# Development of a Car Seat: A Case Study using AUTOFOCUS, DOORS, and the Validas Validator

Peter Braun
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
D-85748 Garching b. München

Oscar Slotosch
Validas Model Validation AG
Software-Campus
Hanauerstr. 14b
D-80992 München

**Abstract:** In this paper we describe the modeling process and the resulting model of a typical car seat. The requirements of this seat are documented in [Chr00] which are the input of our process. We used the tools AUTOFOCUS [AF-02], DOORS [Tel02], and Validas Validator [Val02]. Starting with requirements analysis we develop first model fragments. Afterwards the graphical, component oriented approach of AUTOFOCUS is used to model the system. Requirements management and tracing techniques ensure that all requirements are implemented. The model-based core of the development process helps very much for the requirements tracing. The model fragments of the earlier phases can be updated so that tracing information is consistent. Compared to traditional requirements tracing techniques less manual interaction is needed.

Beside this the test management is also done based upon the requirements. For relevant requirements test cases are specified. This is done using the AUTOFOCUS notation of Extended Event Traces (EETs) a variant of Message Sequence Charts (MSCs). Afterwards the generated code of the model is tested based upon those test cases. Further validation techniques like simulation, consistency, and determinism checks of the Validas Validator have led to the detection of inconsistencies in the model and in the specification.

## 1 Introduction

In the following an adapted process for the development of embedded systems with AUTOFOCUS, DOORS, and the Validas Validator is shown by an example system dealing with the control of a typical car seat. The process starts with some requirements analysis activities. Our graphical, component oriented approach is used to model the system. Requirements management and tracing ensures that all requirements are implemented.

Component orientation is a special form of object orientation. The used description techniques are similar to UML-RT. We use the tools AUTOFOCUS and the Validas Validator to check consistency and to generate code. One difference to UML-RT is that we use EETs [HMS+98, HSS96] instead of MSCs. EETs have a precise semantics, and can be used to express repetition for a certain time, and to generate tests. Simulation is used for building

a prototype of the software parts of the car seat controller. We implemented a GUI for testing the model, based on the given interface specification.

Components have (compared to objects and classes) some advantages. They are static, so their size can be determined which is important for efficient C code generation. The simple, well typed communication model used allows to determine the size for messages and buffers as well. Together with the scheduling concept for the components the model is a perfect base for real-time applications. For prototyping the model we implemented a timer component, and used the Java method `currentTimeMillis()` to access the system time.

In the development of the model we put more effort to the process (requirements structuring), modeling, and prototyping. C-Code generation was not in the focus, because it was easier to test the Java code and integrate it with a simple GUI. The total amount of time spent with this case study was approximately four weeks.

We start with a short description of the method and refer the applied tools. Section 3 contains a sketch of the model and the applied design principles. Section 4 shows some of the validation methods used.

## 2 Method

Nowadays even embedded systems become more and more complex. Beside this challenge even embedded systems are heavily interconnected. To guide developers new methods have to be established. Those methods should be compositional and hierarchical so that the functionality of the target systems can be split into different hierarchical components. The methods should support the developers in every stage of the development from requirements engineering over design till validation and test of the developed system. Equally important is that tools support this method.

The UML [UML99] provides notations for the development of object-oriented systems. These notations are loosely coupled and are heavily influenced by object-oriented programming languages like Java, Smalltalk, or C++. The UML defines no method how this language may be used. There are many different methods which use the UML in the context of internet or business applications, but especially in the context of embedded systems there are only first steps towards methods. As the object-oriented approach used in the UML doesn't seem to support the development of embedded systems very well, another component-oriented language named UML-RT will be integrated in future versions of the UML. The notations of this language support hierarchical components, beside some other concepts and fit the needs better than pure UML.

In the following we will describe some facets of our method, which we have used to develop the software part of the seat controller described in [Chr00]. Our method provides a component based approach to develop and describe the software part of embedded systems. We use fewer notations than provided by UML or UML-RT, but most of the notations are very similar to their counterpart in UML/UML RT. A main difference is that our nota-

tions are founded by a mathematical theory and therefore they are integrated very tightly. Note that a developer has not to know this mathematical theory to access the benefits.

