Behavior Modeling Tools in an Architecture-Driven Development Process – From Function Models to AUTOSAR

Oliver Niggemann, Ulrich Eisemann, Michael Beine and Ulrich Kiffmeier

dSPACE GmbH

Reprinted From: In-Vehicle Software & Hardware Systems, 2007 (SP-2126)
ABSTRACT

This paper will first introduce and classify the basic principles of architecture-driven software development and will briefly sketch the presumed development process. This background information is then used to explain extensions which enable current behavior modeling and code generation tools to operate as software component generators. The generation of AUTOSAR software components using dSPACE's production code generator TargetLink is described as an example.

INTRODUCTION

The reuse of application software modules has become a major challenge to automotive software development. By reusing application modules, manufacturers and suppliers can (i) utilize software modules already tested, thus minimizing potential software hazards, (ii) cut the development workload and (iii) transfer product line approaches more easily into the world of software development. Several approaches for such reusable software modules exist, AUTOSAR being the most well-known in the field of automotive software development [1]. Most of them rely on the same principles:

SEPARATION OF CONCERNS

Different concerns are modeled separately. In an optimal model-based development process, each concern should be reflected in one special type of model. The resulting model types focus on individual aspects of the overall system model, thereby allowing better handling of complex systems. In addition, models for separate concerns may be reused more easily because in many cases only a few concerns change when there is a new scenario. For example, in some cases the electronic control unit (ECU) changes while algorithmic aspects remain constant. Typical concerns are:

- Algorithmic models: These are normally application software modules, e.g., in the form of Simulink® models or C code.
- Functional network: Such models describe the overall function of the system as a hierarchical system of connected functionalities. Each functionality may be further defined using an algorithmic model.
- Hardware platform: This concern comprises the system of connected ECUs, ECU components, and buses. It also covers the basic software, i.e., software modules such as the operating system, ECU services, I/O drivers, and communication stacks.
- Mapping: The mapping of functional networks onto hardware platforms. This normally results in a system of connected applications and basic software modules at run time. Such systems may be defined for a set of interconnected ECUs or for a single ECU.
- ECU configuration: This comprises information regarding the internal configuration of software modules on one ECU, e.g., task definitions or the basic software configuration.

While sounding rather abstract, separation of concerns has far-reaching consequences, e.g., application software must not directly use platform-dependent functions or include information such as task definitions.

SOFTWARE COMPONENTS

To be reused, software modules must be modeled and implemented independently from their environment. Such modules are called components if they (i) formalize the environmental requirements, i.e. a requirement placed on other software components and on the hardware platform, (ii) formalize the service and the signals they provide to their environment, and (iii) communicate with their environment through well-defined channels.

SYSTEM GENERATION

As a result of the separation of concerns, there needs to be a means of creating the overall system, i.e., ECU codes and bus configurations, from the separated systems descriptions. This is normally done by using a
system generator. Such a generator handles tasks such as (i) generating code for the communication between software components, (ii) generating code for the coupling between application software and ECU basic software, (iii) generating the task functions, and (iv) generating parts of the ECU configuration, e.g., task configurations and communication message definitions. Points (i), (ii), and (iii) are often called middleware generation; AUTOSAR calls its middleware generator the "Runtime Environment (RTE) Generator".

Development approaches which take all of the above principles into consideration are called architecture-driven approaches.

**PROCESS CHARACTERISTICS**

In order to discuss the implications of architecture-driven development approaches for behavior modeling tools and code generators, it is necessary to sketch some features of the development process itself. Processes must take into account a company's structure, market demands, and last but not least historical circumstances, making the notion of a generally applicable optimal process rather absurd. In this paper, an abstract process framework is used to illustrate the roles of behavior modeling tools and code generators - no claim to direct applicability in a specific company context is made.

Figure 1 depicts this process framework: Process steps are shown as rectangles which are connected with each other via well-defined outputs and inputs. Broad white arrows symbolize a refinement of models, broad gray arrows represent modularization, and small black arrows correspond to generation steps.

Process steps may use different model types, i.e., different concerns may be involved. Figure 2 roughly sketches the mapping of concerns to process steps.

Three levels of refinement are defined: (i) The logical level, (ii) the system level, and (iii) the implementation level. These can be seen as models at different levels of refinement or different views of one consistent overall model.

![Figure 2: Mapping of concerns to process steps](image)

Modularization is also performed for each of these levels: Top-level artifacts are defined in terms of aggregations of further artifacts. Such modularization is done to support the distribution of work between both companies and individual developers. Modularization is also a major means of handling complex systems. Figure 1 shows only two layers of modularization. Normally, modularization comprises several such layers.

