MoDe: A Method for System-Level Architecture Evaluation

Jan Romberg
Systems & Software Engineering
TU München
85748 Garching, Germany
romberg@in.tum.de

Oscar Slotosch, Gabor Hahn
Validas AG
Lichtenbergstr. 8
85748 Garching, Germany
{slotosch, hahn}@validas.de

Abstract

System-level design methodologies for embedded HW/SW systems face several challenges: In order to be susceptible to systematic formal analysis based on state-space exploration, a modelling notation with a simple formal semantics is desired. Architecture-level engineering practice demands notations which concentrate on certain aspects of system functionality, while other aspects (such as communication and scheduling) are implicitly encoded in the language semantics, and realized using HW/SW components such as operating systems and protocol stacks. We describe a system-level design methodology targeted for automotive control applications. Models in a simple graphical component-based input language are compiled into complex system models incorporating abstractions for hardware, operating systems, and inter-processor communication. System models are based on the synchronous AutoFocus notation and are used as a basis for formal analysis such as systematic worst-case response time analysis. The paper describes a reference architecture for implementation, the MoDe design notation, and the translation to system models along with an outlook giving a perspective for analysis.

1 Introduction

With as many as 80 electronic control units running distributed applications in modern premium-class vehicles, software engineering for automotive control is foraying into the complexities formerly reserved to the business information system and telecommunication domains. The predominant focus in automotive development on subsystem-level ECU design is shifting towards an application- and service-oriented perspective, where the corresponding functionality is distributed across several subsystems. However, with non-functional properties such as timing constraints, per-unit costs, power consumption, and space as important requirements, finding an optimized system-level design is much harder than optimizing a single subsystem.

The goal of the MoDe (*Mo*del Based *De*ployment) approach is to give early guidance for design decisions using architectural-level models of the system. In the following, we will restrict ourselves to those architecture-level decisions, such as choice of the right processor to maximize utilization, deployment of software components, or examination of bus load, that require a performance model of the overall system.

The MoDe approach is based on a design notation with a simple and extensively supported formal semantics, AutoFocus. AutoFocus is used on two levels: Firstly, the AutoFocus formalism is extended for modeling both the functional aspects of the application and the platform aspects of the HW architecture. Secondly, AutoFocus is used in order to represent an overall model of the system used for detailed analysis. The use of system models with a clear formal semantics opens MoDe to a wide variety of formal analysis options based on state-space exploration.

1.1 Deployment

The term deployment in our context denotes the distribution of system functionality onto physical components. *Model-based* deployment uses abstractions of the deployed system to perform analysis such as validation and simulation. We have identified two possibilities of employing AutoFocus models for model-based deployment:

1. Use a given AutoFocus model directly as the behavioral specification of the system. In this case, the main difficulty for deployment is ensuring behavioral consistency between the model and its implementation as a combination of hardware, operating system, and software.

2. Use additional (and possibly nondeterministic) abstractions for communication and scheduling in an overall AutoFOCUS model of the system. The system model is preferably compiled from separate notations for functionality and HW architecture, and standard models for scheduling and communication are automatically inserted.

For MoDe, we choose option 2., using a separate input language and AutoFOCUS system models. Let us briefly discuss the reasons for this choice: When opting for 1. and directly implementing models in a synchronous language such as AutoFOCUS, the architecture must guarantee some assumptions made by the formalism [2]. These assumption tend to be rather strong for globally synchronous models. For distributed communication, strongly deterministic architectures such as Time-Triggered Architectures [13] are definitely be a good candidate for the first approach, and are already foraying into safety-critical automotive applications.

On the other hand, the majority of applications to this date are based on event-triggered, weakly deterministic architectures such as the Controller Area Network (CAN) protocol for inter-processor communication, queued events and shared variables for intertask communication, and preemptive scheduling with intertask synchronization primitives.

When a design is in its architectural stage, much of the detailed performance characteristics of the implementation are not known. If one wants to analyze a design with formal models, one possible abstraction is to drop timing altogether, and an asynchronous system model results. When unwinding the possible executions of asynchronous models, e.g. for checking safety properties, a large number of interleavings of single process executions has to be considered. As a consequence, methods based on state-space exploration quickly become impractical, despite some encouraging progress in the area [9].

In MoDe, functional models of the system are enriched with actual platform and implementation information; the system model which is subjected to analysis incorporates these informations to improve analyzability. Consequently, system models are tied to properties of the platform: the possible executions of the model closely reflect possible executions in the implementation in order to ensure the validity of model-based analysis. By offering automated support for compiling platform abstractions into the system model, MoDe still retains the flexibility with respect to implementation choices that motivate the abstraction from timing properties described above.