As stated above tool support is essential. Our method is based upon a tightly integrated tool chain. The tools we use are DOORS from Telelogic for Requirements Engineering and Requirements Management, our own component-oriented CASE-Tool AUTOFOCUS for the specification of software systems and code generation, and the Validas Validator for validation and verification support.

### 2.1 Requirements Engineering and Requirements Management

Usually the software development processes starts with analysis and specification of the problem space. The process starts with roughly structured User Requirements which are transformed into System Requirements. This is a very universal process which deals with lots of informal notations and very few structure. Tools like DOORS support those steps by providing the possibility to structure text to some extend and by focusing on requirements management and tracing. The support given by DOORS mainly helps developers to not lose the overview and to manage the relationship between different kinds of information at this early stage. Starting with the System Requirements the step from the problem space towards the solution space has to be done. This step from "What" to "How" or from analysis towards design has to be taken carefully. It is essential that this step is at least traceable. Preferable this step is provided by a continuous method, so that as many links as possible between design and analysis information are generated automatically.

In our method we first import and split the requirements so that they can be managed with DOORS. The result of this step are one or more documents with all System Requirements. The step to these System Requirements from the User Requirements can be supported by DOORS. The resulting documents are plain text, which is structured into smaller quantities which contain requirements. Now these text blocks are classified by their kind. They may specify requirements for the software, the hardware, the environment, or any combination of them. This classification is important as we only concentrate on software. Design of hardware could be carried out in parallel. The decision which parts of the overall system are realized in software and which are realized by hardware can be revised later. The decision that a part is realized in software is a decision, that description techniques provided by AUTOFOCUS are used to describe the functionality of that part.

After this classification a new document containing the requirements for the software system could be generated. The generation ensures the traceability between the original document and the document with the software requirements. Depending on the structure of the original document some work has to be done, to adapt the new document so that it is at least readable and structured appropriately.

The next step is the identification of some coarse model-frame for the design. Therefore the software requirements document is used. Here some requirements are attributed with e.g. components or component-types fulfilling those requirements. Within a new document "model-frame" these recognized component-types are further described. Instances

of component-types can be connected to other component-types using links. The model-frame document contains further model-types and relations.

Using the information in the "model-frame" a so called surrogate module can be generated which is another DOORS document. This document is used to generate a first AUTOFOCUS model, which can be developed further using AUTOFOCUS. As every "object" in this surrogate module has links to the "model-frame" and the software-requirements document and that contains links to the original requirements document it is initially ensured, that a developer can see all requirements which resulted in model-elements. As the model is developed further a translation back to DOORS is possible, using the abovementioned surrogate module (which relates the unique identifiers of model-elements used in DOORS and in AUTOFOCUS). Surely new relevant model-elements must be related to their requirements appropriately.

With this technique it is at least possible to control if all recognized requirements are "satisfied" by some model-element. Further it is possible to show at all requirements for a given model-element, and even more important to identify all model-elements which are related to a requirement. This traceability is very important if some requirements are changed.

Figure 1 shows an overview of the above described process. The process starts with the system requirements. The system requirements are structured and classified. A resulting document containing the software requirements is generated. This document has to be further refined and structured. By identifying some model-elements within the software requirements and by providing a model-frame, some first fragments of a model can be generated. In parallel the development of test-sequences ensuring some software requirements can be described within the test-frame. From the test-frame EETs containing these test-sequences can be generated.

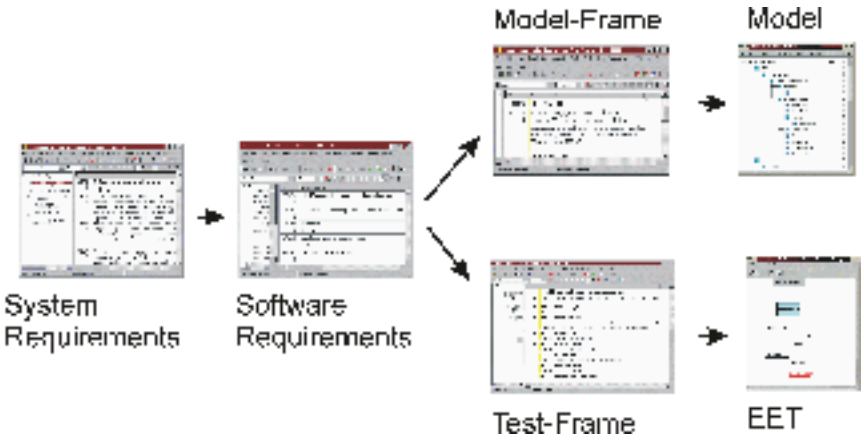


Figure 1: The overall process

## 2.2 Design

The design phase the model build upon the recognized model-elements is further developed. The design is carried out using AUTOFOCUS which supports an important subset of UML-RT:

- System Structure Diagrams (SSDs) describe the system structure and the interfaces of the components.
- State Transition Diagrams (STDs) describe the behavior of the System.
- Data Type Definitions (DTDs) define data types which are e.g. used for the specification of communication channels of the system. Together with the data types auxiliary functions dealing with them can be defined.
- Extended Event Traces (EETs) describe the dynamic behavior by example communication sequences between the components. EETs are automatically generated during simulation or from other validation techniques.

All description techniques are hierarchical, so that the model can be structured appropriately.

The system is developed top-down, i.e. the structure is designed hierarchically. Additional requirements lead to refinements or incremental changes of the system. It is easy to extend the interfaces in order to send additional messages required for additional features. Requirements tracing allows the designer to check if all features have been modeled and tested.

## 2.3 Validation

Building a model with AUTOFOCUS is quite simple, however building a correct model is not. The first step is to check the consistency of the model. This allows to detect for example unconnected channels or misspelled ports in transition diagrams.

The next validation step is to simulate the model (or parts of it). Simulation shows the developer the dynamic behavior of the system and allows to debug the behavioral descriptions. Within the simulation the model is animated according to given inputs using the input examples of the user.

Additional validation techniques allow the developer to detect nondeterminism in the models and to verify that no messages are lost, or certain inputs lead to certain outputs. In the example all components are simple enough to apply formal methods (model checking) for validation, however the whole system cannot be model checked.

Simulation cannot guarantee correct models as usually only some trace through the system can be tested. Unfortunately automated validation techniques like model checking also cannot be used for most practical relevant systems as those systems are often already to

complex. But model checking can be used to ensure the correctness of some smaller parts of the system. Often even those smaller parts have to be modeled more abstract with e.g. restricted data types. So this is usually only done with some critical parts of the system.

After the validation of the components and the system the integration test is build. The generated code supports textual inputs (test driver generation). This allows the automated run of test cases specified by text files.

In addition to the integration test we build a GUI for testing the system using the given interfaces.

# 3 Model

In this section we describe important principles of the process and the model.

## 3.1 Requirements Engineering and Requirements Management

As described in Section 2.1 we start with the requirements specification of [Chr00]. First we imported the original text into DOORS. Thereby we divided the text into smaller text blocks (requirements). After that those requirements are classified as described above. In Figure 2 some parts of the specification describing the seat heating are shown.

After the classification into requirements for the hardware, the software and the environment a document containing the software requirements is generated. This document is very similar to the original document as the original document already is focused on the software. This is not the general case as some case-studies with BMW have shown.

During further "development" two main components in the "SeatControlModel" are identified. The Heater deals with requirements of the seat heating, the MotorController handles the control of the motors. Within the MotorController, components dealing with the memory, the switches, and the hall sensors are identified. Further components for the core control of each motor are identified.

The document shown in Figure 3 contains these initial model elements. It describes components and component-types. Some further attributes of those components are identified and described here too. E.g. local variables of each motor containing their current position and their maximum positions are identified in this phase.

Starting with these components further modeling in AUTOFOCUS is carried out. To control the development at some stages where the model or parts of it seems to have reached a stable state, the back-translation to DOORS was done. Here the description of the model-elements had to be further refined and the links of the model-elements have to be adapted. So after these steps one can see if there are further requirements, which are not yet "satisfied" by a model-element. The relation between model-elements in AUTOFOCUS and DOORS is stored in a surrogate module. This surrogate module is normally hidden.

