and the application of each ECU, but also the communication among the different ECUs exhibits a deterministic temporal behavior. In order to achieve this determinism at communication level, time-driven communication protocols (e.g., FlexRay [15, 16]) are used especially for safety-critical applications.

In order to abstract from the peculiarities of a concrete communication protocol, the OSEK/VDX committee defined the middleware layer OSEKtime FTCom. This provides a standardized interface for the exchange of messages (i.e. parts of communication frames) among application tasks (possibly located on different ECUs).

3.1 OSEKtime FTCom Software Layers

OSEKtime FTCom itself is structured into different software layers. These layers are depicted in Figure 3 and are described in the following sections.

3.1.1 Interaction Layer

The interaction layer takes care of the representational issues of messages like byte ordering and alignment. On the sender’s side, this layer converts messages from the byte order of the sending ECU into the network byte order. Furthermore, the interaction layer packs multiple messages into a single communication frame in order to reduce communication bandwidth consumption.

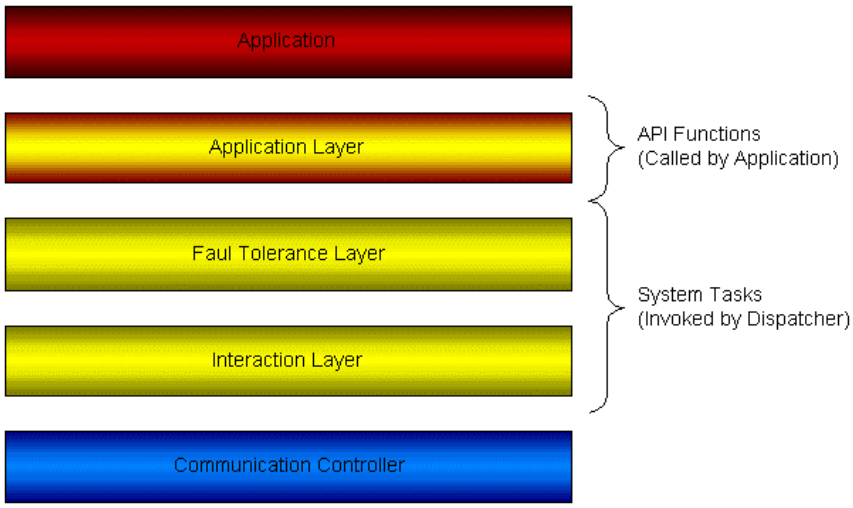

Figure 3: OSEKtime FTCom Layers

On the receiver's side, the layer extracts the multiple packed messages from a single communication frame and converts the message from network byte order into the byte order of the receiving ECU.

3.1.2 Fault Tolerance Layer

On top of the interaction layer, the fault tolerance layer manages all fault tolerance issues, namely message replication and message reduction.

On the sender's side, this layer replicates a single application message and thus produces multiple message instances. These message instances are handed over to the interaction layer for byte order conversion and packing. Afterwards, the message instances are transmitted via redundant communication paths. Hereby temporal redundancy (multiple transmissions on a single communication channel) and spatial redundancy (transmission on multiple communication channels) can be distinguished.

On the receiver's side, multiple message instances are collected from the interaction layer and reduced to obtain a single application message, which is handed on to the application layer. The number of message instances per application message as well as the reduction algorithms used depend on the fault model[1] assumed and on the number of faults[2] to be tolerated.

In addition to a fixed number of predefined reduction algorithms, OSEKtime FTCom provides the possibility to defined custom reduction algorithms as well.

3.1.3 Application Layer

The top-most layer, the so-called application layer, acts as the application interface of OSEKtime FTCom. This layer provides three API functions for message exchange: one for the transmission of a message, one for the reception of a message, and one for tagging a message as invalid[3].

3.2 System Tasks vs. API Functions

The functionality of the interaction layer and the fault tolerance layer is provided by so-called system tasks. These system tasks are invoked on a behind-the-scenes basis by the OSEKtime dispatcher.