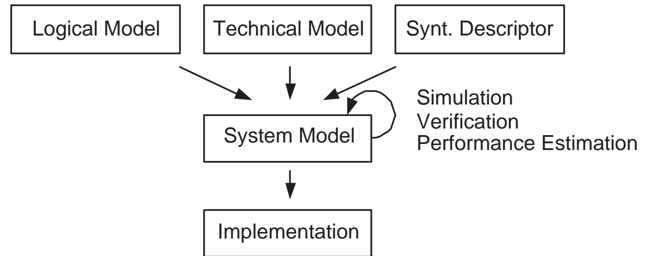

Figure 1. Design Flow

In the remainder of this paper, we will give an overview of our methodology and the design flow. The application domain is characterized in terms of the supported reference architecture. We will then give a definition of the visual notation used to describe HW/SW architectures. An application example is presented and the currently available analysis techniques are described. Finally, we present an outlook discussing future extensions of our method.

2 Overview

2.1 Design Flow

Figure 1 shows the design flow for the MoDe approach. The designer starts out with a functional view on the application, the *logical model*, a view on the technical architecture of the system, the *technical model*, and an additional *synthesis descriptor*. The MoDe System Model Generator reads the input representations and generates an AutoFOCUS *system model* used for simulation, verification, and performance analysis.

Logical model

The logical model captures the software architecture in terms of *software components*, component *interfaces*, and *communication dependencies* between components. Each component is specified in terms of its functional and temporal behavior, the latter of which depending on properties of the technical model described below. The logical model abstracts from the actual deployment, i.e. communication protocols and scheduling are not part of the logical specification.

Technical model

The characteristics of the platform are specified within the *technical model*. The technical model captures the structure of the underlying hardware, the characteristics of the different nodes of the platform, the characteristics of the links between nodes, and parameters

of the operating system such as supported scheduling policies and I/O drivers. In a way, the technical model fills the behavioral and structural "blanks" left unspecified by the logical model. Note that the technical model is intended to be an independent and self-contained view of the platform that is reusable across several designs. The concepts of the technical model are supported by *component libraries* which define the components required to generate the system model.

Synthesis descriptor

As the logical and technical models alone are not expressive enough to perform an automated synthesis of system models, some additional mapping information has to be provided by the developer in the form of a *synthesis descriptor*. It is required to ensure an unambiguous translation from logical and technical models to system models. The synthesis descriptor captures information that depends both on the logical and technical models; this typically includes the mapping from logical components to technical nodes, and additional information required for scheduling such as static schedules or priorities, depending on the scheduling policy.

The separation into three distinct descriptions ensures, to the greatest possible extent, a separation of concerns between application and platform aspects, and enables the developer to separately focus on desired functionality and technical realization. We will refer to the combination of logical model, technical model, and synthesis descriptor as the *deployment model* in the remainder of this paper.

System Model

The AutoFOCUS System Model Generator compiles the logical and technical models into an AutoFOCUS system model. The synthesis step uses the mapping from logical to technical components in the synthesis descriptor to synthesize the system model as a complete representation of the system behavior relevant to analysis. All analysis is based on the system model; if several different deployments are evaluated, a corresponding number of analysis cycles are required.

2.2 Analysis and Iterative Development

The system model is used to perform analysis of the system. There is a wide variety of analysis options supported by AutoFOCUS which are all applicable to MoDe system models: a visual simulation interface using precompiled test vectors or user interaction as

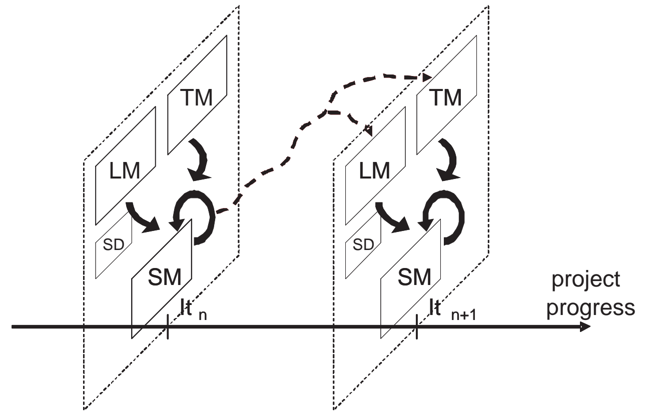

Figure 2. Iterative Development

input [10], formal verification using either the symbolic BDD-based model checker SMV or the bounded-depth solver SATO [18], and automated test case generation using a Constraint Logic Programming (CLP) environment [15].

For architecture evaluation, the analysis capability which is currently of main interest is system-level response time analysis. The current framework supports this kind of analysis both through the AutoFOCUS simulation interface, and by systematic quantitative analysis sketched in the "Results" section. The results of the analysis serve as a basis for modifications of the deployment model in the next iteration (Fig. 2).

3 Reference Architecture

3.1 Inter-processor communication

The reference architecture for MoDe consists of a number of single-processor boards (*nodes*) communicating over a number of communication *links*. In MoDe system models, properties of the processor are modeled by service timing functions (see below). For modeling inter-processor communication, MoDe system models incorporate abstractions for *drivers* and *synchronization*:

Drivers. In a layered communication architecture, the logical model typically specifies the application layer. For distributed communication, models of *drivers* are required to specify the behavior of layers between the application layer and the layer modeling the actual link in the system model. Note that this layer is not necessarily the the physical layer. The choice depends on the level of detail required for the analysis.