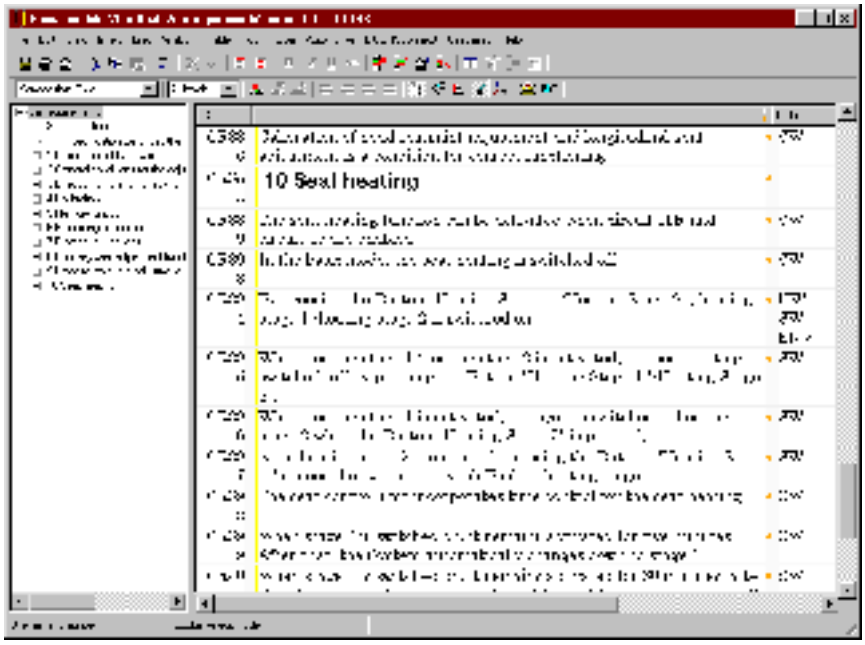


Figure 2: Requirements for the seat heating

Beside these control-steps ensuring that every software requirement is fulfilled by some model-element, the management of tests was done similarly. So for some interesting requirements a test-case was written. These test-cases were specified graphically by EETs, a variant of MSCs used in AUTOFOCUS. So it can be seen, if there are test-cases for a specific requirement. For every test-case the requirements leading to a test-case can be shown. In this case-study the test-cases and the model are developed independently by different developers. An example for a test-case which is linked to a requirement is shown in Figure 4.

### 3.2 Design

The model is described using simple data types for internal and external messages of the system. System structure diagrams are used for the description of the architecture of the system. The behavior is described using auxiliary functions dealing with data, and state transition diagrams for the behavior of atomic components. Event traces describe interactions within the system. Event traces are generated during simulation.

The model covers all requirements as far as this has been possible from the short and imprecise specification.

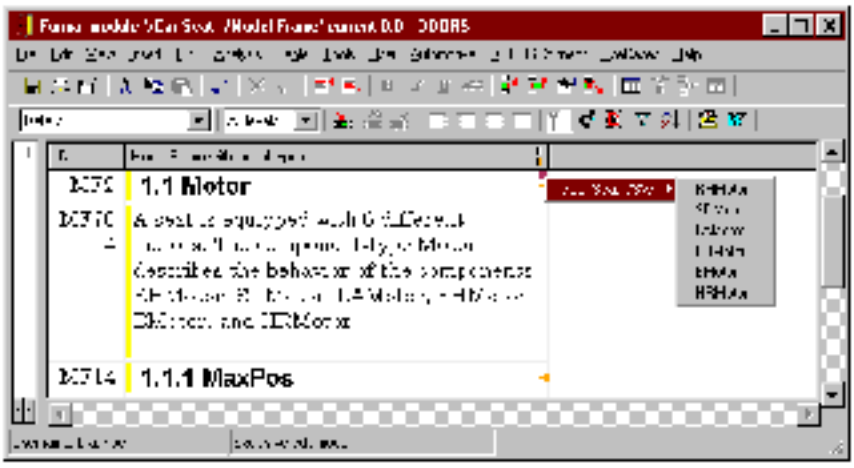


Figure 3: Model Frame

### 3.2.1 Interfaces

The design of the model has been started from the given interfaces. In order to apply our method to the given interfaces, the interfaces are transformed into a protocol definition that describes the interface values of the system model.