The functionality of the application layer, however, is implemented in the form of API functions that are activated by the application program by means of function calls.

---

[1] In case of consistent faults in value domain, majority voting would be an appropriate reduction algorithm.

[2] In order to tolerate one consistent fault in the value domain, 3 message instances are required.

[3] In time-driven communication systems it is generally not possible to simply omit a message transmission. Thus some possibility must be provided to mark a message in a way that the receiver can classify the message as invalid.

3.3 Time Management

In addition to the previously presented services for message exchange, OSEKtime FTCom provides services dealing with (global) time. Among these services are API functions for access to the global time provided by the communication protocol and for the synchronization of the OSEKtime dispatcher with this global time. This kind of synchronization makes it possible to state an end-to-end exchange latency for each message transmitted from one ECU to another and is thus an important prerequisite for the determinism of the overall system.

4 Standard Software Components

In addition to the services provided by the OSEKtime operating system and OSEKtime FTCom, standard software components are included in TimeCore. The configuration of these standard software components is done via the tools described in the following sections.

The first release of TimeCore (V1.0.0) provides a generic communication driver for FlexRay, several debugging aids like stack checking and ORTI [8, 9] support as well as timing facilities like real-time clocks and task tracing. Future versions of TimeCore will contain standardized software components for transport layers [1], network management components [2] for various communication protocols, components for diagnostic services [7], driver packages for on-board modules like analogue digital converters [5, 6], and boot loading support.

5 Development Process - Tooling

To ensure the consistency between application code, configuration, and standard software components, the whole development process should be accompanied and assisted by a tool chain. Commonly this development process is based on the so-called V-model, which is illustrated in Figure 4.

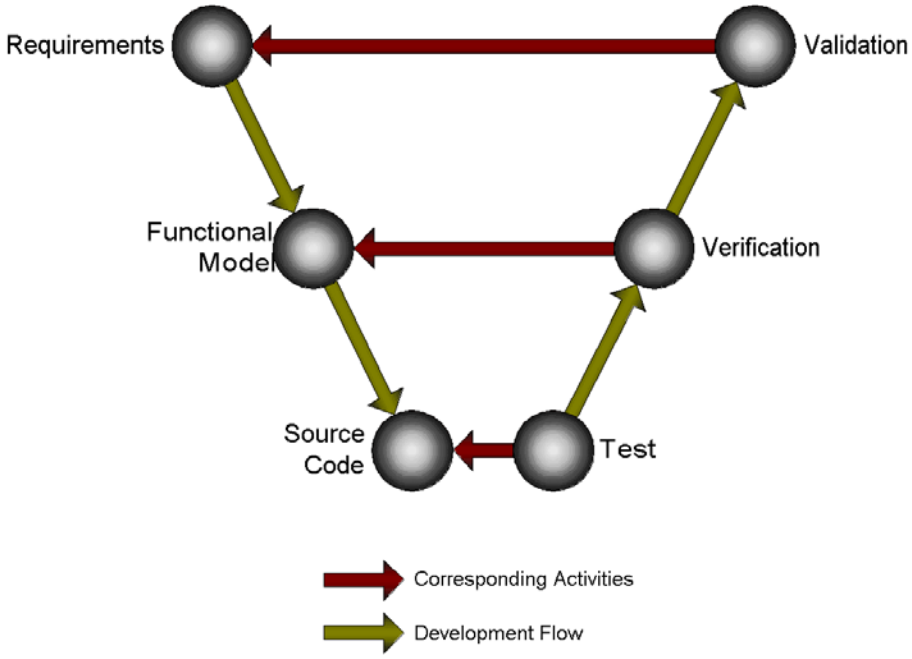

Figure 4: The V-Model

Based on the system requirements a functional model of the system is constructed. It is common practice to create this model in a simulation language like MATLAB/Simulink or ASCET SD. Thus, it is possible to perform simulations with this model in order to detect deficiencies of the modeled system at an early development stage. Using code generators the *application specific* parts of the model are then translated into target code. The right part of the V-model deals with testing of the (generated) code, verification of the system's functionality against the functional model, and validation of the system against the initial system requirements.