Transport medium and synchronization. This includes synchronization between multiple nodes (e.g. for buses), potential message loss, etc. For instance, a synchronization component for a bus with priority-based arbitration yields the bus to the requesting driver with the highest priority for a given arbitration cycle.

Models for drivers and synchronization are imported through a library mechanism. The MoDe communication library currently supports two types of links:

RS232 serial communication. RS232 is a simple unidirectional serial protocol. In the MoDe library, RS232 is modeled with comparatively simple driver components; a separate synchronization component is not necessary for point-to-point links if the physical communication is assumed to be reliable.

CAN bus communication. CAN is a Local Area Network protocol with priority-based arbitration geared at real-time and automotive applications. The MoDe library includes both a driver model representing the sub-application-layer functionality, and a synchronization component modeling arbitration.

3.2 Inter-Task Communication

Nodes are assumed to run a real-time operating system with support for multitasking, mutual exclusion protection, and I/O drivers. The mapping of software components to nodes is assumed to be fixed during run-time, i.e. migration of software components is not considered.

The MoDe approach imposes some constraints on the OS services used for implementation:

- The OS primitives used are *threads*, *message queues*, and *shared resources*.
- All computations are done by threads. Threads are assumed to share memory with other threads, and inter-thread communication is performed either via message queues, or by shared-variable communication through protected memory resources. A thread may keep some of its internal state between consecutive activations.
- Queues realize FIFO queues of unbounded size. Messages are never lost.
- Access to protected resources is restricted to one thread at a time (mutex protected). A running thread may acquire a free resource or release a locked resource at any time.

The abstractions currently used for scheduling and inter-task communication correspond to the OSEK OS 2.2 standard [14]. OSEK OS is an open operating system standard for statically defined applications with small system resources, and was developed by a consortium of European automobile manufacturers.

Scheduling

OSEK defines two task models: *basic* and *extended* tasks. Basic tasks can be either in *suspended*, *ready*, or *running* state, while extended tasks have an additional fourth state, *waiting*, similar to the ready state, in which the stack context is saved. MoDe currently supports two options for scheduling:

- **Static (offline) Scheduling, no waiting states**. Threads are activated according to a fixed cycle. Static scheduling can be enforced in OSEK's fixed-priority framework by ensuring a deterministic cyclic sequence of task activations.
- **Fixed priority scheduling, no preemption, basic task model**. Threads are dynamically scheduled according to their priorities. Scheduling is restricted to the OSEK basic task model and non-preemptive scheduling.

3.3 The Mine Pump Example

For the following explanations, we will use an example mine pump system whose specification was taken from [11]. Though not from the automotive domain, the example exhibits some typical properties of automotive control applications, namely *real-time requirements*, *safety requirements*, *distribution*, and *resource constraints*.

The mine pump is used to pump water out of the bottom of a mine. It is equipped with water level (high, low) and gas sensors. A pump controller switches the pump on when the water reaches the high water level and turns it off when the water is below the low water level. As an additional operational requirement, the mine pump may only operate if the methane level is below a critical threshold. If a methane alarm occurs, the operator has to be notified by the system. If the methane sensor fails, the mine pump has to be transferred to a fail-safe state. This leads to the following requirements:

1. A high methane value reading by `MethaneSensor` shall cause an emergency shut down of the mine pump within 30 milliseconds.

2. A methane sensor failure shall lead to a shutdown of the pump within 65 milliseconds.

In the following section, we will illustrate how the MoDe approach allows flexible modeling of the mine pump system and how different architectural alternatives can be evaluated using MoDe.

4 Design Notation

This section presents the models used as the input for the MoDe system model generator.

4.1 Logical model

The visual notation for logical models is loosely based on the AutoFOCUS notation. In AutoFOCUS, systems are specified as hierarchical component networks, where components communicate via typed and directed channels. Similar to UML-RT and some Architecture Description Languages (ADLs), components networks are specified in System Structure Diagrams (*SSDs*). Rectangles represent components, arrows between components represent channels, and interface of a component to a channel is called port. Outgoing ports may be connected to several channels, while incoming ports are connected to one channel. Components may be defined by other SSDs, and the leaf components in the hierarchy are defined by a state machine-like formalism. All visual elements may be extended by UML-like stereotypes of the form «ID», and *tags* of the form {KEY=VALUE}. AutoFOCUS models may be extended by datatype and function definitions in a functional language. The corresponding text documents are called Data Type Definitions (*DTDs*).