The interface of the model is described in Figure 5. We used the following DTDs to define the interfaces:

```
data SeatSwitches = LAfwd | LArev | LAstop
  | RHup | RHdown | RHstop
  | SDfwd | SDrev | SDstop
  | Bfwd | Brev | Bstop
  | FHup | FHdown | FHstop
  | HRup | HRdown | HRstop
  | Mdown | Mup | M1down | M1up | M2down | M2up
  | Heat1pressed | Heat2pressed;

data CarEnvironment = setDoorOpen | setDoorClosed
  | setClamp15Off | setClamp15C | setClamp15R | setClamp15
  | setClamp15x
  | setSpeed(Int) | setVoltage(Int);
```

### 3.2.2 Structure

The structure of the system is derived from the requirements that have been structured into functions. The system has two main functions: one for the motors, and one for the seat heating. The structure refines the interface model of Figure 5 and is described by the SSD

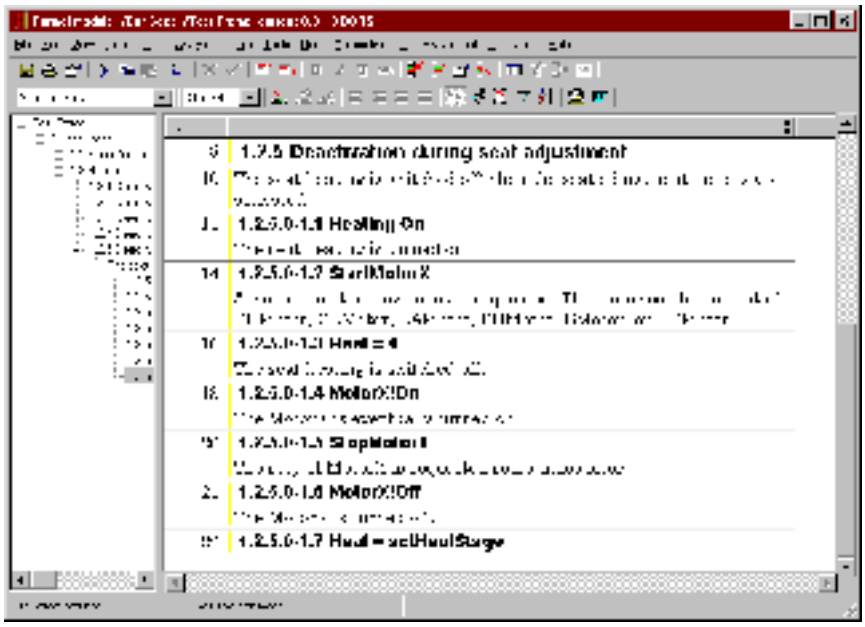

Figure 4: A textual specified test-case

shown in Figure 6. Since the heating shall be switched off if the motors are running, there are channels from the `MotorController` calculating if the motors are activated to the `Heater`. The system environment is used by both functions.

We start with a description of the `Heater` (see Figure 7). The `Heater` model consists of several components. The core component is `HeatControl`, it contains the controller that reacts on messages from the environment and from the panel. This component interacts with a timer for time control (`HeatTimer`). Furthermore there are additional checks (performed by other components). If one check fails a signal is passed to the merge component `MergeCheck`). `MergeCheck` combines these signals (logical or) together with the Motor-Commands that also switch of the heating. The arbiter `HeatArbiter` switches the heating off, if some checks fail. If the checks are OK, the arbiter switches the heating on to the last (or current) value. `HeatCheck15R` checks the position of Clamp15.

The more interesting part is the description of the motor controllers. Several design patterns for embedded modeling have been applied to design the structure of the motor controller:

- FlipFlop: translates button events into states and send them to other components constantly.
- Scheduler: ensures that at most two motors are running in parallel, and that at most one motor receives a command at a time.

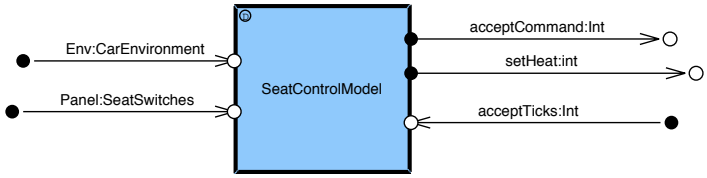


Figure 5: SSD of the System Interface

- Messaging: instead of global variables messages are used from memory components to the relevant components.
- Splitting: the acceptCommand signal is split into several signals for the different motor components.
- Merging: if several commands (or other signals) for one component are present, they are merged according to their priorities.
- Local timing: for the modeling of real-time behavior timer components are used where they are needed. Other possible variants are the use of a global time or an external time.