It should be noted that, as far as communication is concerned, the step from the functional model to the (generated) code is not supported by conventional tool chains. This fact is a major drawback since, especially in tightly synchronized time-driven systems, all actions (application and communication) should be planned and developed in a holistic manner. Furthermore, the continuity of the tool chain is an essential property ensuring that the simulation results obtained at intermediate abstraction and modeling levels match the behavior of the resulting real system on the respective target hardware.

Therefore, the TimeCore tool chain introduces a new sub-model, the so-called A-model, between the functional model and the (generated) code. This A-model is illustrated in Figure 5.

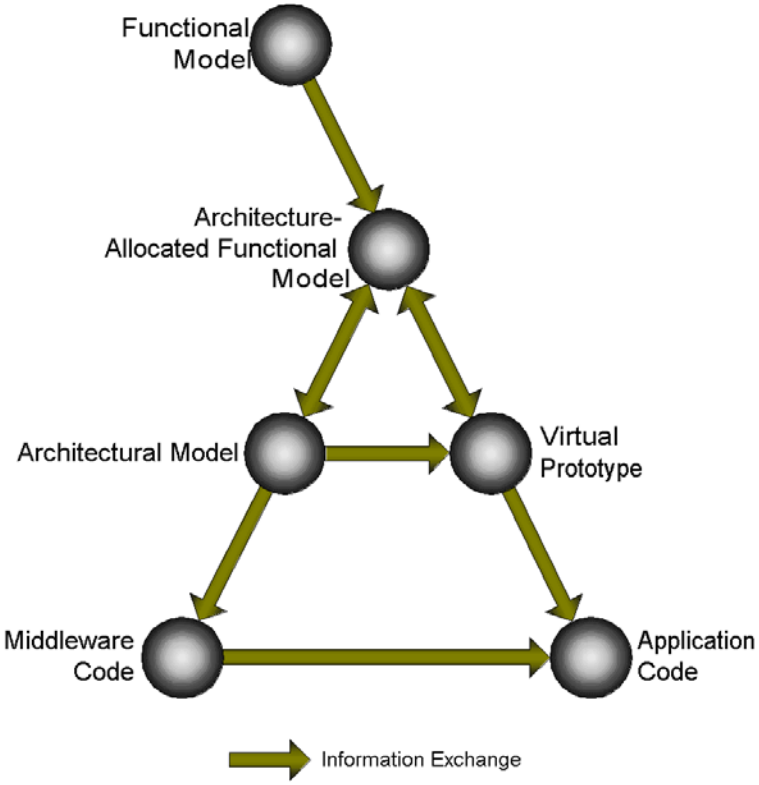

Figure 5: The A-Model

5.1 The Architecture-Allocated Functional Model

Starting from the plain functional model, an architecture-allocated functional model is created. This model takes the architectural entities (e.g., tasks, ECUs, communication controller) of the distributed system into account. The plain functional model is partitioned into multiple model parts, which are then assigned to logical execution units, namely tasks. The tasks themselves are supplied with parameters like period, temporal offset within this period, deadline, and worst case execution time (WCET).

The deadline describes the latest possible point in time by which the execution of a task has to be completed. The worst case execution time depends on the task itself (i.e. on its program code) and on the preemptions caused by other tasks on the one hand and by interrupts (and the respective ISRs) on the other hand. While the preemption caused by other tasks can be derived from the task scheduler (i.e. from the dispatching table), one can only make assumptions about the preemptions caused by the interrupts.