Figure 3 shows the SSD for the logical model for the mine pump example. Ignoring the stereotypes for now, the system is decomposed into the following logical components:

- A pump controller (`Controller`)
- An operator panel (`OperatorAlarm`)
- Two components handling high and low water events, respectively (`WaterLowComponent`), `WaterHighComponent`)
- One component monitoring the methane level (`MethaneSensorComponent`)

A component may be refined by a network of sub-components defined in another SSD. In the mine pump example, the `Controller` component is refined by a sub-SSD (Fig. 4). The interface ports of `Controller` appear as external ports; there are three components refining `Controller`'s behavior:

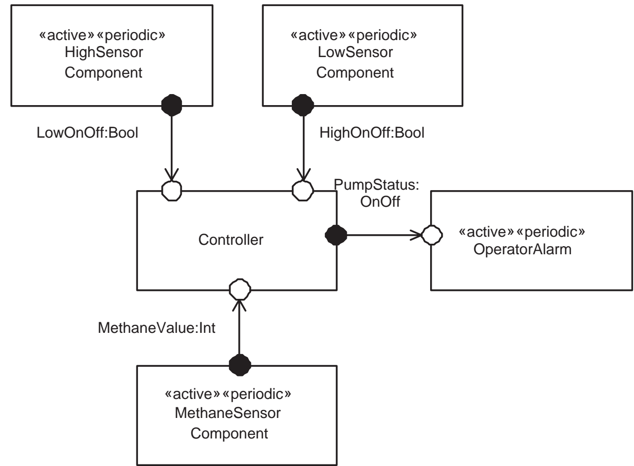

Figure 3. SSD for Logical Model

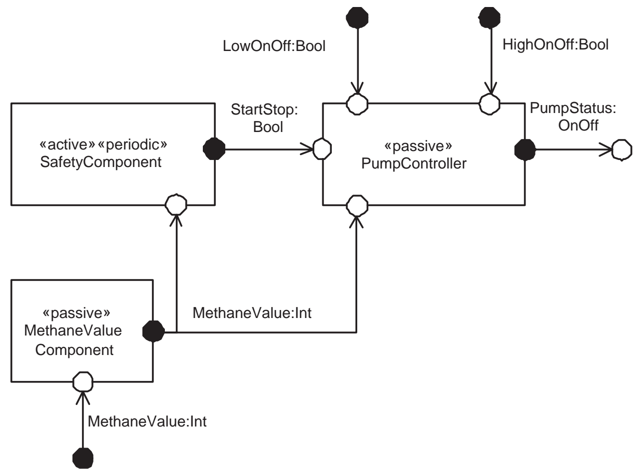

Figure 4. SSD for `Controller`

- A safety component acting as a watchdog (`SafetyComponent`)
- A component managing the state of the pump (`PumpController`)
- A component holding the current methane level, and putting timestamps on incoming new values (`MethaneValueComponent`)

Leaf components in AutoFOCUS component hierarchies are defined by State Transition Diagrams (*STDs*), which are basically an FSM dialect extended by local variables and message send/receive statements. Each

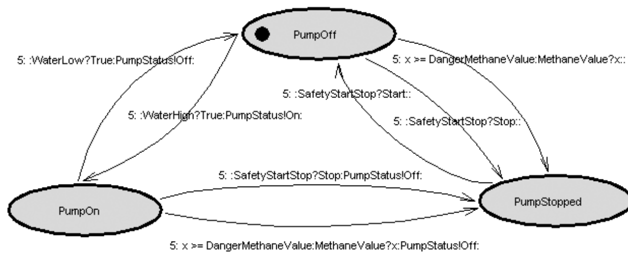

Figure 5. Simplified STD for `PumpController`

STD has a number of *locations* and *transitions* between locations. Transitions are labeled; the syntax of transition labels is PRE:IN:OUT:POST, where PRE is the *precondition*, IN is an *input statement*, OUT is the *output statement*, and POST is the *postcondition* (action). The input statement is of the form IN_PORT_ID ? EXPR, where EXPR is a functional language expression over (free) temporary variables and (bound) local variables. An empty EXPR tests for absence of a message. The guard is a boolean expression over local and temporary variables. The transition is chosen (and the temporary variables are bound to the corresponding values in the input) if the input matches the input expression, and the guard evaluates to true. An output statement of the form OUT_PORT_ID ! EXPR sends the message EXPR to an output port, and postconditions perform assignments to local variables.

Figure 5 shows a simplified STD for the `PumpController` without timing annotations. The pump controller's STD has three locations: the pump is turned off (initial locations, marked by a dot), the pump is turned on, and the pump has been stopped by the safety component. The controller switches between the `PumpOn` and `PumpOff` locations whenever the high and low water levels are reached, respectively. If either `Stop` message on the `StartStop` channel or a high methane value is encountered, the controller switches to the `PumpStopped` location. It is switched back to `PumpOff` when a `Start` message is encountered.

In logical models, as opposed to standard AutoFOCUS models, leaf components are further differentiated along their mode of activation as *active* or *passive* components.

In the mine pump example (Fig. 3), the `PumpController` and the `MethaneValueComponent` are stereotyped as passive, while all other components, such as the safety component or the sensor monitoring components, are active.