These modeling patterns (and others, for example for cryptography) have been developed during many projects with AUTOFOCUS.

In this paper we do not show the whole system, but for space reasons we only describe some components more detailed.

### 3.2.3 Behavior

The behavior is described by numerous state transition diagrams for each atomic component. For similar components STDs are reused several times. AUTOFOCUS allow to assign to each component a behavior (STD), such that STDs, can be reused. This feature has been used heavily, because six motors six controllers (and six timers, etc.) have been modeled.

The behavior of the timer is shown as an example in Figure 8.

The real-time behavior of the timer is ensured by using the system clock (instead of the variable `CurrentTimeMillis`).

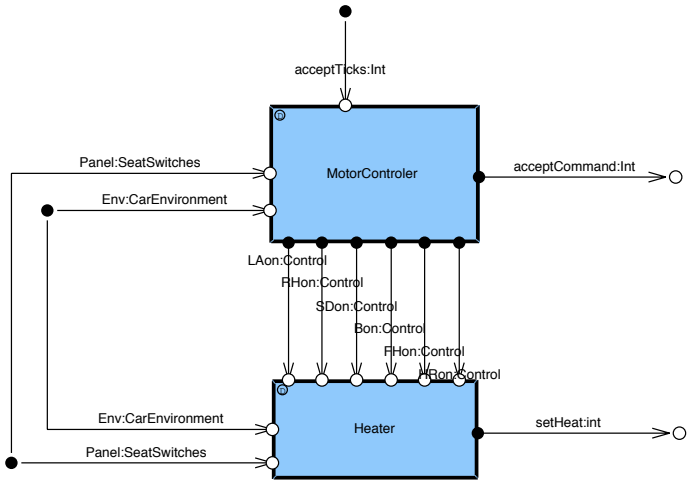


Figure 6: Structure of `SeatControlModel`

### 3.3 Code

The code consists of a generated model, and a manually implemented wrapper class that implements the given interfaces. For testing we added two other classes: one for the system environment (that implements `CarEnvironment`) and one for running the test. We used threads for running them independently.

#### 3.3.1 ValidasSeat

Since neither the simulation, nor the prototyping code implements the given seat interfaces a simple wrapper has been designed, and manually implemented (ValidasSeat.java). The wrapper initializes the model and implements the methods by simply sending the events to the system. Since AUTOFOCUS has a synchronous execution model, it has to be ensured that no messages are lost when passing them to the model. Therefore we used a asynchronous model wrapper around the synchronous model. This model wrapper has synchronized input queues and passes the values to the core model (of course this wrapper is also generated).

The interfaces have been modeled with data types for all possible method calls. For example the data type `SeatSwitches` has (among others) the following values:

```
data SeatSwitches = LAfwd | LArev | LAstop
                  | SDfwd | ...   | Mup;
```