Once supplied with valid parameters, the tasks are assigned to physical execution units, namely ECUs. Depending on this assignment, message relations can be derived. These message relations are resolved either as local (inter-task) communication or as global (inter-ECU) communication. Similarly to tasks, attributes like period, offset and data type are supplied to messages.

Using TimeCore's SIMSYSTEM block set for Simulink, which provides special blocks for ECUs and tasks as well as connectors for the modeling of the message relations, the previously described partitioning can be performed. The resulting partitioned architecture-allocated functional model can then be used to simulate the effect of task execution times on the overall system.

5.2 The Architectural Model

The architectural model solely contains architectural information. This model is derived from the architecture allocated functional model by removing all functional aspects from the model.

The architectural model can be split into a software model containing tasks and messages, and a hardware model containing ECUs, communication controllers, and communication media. The DESIGNER tool manages the architectural model and provides a graphical user interface for manipulating the model (see Figure 6).

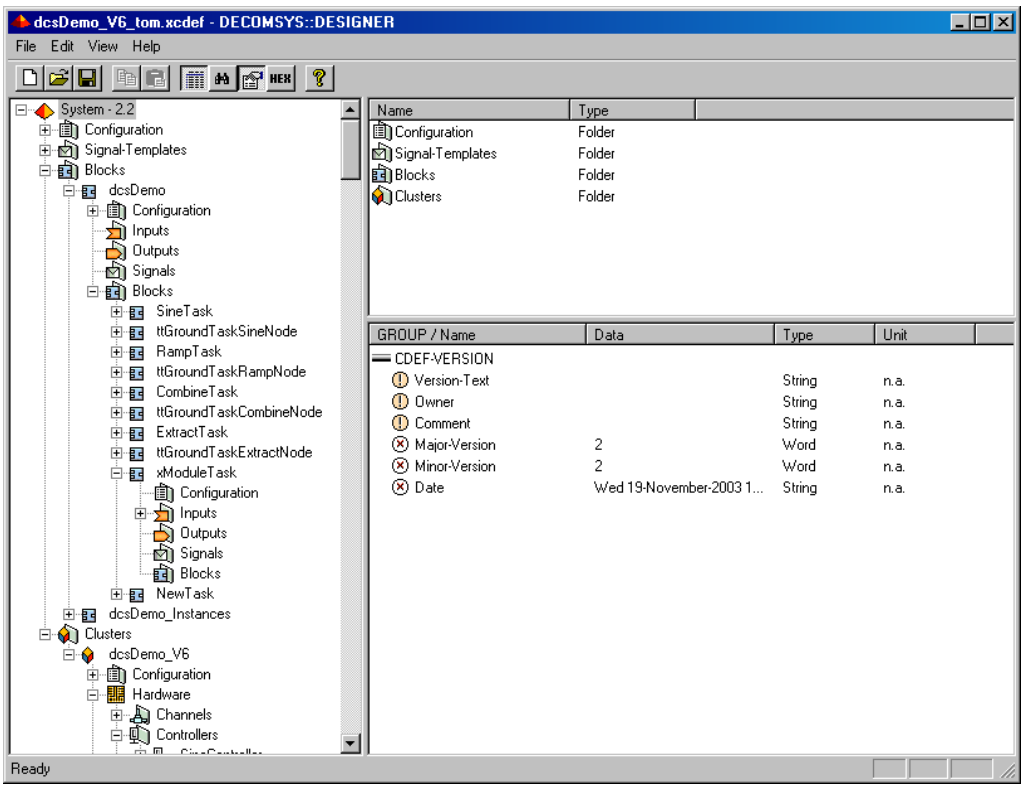

Figure 6: The DESIGNER GUI

Based on the information contained in this model, a communication schedule and appropriate configuration data for the communication controllers can be generated automatically.

The task schedule currently has to be designed by hand (using the DESIGNER GUI). In future extensions of the TimeCore tool chain however, this task schedule will be generated automatically similarly to the communication schedule.