Active components

Active components are triggered by external events such as interrupts or periodically elapsing timers. Each active component corresponds to a lightweight task (thread) on the implementation level. The additional stereotype «`periodic`» defines that the activation is periodic, i.e. time-triggered, while the stereotype «`sporadic`» indicates that the activation of a component is triggered by incoming messages. Note that in combination with a static (offline) scheduling policy, the «`periodic`» and «`sporadic`» stereotypes are ignored, and a cyclic activation results.

The behavior of an active component is specified by an STD. The STD defines an initial location and a number of *final* locations. Well-formed STDs for active components have at least one final location; this is checked by the generator. The active component keeps its local control and data state between subsequent activations.

Messages passed to other active components will be queued and processed in FIFO order by the receiving component, while messages to passive components are immediately processed (see below).

Passive components

Passive components are activated whenever the component receives a message from another component. The activation ends whenever a final state is reached.

In terms of real-time programming, the implementation of a passive component is best described as a number of mutual exclusion protected procedures performing reads and updates on a central memory resource.

Passive components are mutual exclusion protected, that is, no two active components may access the same passive component at the same time. In an OSEK-based implementation, for instance, this would be ensured by using the default priority ceiling policy for shared resources.

Like an active component, each passive component is defined by an STD. An STD of a passive component is well-formed if all outgoing transitions from final locations have exactly one input statement with a nonempty expression, and all other transitions do not have input statements.

Because messages sent to passive components causes their sequential activation, the calling components may not take advantage of some aspects of parallelism allowed in the basic AutoFOCUS model: The calling component may not have a transition with more than one nonempty output statements, and the call may not be distributed to more than one port.

Timed STDs

MoDe system models are interpreted as discrete-time models where each system step corresponds to a fixed time period. A logical component delays the overall computation by remaining in a non-final state for a number of steps. In principle, the expressiveness of STDs is sufficient for modeling delays; however, the resulting models tend to be somewhat clumsy. The MoDe tool provides two extensions of the basic STD notation, *clocked STDs* and *timed STDs* [16].

Clocked STDs allow local variables to be qualified as *clocks*. Clocks are of (nonnegative) integer type and are increased by one with each system step; the value of clocks may be modified by the STD in the same way as local variables, i.e. resetting and comparison of a clock's value is possible. Each of the locations is optionally labelled with an additional *invariant* over the component's clocks; the invariant defines which clock valuations are permissible for the respective location. For those system states in which the both the invariant and some outgoing transitions evaluate to true, the resulting behavior is nondeterministic. In combination with preconditions over clock valuations for outgoing transitions, invariants allow for the specification of *counting locations* modeling execution time for computations. Note that the current generator does not check whether time is allowed to progress to infinity by the clocked STD. In principle, this requires that for each location and each state, either the invariant evaluates to true, or one of the outgoing transition's preconditions.

Timed STDs have the same syntax as basic STDs; in addition, transitions may have *timing labels* of the form `«time(MIN;MAX)»` or `«time(VAL)»`, where `MIN`, `MAX`, and `VAL` are expressions of integer type. The `(MIN;MAX)` notation specifies a nondeterministic interval, while the `(VAL)` syntax refers to a deterministic delay.

Figure 6 shows the timed STD of the component `SafetyComponent`. The safety component may perform one of the following two actions:

- If the methane value is within a safe range (`x <= DangerMethaneValue`), a `Start` command is issued to the pump controller
- If the methane value exceeds the maximum (`x > DangerMethaneValue`), or if no methane value has been available for a certain time interval (`t - lastTime > TMaxSilent:MethaneValue?::`), then a `Stop` command is sent to the pump controller.

We assume that the execution time for either of

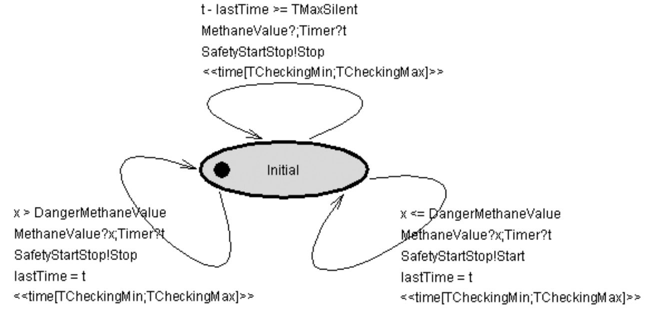

Figure 6. Timed STD for `SafetyComponent`

the three transitions is between `tCheckingMin` and `tCheckingMax`, which are predefined constants. The generator will translate the timed STD to a clocked STD. In the translation step, a new clock c is added to the component. Each timed transition δ with label PRE_δ : IN_δ : OUT_δ : POST_δ :`«time(MIN;MAX)»` is replaced with

- a counting location with invariant $c <$ MAX modeling the ongoing computation,
- a transition labeled PRE_δ : IN_δ :: $\text{POST}_\delta \wedge (c = 0)$ from the source location to the counting location,
- and a transition labeled $c \geq$ `MIN` :: OUT_δ : from the counting location to the destination location.