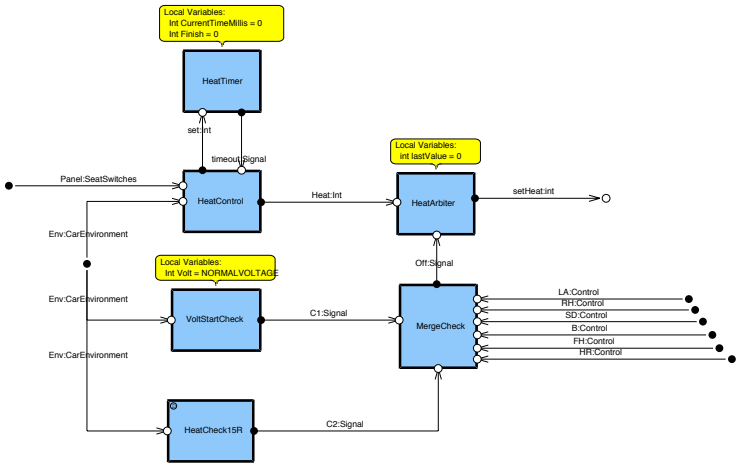


Figure 7: Structure of `Heater`

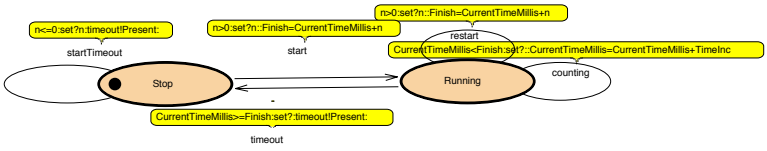


Figure 8: Behavior of Timers

These values are passed to the system every time the corresponding method is called. The return values are processed similar.

# 4 Validation

In this section we describe the applied validation techniques. Within the given requirements there are no quality challenges like for example:

- ensure that every user input has an effect,
- ensure that no signals are lost (due to nondeterminism),
- ensure that the response time for a motor is below 0.1 s,
- every state in the description is reachable and tested.

We do not concentrate on the validation according to those requirements, even though the Validas Validator supports the validation of critical properties using advanced methods

[Slo98]. We also don't use the generation of test cases according to different coverage criterion, in spite we use some manually specified test cases from the requirements specification.

The applied validation techniques are:

- consistency checks,
- determinism checks,
- graphical simulation of components and the whole system,
- testing the model (textually and in batch mode),
- testing the system via the specified interface and a simple GUI, and
- requirements tracing (to the model and the tests).

In this section we describe some test results and test procedures.

### 4.1 Improved Models

Several errors have been detected and corrected, the most interesting error was found during simulation of the complete system. It showed an integration error of the heater and the motor controller due to the switch off signals from the scheduler: The scheduler does send alternating commands to motors of group 1 and group 2. These signals are used to switch off the heating during motor movement. The result of this toggling signal was a toggling heating signal. A simple delay in the merge component of the heater fixed the error.

### 4.2 Consistency Checks

Several copy & paste errors have been found using the consistency checks, especially during bottom-up operations, when additional features have been added.

### 4.3 Simulation

AUTOFOCUS models are complete, (even if they do not allow to import code for components, actions, etc.). This allows to generate code for products, testing and simulation from the models. Simulation runs graphically.

With simulation the dynamic behavior can be evaluated. The method requires to test all requirements to components and to the system. For example the requirement that the heating shall be switched off, if motors are running (see Fig. 9) is tested by entering

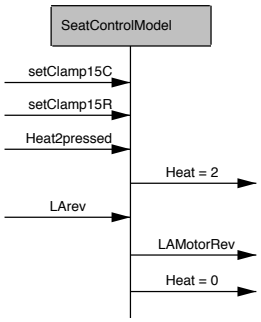


Figure 9: Heater Requirement

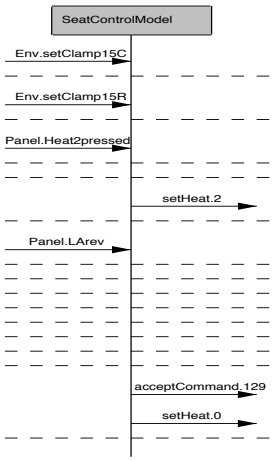


Figure 10: Protocol of Heater Simulation

the commands into the simulation environment. The result is a simulation protocol that contains the port names for the input values, the concrete values (also for the output), and the ticks (timing information represented by dashed lines). Figure 10 shows the protocol.

In addition to the system view also inner views can be animated. For example for developers it might be interesting to see if the timer for the heater has been set to the correct value. AUTOFOCUS simulation also generates EETs for the subcomponents.

The real-time behavior depends on the time required for a single step (tick) on the concrete system. For the graphical simulation the Timer can be configured via the constant `TimeInc` in the DTD `Misc`. The value of the constant represents the amount of time (in ms) the timers are incremented each step. This allows a flexible simulation of the real-time behavior.

### 4.4 Deterministic Check

Since we are working with a synchronous hardware oriented model, some events can occur simultaneously. In components with several inputs this can cause nondeterministic situations. Furthermore there are no message queues in the semantics, such that messages can get lost if they are not processed. The determinism check of the Validas Validator helps to detect such situations. For example in the Controller of the heat component the some nondeterministic situations have been detected (see Figure 11). Since the timeout of the timer does not occur frequently, it is very improbable that this error would have been found during the simulation (Note that the simulation of AUTOFOCUS also detects nondeterministic situations if they occur).