5.3 The Virtual Prototype

After the task and communication schedules have been created, this scheduling information is back annotated into the architecture-allocated functional model. In this step, the SIMSYSTEM connectors are replaced by SIMCOM connectors with the ability to simulate communication according to the communication schedule generate in the architectural model. The resulting model is called the virtual prototype.

Any simulation performed with this virtual prototype uses a special simulation engine that takes temporal (e.g. communication latencies) and functional (e.g. quantization errors) aspects of the communication among the tasks into account. Therefore, the virtual prototype exhibits the same functional and temporal behavior as the real system. Using the virtual prototype, errors made during system design can be identified very early in the development process prior to building a real hardware prototype of the system.

5.4 Middleware Code Generation

Using the data contained in the architectural model, the GENERATOR produces OSEKtime FTCom compliant middleware code. Hereby, for each message, dedicated optimized code is generated taking into account the peculiarities of each specific message. The major benefit of this approach, when compared to a general communication stack, is the fact that parameters like signal length and position of the signal within the communication frame are constants in the generated code and allow for strength optimizations during compilation. Thus, this approach drastically reduces the execution time of the rather time consuming actions that have to be performed by the OSEKtime FTCom compliant middleware layer.

5.5 Application Code Generation

Using SIMTARGET, a target module for Real-Time Workshop, the virtual prototype can be used for the generation of application code. Hereby, for each SIMCOM connector, a call to the respective OSEKtime FTCom API function provided by the generated middleware code is produced during application code generation.

5.6 OIL Export and ECU Configuration

Based on the information contained in the architectural model, the OILEXPORTER generates an OIL file for every ECU in the distributed system. The OSEKtime configuration in these files is consistent between the various ECUs and time driven tasks so that they can meet their time-based boundary conditions. However, there are still operating system resources in a single ECU, which can or have to be configured with the ProOSEK Configurator (Figure 7) independently from the rest of the time-driven network:

- The starting points of time-driven tasks, which do not have a network, related context or the enabling of IRQs may be adjusted to the conditions of an ECU. In this way, an ECU specific fine adjustment can be achieved.
- The OSEK/VDX configuration is not part of the global configuration and therefore not contained in the OIL files generated by the OILEXPORTER. An OSEK/VDX subsystem has to be completely configured with the ProOSEK Configurator.

After finishing the configuration, procedure the ProOSEK Configurator finally generates an operating system kernel which is optimized according to the underlying OIL information.