4.2 Technical model

The properties of the technical platform are specified in the technical model. The technical model is an independent and self-contained view of the system.

Technical models consist of networks of *nodes*, *links* between nodes, and *connectors* on nodes. *Nodes* represent physical computers with their associated I/O interfaces and their operating system. Specification of node-specific properties like scheduling policy, operating system, etc. is done by a number of tags on the respective nodes. In our example, all three nodes use the OSEK operating system with a (nonpreemptive) Fixed Priority Scheduling policy.

Links denote communication links between nodes which can be either directed or undirected. Every directed link may be connected with a source connector and arbitrarily many destination connectors (broadcast). For undirected links there is no inherent restriction on the number of connectors, but restrictions on the number of connectors may be part of the associated *link type* definition. The behavior of the link is defined by associated library components. The corresponding drivers are also identifed by link types. In

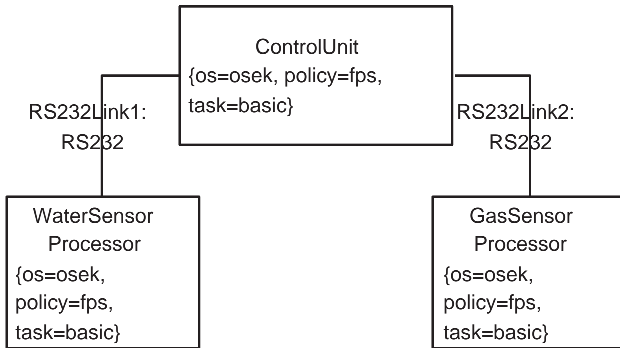

Figure 7. Technical model

the mine pump example, both links are of type `RS232`, and the corresponding driver components are taken from the MoDe communication library.

Connectors denote those "access points" of a node where links can be attached. Connectors, like links, are typed; for directed link types, the connector carries an additional source/destination flag. Naturally, the connectors of a given node are restricted to those protocols that are supported by libraries for the component.

Fig. 7 shows the technical model for the mine pump example in a simple graphical notation. The technical model consists of the following entities:

- Three nodes for processing signals from the water high/low sensors (`WaterSensorProcessor`), processing signals from the gas sensors (`GasSensorProcessor`), and a central control unit (`ControlUnit`), respectively.

- Additional OS specifications for the nodes. Each OS specification is shown as a tag in the graphical notation. The `os` key selects the library used for OS abstractions, while the `policy` key is set to the desired scheduling policy (fixed priority scheduling for the example).

- Two directed links of type `RS232` from the two signal processing units to the control unit.

- Four connection points defining the interfaces from nodes to links - not shown in the diagram.

4.3 Synthesis descriptor

The third element of the deployment model is the synthesis descriptor. The synthesis descriptor currently consists of two parts: a `mapping` part, and a `schedule` part. Figure 8 shows the example synthesis descriptor in a pseudo language. Though conceptually the synthesis descriptor is considered a separate document, the MoDe tool facilitates navigation within the model by supporting direct annotations to diagrams of the logical and technical models.

```
syntdesc{
  mapping {
    {PumpController, MethaneValueComponent,
      SafetyComponent} mapsto ControlUnit
    MethaneSensorComponent mapsto GasSensorProcessor
    {HighSensorComponent, LowSensorComponent}
      mapsto WaterSensorProcessor
  }
  schedule {
    priorities {
      {SafetyComponent, 1}, {MethaneSensorComponent, 1},
      {HighSensorComponent, 2}, {LowSensorComponent, 1}
    }
  }
}
```

Figure 8. Synthesis Descriptor

Note that if the Fixed Priority option is chosen in the technical model, each leaf component must be assigned a priority. In the example, only the two water sensor tasks need priorization, and all other priorities are set to 1. Priorities for passive components remain without effect in this case as OSEK implements the Priority Ceiling convention for shared resources, so the run-time priorities can be inferred from the priorities of the active components.

4.4 Service Timing Functions

MoDe models the timing of complex computations in terms of the delay of simple computations (basic blocks). Assuming that these basic blocks can be defined so that the result is sufficiently precise for analysis, then basic blocks will in most cases correspond to basic HW services such as processor operations, and the timing of complex algorithms running in HW or SW may be described as some function of these basic blocks, possibly varying with additional parameters. In MoDe, these mutual timing dependencies are expressed with service timing functions (STFs).

An STF is a function in AutoFocus's built-in functional language yielding a nonnegative integer for a timing estimate. STFs may be passed a number of parameters that express some data-dependence of the delay. An STF is either associated with a component in the logical model, or a node in the technical model.

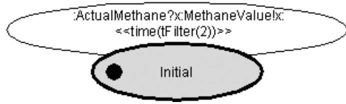

Figure 9. Timed STD for `MethaneSensorComponent`

STFs may compute their result by calling other STFs; in particular, component STFs may refer to the STFs of the corresponding node. This is indicated by the prefix `$Node$`.