In a similar way it is possible to check completeness. For safety critical systems it is important to ensure that all messages are processed. This can be done using the Validas

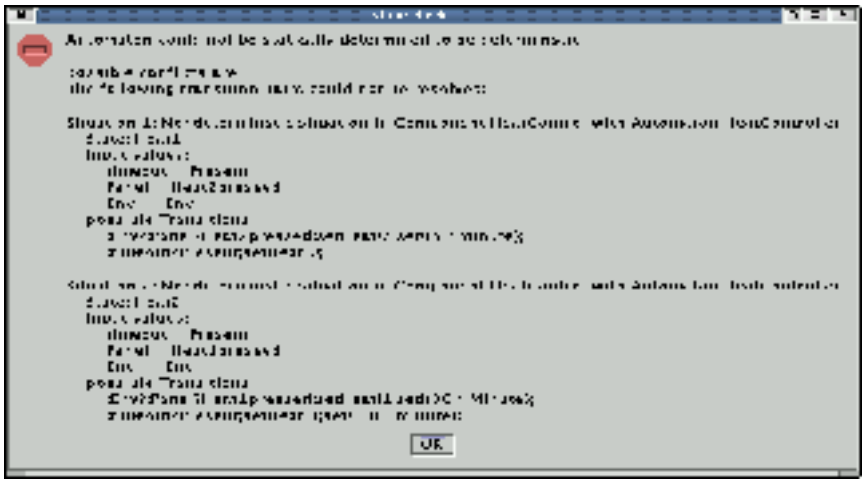


Figure 11: Nondeterministic Situations in the Heater

Validator. For example it could be detected that certain states do not process timeout interrupts.

### 4.5 Testing

There are several forms of testing the model, the simplest one is to simulate the system interactively (see Section 4.3). Next step is to simulate the generated code without graphical animation (textually). The Validas code generator supports this form of testing with an interactive code that can be executed.

### 4.6 System Test

In order to apply an overall system test, a graphical interface has been build (see Figure 12). The interface uses the specified interfaces (seat_interface) and visualizes the state of the tests.

## 5 Conclusion

The component oriented approach with the synchronous model is working fine. The hierarchic description techniques are very helpful in keeping the system modular (all state transition diagrams have less than ten states).

Figure 12: Test Environment

The generated code is well suited for real-time applications, however the size and run-time of the Java-Code can be improved. The generated C code is static and efficient. Reusing of parameterized functions for reused components keeps the code small.

We thank Bekim Bajraktari, Martin Rappl and Bernhard Schätz for many interesting discussions during the work on this paper.

## References

[AF-02] AUTOFOCUS Homepage. http://autofocus.in.tum.de, 2002.

[Baj01] Bekim Bajraktari. Modellbasiertes Requirements Tracing. Master's thesis, Technische Universität München, 2001.

[Chr00] Daimler Chrysler. The Challange: Seat specification, 2000. Internal paper.

[HMS+98] F. Huber, S. Molterer, B. Schätz, O. Slotosch, and A. Vilbig. Traffic Lights - An AutoFocus Case Study. In *1998 International Conference on Application of Concurrency to System Design*, pages 282–294. IEEE Computer Society, 1998.

[HSS96] F. Huber, B. Schätz, and K. Spies. AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme . In Ulrich Herzog Holger Hermanns, editor, *Formale Beschreibungstechniken für verteilte Systeme*, pages 165–174. Universität Erlangen-Nürnberg, 1996. Erschienen in: Arbeitsbereichte des Insituts für mathematische Maschinen und Datenverarbeitung, Bd.29, Nr. 9.

[Slo98] O. Slotosch. Quest: Overview over the Project. In D. Hutter, W. Stephan, P Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998.

[Tel02] Telelogic Homepage. http://www.telelogic.de, 2002.

[UML99] *OMG Unified Modeling language specification*. http://www.omg.org, 1999.

[Val02] Validas Homepage. http://www.validas.de, 2002.