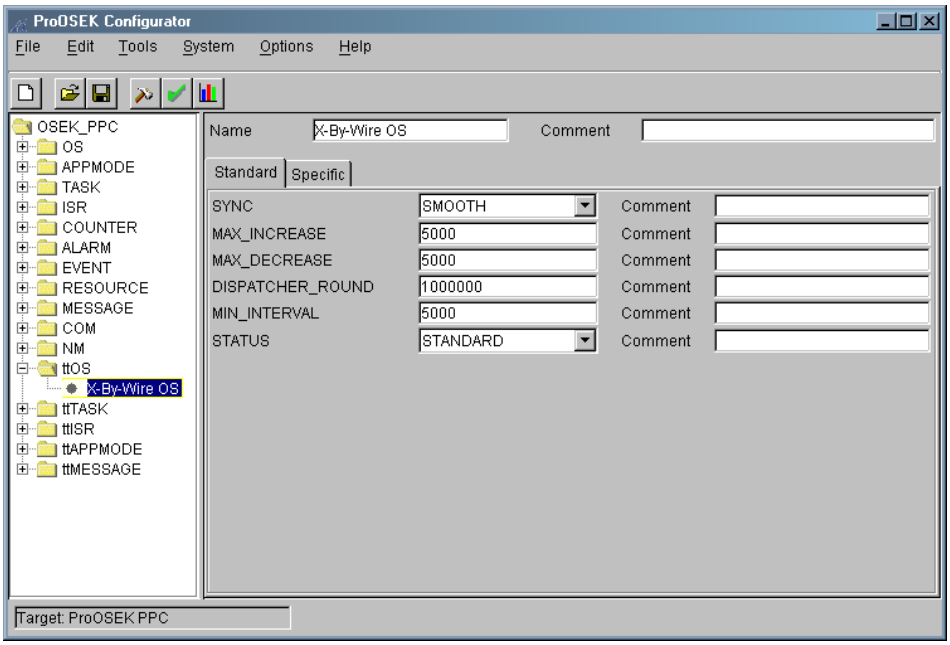

Figure 7: The ProOSEK Configurator GUI

By the planned embedding of the TimeCore into the tresos environment, it will be possible to extend the configuration procedure to a wider scope. With tresos, components like I/O drivers, cryptographic libraries or a bootloader can be configured together with the operating system. The integrated approach of tresos makes sure that all inter-module dependencies are resolved. For example, OSEK/VDX tasks needed by a driver will be configured automatically in the proper way with respect to the whole system.

5.7 Compile and Link

In a last development step all generated code parts (operating system code, application code, and middleware code) are compiled separately and linked together, resulting in the executable application program, which can be downloaded onto a single ECU.

For a summary, Figure 8 illustrates the overall TimeCore development process.