In the mine pump example, component MethaneSensorComponent reads a value from an external sensor port `ActualMethane` and forwards its value to the `MethaneValue` port. MethaneSensorComponent filters the sensor signal in order to eliminate noise. The STF `tFilter` is used to derive an estimate for the worst case execution time of the filter; note that the model abstracts from the actual filter operation.

The component's STF `tFilter` models the worst case execution time for a discrete Butterworth filter of order n:

```
MethaneSensorComponent_tFilter: Int -> Int;
fun MethaneSensorComponent_tFilter(n:Int) =
  $Node$_tIntAdd(16) * (2*n)
+ $Node$_tIntMult(16) * (2*n+1);
```

The corresponding node, `GasSensorProcessor`, defines two service timing functions, `tIntAdd` and `tIntMult`, giving some estimates for the worst case execution time of integer additions and multiplications on its processor:

```
GasSensorProcessor_tIntAdd: Int -> Int;
fun GasSensorProcessor_tIntAdd(16) = 2 |
  GasSensorProcessor_tIntAdd(32) = 4;
GasSensorProcessor_tIntMult: Int -> Int;
fun GasSensorProcessor_tIntMult(16) = 13 |
  GasSensorProcessor_tIntMult(32) = 22;
```

The STF `tFilter(2)` evaluates to $2\times(2\times2)+13\times(2\times2+1) = 73$. Consequently, `MethaneSensorComponent`'s transition will be delayed by the same number of steps.

4.5 System model

The system model defines the semantics of a deployment model. While components in the basic AutoFOCUS model are assumed to run in parallel, each set of logical components mapped to the same "abstract" node in the technical model are executed sequentially. Therefore, a logical component is not in an activated state during all system steps, so certain abstractions are necessary to ensure that messages are not lost.

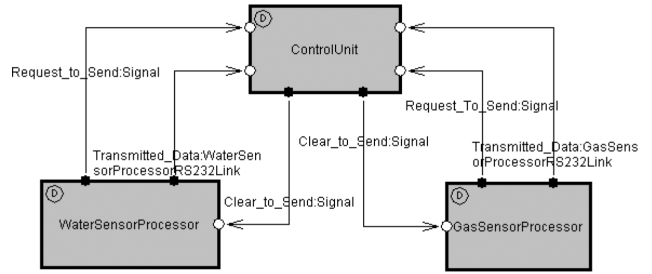

Figure 10. SSD for system model (top level view)

There are two ways control is passed between components running on the same abstract processor: *sequential* and *scheduled*. Sequential passing of control closely resembles a procedure call in the implementation; it applies whenever a message is sent to a passive component. Scheduled passing of control may happen at any point in time during execution of the system.

Active components enforce a scheduled passing of control whenever one of their final locations are reached. Note that the passive components may be accessed by several active components.

After the synthesis performed by the system model generator, a AutoFOCUS system model with abstractions for scheduling, drivers, and distributed communication results. Figs. 10 and 11 show the top-level structure and a part of the model for the `ControlUnit` node, respectively. Note that the hierarchy in the logical model is flattened.

5 Results

5.1 Models

We have used our framework to model both the Mine Pump example and a simplified HW/SW architecture of a car periphery supervision system by an automotive supplier.

5.2 System Model Generator

The MoDe method is supported by a tool extension of the publicly available AutoFOCUS tool. The generator is implemented in about 12,000 lines of Java code.

As an input notation, the MoDe generator currently reads annotated AutoFOCUS models. Because of the

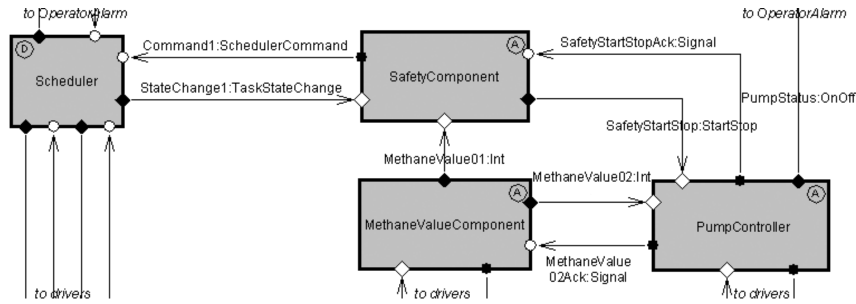

Figure 11. SSD for `ControlUnit` (detail)

difference in syntax between deployment models and basic AutoFOCUS models, the generator applies an additional well-formedness check on the deployment model, and automatically generates system models if the deployment model is found to be correct. The system model is used for simulation or quantitative real-time analysis.

5.3 Analysis

In architecture-level design, the following questions are of major interest for analysis:

- Are the system-level timing requirements fulfilled for the given architectural choices?
- What is an optimal scheduling and partitioning of the system with respect to timing requirements and robustness?
- How flexible is the system with respect to addition of further functionality?

From the application point of view, the most interesting real-time analysis capability may be classified as *worst case response time* analysis.