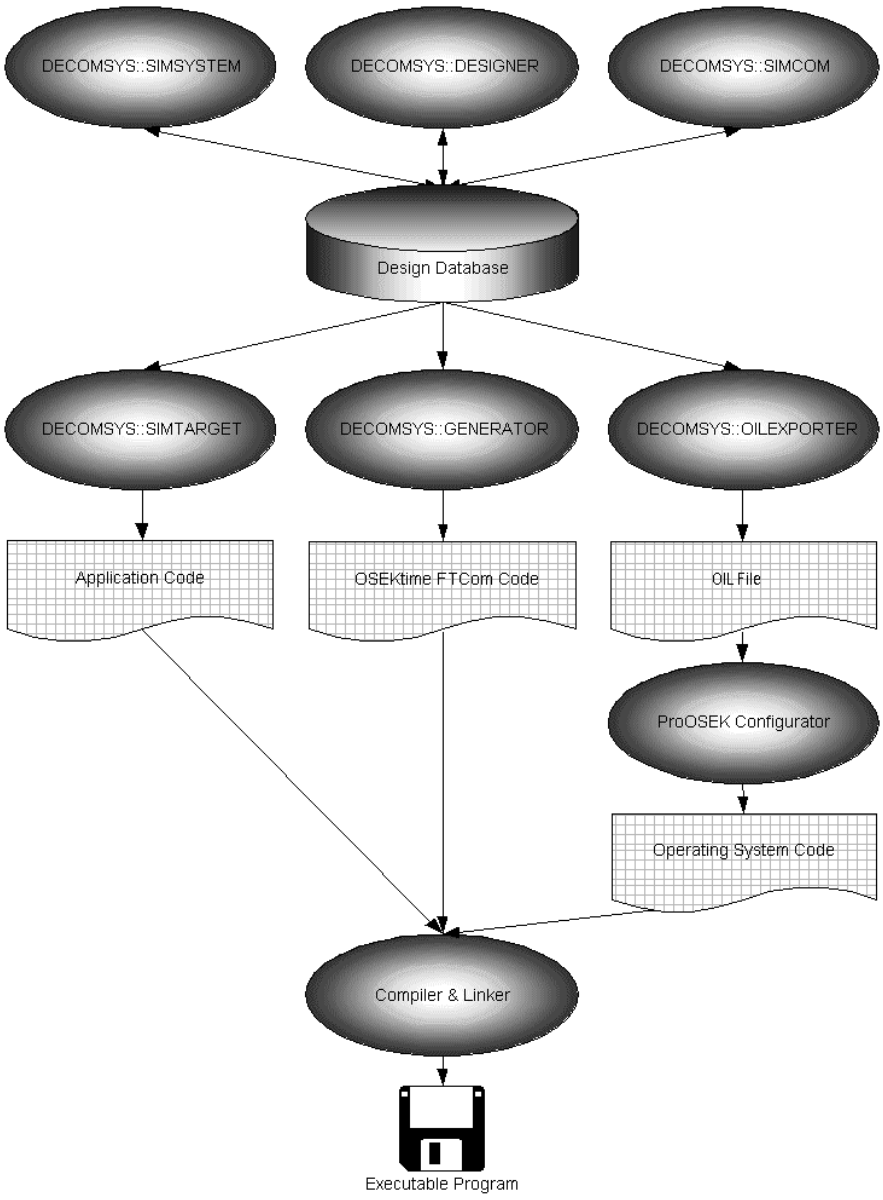

Figure 8: TimeCore Development Process

6 Making Ends Meet

As stated in the development process section, several assumptions about parts of the system have to be made during design time. At system run-time, however it is essential to ensure that the assumptions made during system design time really hold during operation.

In the following it is illustrated how design assumptions can be enforced by services provided by the OSEKtime operating system.

6.1 Worst Case Execution Times

All task scheduling and all message exchange latency calculations are based on the worst case execution time of the application and the system tasks. If, however, one of the tasks exceeds the assumed worst case execution time, the behavior of the system deviates from the modeled and simulated behavior. Furthermore, once this happens, the overall system behavior is no longer clearly defined.

Using the deadline monitoring service provided by the OSEKtime operating system and by placing the task deadlines close to the tasks latest possible termination time (i.e. task start time plus worst case execution time), the assumptions made about the task WCET at design time can be enforced at run time. If in such a setup a task exceeds its assumed worst case execution time, the operating system can initiate error handling strategies or guide the whole system into a safe state, thus maintaining a defined overall system behavior.

6.2 Interrupt Occurrence

Apart from assumptions about the WCETs of tasks, assumptions about the occurrence of interrupts (and about possible preemptions of tasks resulting from these interrupt occurrences) have been made. If due to a malfunction of the interrupt line, one interrupt occurs more often than initially assumed, the system behavior again deviates from the modeled behavior since the increased occurrences of interrupts causes more task preemptions, which in turn causes increased real WCETs.

With the interrupt supervision service of OSEKtime, the occurrence of interrupts can be limited to the number of occurrences assumed during system design time. Thus whenever this feature is used appropriately, a deviation of the system behavior during run-time caused by interrupt showers is prevented.

7 Conclusion

TimeCore provides a combination of the time-driven operating system OSEKtime OS and the fault-tolerant middleware layer OSEKtime FTCom. Due to the time-driven operation, tight temporal synchronization can be achieved in the distributed system.

An integrated development process using a model-based approach guides the application developer from system design to optimized application code. At different levels of abstraction the intermediate system models facilitate system verification at an early stage in the development process without the need to build up a hardware prototype of the system.

Run-time services of the operating system like deadline monitoring and interrupt supervision ensure that assumptions made during system design hold during the system's run-time.

The predictability resulting from the tight temporal synchronization together with the inherent fault tolerance provided by OSEKtime FTCom and the increased overall test coverage through the use of standard software components makes TimeCore well suited for deployment in safety-critical automotive applications.

References

[1] ISO (International Organization for Standardization), *Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 2: Network Layer Services*, ISO/DIS 15765-2.2, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, June 2003

[2] C. Hoffmann, J. Minuth, J. Krammer, J. Graf, K. J. Neumann, F. Kaag, A. Maisch, W. Roche, O. Quelenis, E. Farges, P. Aberl, D. John, L. Mathieu, M. Schütze, D. Gronemann, J. Spohr, *OSEK/VDX – Network Management - Concept and Application Programming Interface, Version 2.5.2*, January 16th 2003; Available at http://www.osek-vdx.org/mirror/nm252.pdf

[3] V. Barthelmann, A Schedl, E. Dilger, T. Führer, B. Hedentz, J. Ruh, M. Kühlewein, E. Fuchs, Y. Domaratsky, A. Krüger, P.Pelcat M. Glück, S. Poledna, T. Ringler, B. Nash, and T. Curtis, *OSEK/VDX - Time-Triggered Operating System, Version 1.0*, July 24th 2001; Available at http://www.osek-vdx.org/mirror/ttos10.pdf

[4] A. Schedl, E. Dilger, T. Führer, B. Hedenetz, J. Ruh, M. Kühlewein, E. Fuchs, T. M. Galla, Y. Domaratsky, A. Krüger, P. Pelcat, M. Taï-Leung, M. Glück, S. Poledna, T. Ringler, B. Nash, and T. Curtis, *OSEK/VDX - Fault-Tolerant Communication, Version 1.0*, July 24th 2001; Avalable at http://www.osek-vdx.org/mirror/ftcom10.pdf

[5] Duda, Kraneis, J. Spohr, and C. Stellwag, *HIS - API IO Driver, Version 2.1.1*, September 2nd 2003; Available at http://www.automotive-his.de/download/API_IODriver_2_1_1.pdf

[6] Duda, Kraneis, and C. Stellwag, *HIS - API IO Library, Version 2.0.1*, September 2nd 2003; Available at http://www.automotive-his.de/download/API_IOLibrary_2.0.1.pdf

[7] ISO (International Organization for Standardization), *Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 3: Implementation of Diagnostic Services*, ISO/DIS 15765-3.2, 1, rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland, October 9th 2003

[8] Barthelmann, Büchner, Dienstbeck, Elies, Fathi, Hoogenboom, Janz, Kriesten, Nieser, Nishikawa, Schimpf, Stehle, Ulcakar, Vetterli, Wertenauer, Winters, *OSEK/VDX – OSEK Run Time Interface (ORTI); Part A: Language Specification, Version 2.1.1*, March 4th 2002; Available at http://www.osek-vdx.org/mirror/ORTI-A-211.pdf

[9] Barthelmann, Büchner, Dienstbeck, Elies, Fathi, Hoogenboom, Janz, Kriesten, Morgan, Nieser, Nishikawa, Schimpf, Stehle, Ulcakar, Vetterli, Wertenauer, Winters, *OSEK/VDX – OSEK Run Time Interface (ORTI); Part B: OSEK Objects and Attributes, Version 2.1*, April 17th 2002; Available at http://www.osek-vdx.org/mirror/ORTI-B-21.pdf

[10] P. Puschner, *Timing Analysis for Real-Time Programs*, PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, 1993

[11] Raimund Kirner, *Extending Optimising Compilation to Worst-Case Execution Time Analysis*, PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, May 2003

[12] C. Y. Park and A. C. Shaw, *Experiments with a Program Timing Tool based on a Source-Level Timing Schema*, Computer, 24(5):48–57, May 1991

[13] P. Puschner and A. V. Schedl, *Computing Maximum Task Execution Times – A Graph-Based Approach*, The Journal of Real-Time Systems, 13:67–91, 1997

[14] A. Vrchoticky, *The Basis for Static Execution Time Prediction*, PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3/3/182-1, 1040 Vienna, Austria, April 1994

[15] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluher, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann, *FlexRay - The Communication System for Advanced Automotive Control Systems*, SAE 2001 World Congress, Detroit, MI, USA, March 2001

[16] T. Führer, F. Hartwich, R. Hugel, H. Weiler, *FlexRay-The Communication System for Future Control Systems in Vehicles*, SAE 2003 World Congress & Exhibition, Detroit, MI, USA, March 2003