MoDe system models have a discrete time interpretation. This abstraction works very well for systems with tight global synchronization, like synchronous hardware. For the kind of systems considered here, we do not necessarily assume global synchronization; asynchronous links between nodes are abstracted with finite FIFO buffers, and it is assumed that the system model does not underestimate the ratio of production rate to consumption rate. We further assume that the degree of precision allowed by our approach is adequate for the architectural-level analysis desired.

We have subjected the mine pump model to quantitative real-time analysis using the NuSMV model checker [7]. A simplified version of the mine pump system model was automatically translated to an equivalent specification in the SMV input language [18]. For the two timing requirements stated above, the corresponding pairs of events were characterized as statements in the temporal logic CTL. Using the upper bound algorithm in [5], the maximum delay between the event pairs was computed by NuSMV. For the mapping of (conceptual) real time to system steps, a basic period of 1ms was assumed. For the mine pump model, the analysis proved to be computationally tractable if abstractions for datatypes were introduced, and if the environment of the mine pump (water level) was further characterized by a simple model. We found our preliminary results encouraging, and expect quantitative real-time analysis to be a future cornerstone of our approach.

6 Related Work

The concept of active and passive components is inspired by Burns and Wellings's model as described in [11]. In contrast to the programming-language and scheduling-theoretic foundations of the Burns and Wellings model, MoDe is intended to be susceptible to formal analysis using assertional methods to prove real-time properties.

The POLIS [1] approach of UC Berkeley models systems as an asynchronous composition of components defined either by an FSM-like formalism or a composition of Esterel processes. Verification of timing properties in POLIS is mainly based on simulation of models based on C code or discrete event simulation. MoDe models are somewhat closer to the actual concepts supported by common operating systems, so a wider spectrum of inter-task communication primitives is supported.

There are several current approaches that compile distributed implementations from specifications in synchronous languages [2][6]. As opposed to MoDe, scheduling and timing is a priori assumed to meet the synchrony assumptions implicit in the language, so a

separate timing analysis is not necessary.

7 Conclusion and Future Work

We have presented a concept architectural-level design and analysis of real-time systems using a formal notation. Our approach is aligned with the predominant platforms in the domain of automotive control applications.

A more synthesis-oriented use of MoDe models is straightforward and can be built on existing technology. Being a synchronous formalism, AutoFOCUS leads itself naturally to both software [4] or hardware synthesis, similar to other synchronous formalisms such as SpecCharts [17], Esterel [3], or Statecharts [8].

An integrated simulation tool should allow interactive simulation of the system model while instrumenting the design with the logical and technical views on the system. When realized, we expect this option to help users evaluate models while retaining their original abstractions.

Quite obviously, the method described here requires some further experimental evaluation. This should include combination with existing approaches for BCET and WCET analysis [12], and a detailed comparison of analysis results with the actual timing of an implementation.

Quantitative real-time analysis is subject to the same scalability issues as other formal methods based on full state-space exploration. Though it is too early at this point for qualified judgement, we feel that automated techniques for abstraction, or incomplete searches of the state-space, may improve scalability in the future.

An extended version of this paper can be found in [16].

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer, Boston, 1997.

[2] A. Benveniste, P. Caspi, P. L. Guernic, H. Marchand, J.-P. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proceedings of EMSOFT 2001*. Springer-Verlag, 2002.

[3] G. Berry. A hardware implementation of Pure Esterel. Technical Report 06/91, INRIA, Sophia-Antipolis, France, 1991.

[4] A. Blotz, F. Huber, H. Lötzbeyer, A. Pretschner, O. Slotosch, and H.-P. Zängerl. Model-based software engineering and ada: Synergy for the development of safety-critical systems. In *Proc. Ada Deutschland Tagung*, Jena, 2002.

[5] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 1994.

[6] P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *Proceedings of SAFECOMP 2001*, pages 215–226, 2001.

[7] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open source tool for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, Copenhagen, Denmark, July 2002.

[8] D. Drusinsky and D. Harel. Using statecharts for hardware description and synthesis. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, 8(7):798–807, July 1989.

[9] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd Workshop on Computer-Aided Verification*, LNCS 531, pages 176–185. Springer, 1990.

[10] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[11] M. Joseph, editor. *Real-time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996.

[12] R. Kirner and P. Puschner. International workshop on WCET analysis - summary. Research Report 12/2002, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[13] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, Boston, 1997.

[14] OSEK VDX consortium. *OSEK/VDX Operating System Version 2.2*, 2001.

[15] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model based testing in incremental system development. *Journal of Systems and Software*, 2003. To appear.

[16] J. Romberg. The Model-Based Deployment approach. Technical report, Technische Universität München, Institut für Informatik, 2003. To appear.

[17] F. Vahid, S. Narayan, and D. D. Gajski. SpecCharts: A VHDL front-end for embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):694–706, 1995.

[18] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification based test sequence generation with propositional logic. *Journal of Software Testing, Verification & Reliability (STVR): Special Issue on Specification Based Testing*, December 2000.