**LOGICAL LEVEL**

The logical level comprises all the functional aspects of the system. Effects caused by the underlying hardware platform are not taken into consideration. On this level models are often simulated in non-real-time scenarios on a PC, i.e., using offline simulation.

![Figure 1: The development process](image)

Typical concerns for this level are algorithmic models and functional networks. Figure 3 shows an example model: Here a software component methodology is used to model functionalities. Two top-level software components are connected. The right component is further modularized. These four components correspond to the “functional network” concern. The left component is defined using an algorithmic model. Functional models can be defined using Simulink/TargetLink or C code.
The design on the logical level is modularized, i.e., the structure of the functionalities is defined at finer and finer levels of detail, finally leading to atomic functionalities. The behavior of such atomic functionalities is then defined by algorithmic models, e.g., using Simulink/TargetLink or C code. If code is generated from a model on this level, the resulting code is platform-independent.

SYSTEM LEVEL

Models on the system level are used to connect the logical model to a specific hardware platform. The typical concerns of such a model are the hardware platform and mapping. Of course, since this level is a refinement of the logical level, the algorithmic model and functional network concerns are also used on this level.

In many cases the functional now uses additional information. Often interfaces between functionalities are defined in more detail here, e.g., by adding signals and their respective types to interfaces which were defined earlier on the logical level using simple names only, or further levels of modularization are introduced. Sometimes this detailed functional network is now called a (logical) software architecture.\(^1\)

This modularization and the algorithmic models may be modified - compared to the logical level - to reflect their mapping onto the hardware platform. Typical examples are specific preprocessing functionalities (e.g., mapped to a DSP) or functionalities which compute values not directly available as sensor signals on the respective ECU.

The hardware platform is defined rather coarsely on the system level. It mainly comprises the following information:

- **ECU description:** The user normally defines which ECUs exist and what their main purpose should be. A detailed description of the hardware is not necessary on this level. The processing units and their classification (CPU, DSP, etc.) are mostly specified.
- **I/O:** The actuators and sensors, i.e., the I/O, are decided for a specific ECU.
- **Bus description:** Buses play an important role on the system level, since their configurations, i.e., the communication matrices, are currently being used to synchronize between manufacturers and suppliers. Besides the bus configuration, the system level mainly comprises the association of ECUs to buses and the bus topology.
- **Basic SW:** Normally, only the top-level basic software modules, i.e., the basic software modules with a direct connection to applications, are modeled on this level.

The functional network is mapped onto this hardware platform. The mapping includes:

- **ECU mapping:** Functionalities are mapped onto ECUs.
- **I/O connection:** Open ports of functionalities have to be connected to I/O.
- **Communication mapping:** Communication between functionalities has to be mapped onto communication channels. For example, data exchanged between software components on different ECUs may be mapped onto specific signals in CAN frames.

Such a system model is sketched in Figure 4: The components from Figure 3 are now mapped onto two ECUs. The components on the left ECU are connected to an I/O driver.

Since the hardware is not defined in detail, the resulting C code may still be platform-independent. The resulting software system is in many cases executed on prototyping platforms, e.g., to verify the behavior with hardware-in-the-loop simulators or to run initial tests in vehicles.

A different way to look at the refinement step (bold, white arrows in Figure 1) is from a verification point of view. The refined model, in this case the system level models, can be tested and verified using the models on the coarser level, here the logical level, as a reference model.

---

\(^1\) The term is not used here because it is not clear whether e.g. basic software is also part of this software architecture.
IMPLEMENTATION LEVEL

The purpose of this level is to serve as a basis for implementing the entire ECU production code. In many cases this level is employed by supplier companies. The “functional network”, “hardware platform” and "algorithmic model" concerns are enriched with further details and the new concern “ECU configuration” is defined on this level.

Functional networks must now contain all the information necessary to integrate implemented functionalities. That is, software components implemented by different developers or companies are integrated into the previously planned overall functional network at this stage. Thus, functionalities are augmented with information such as:

- All requirements for the context of the functionality, e.g., other functionalities or requirements for basic software, must be formalized.
- All data and services offered by the functionality to its environment must be formalized.
- The implementation, e.g., the C code, must be described.
- In many cases the resource consumption, e.g., ROM and processing power required, is also specified.
- There must be a description of how the functionality, or parts of it, can be mapped onto tasks.
- The necessary calibration parameters and available measurement variables are defined.

Functionalities which contain all this information are often called software components. As mentioned before, the defining feature of software components is that they can be reused in different settings and environments.

The hardware platform model is extended by (i) the precise structure of the hardware, e.g., by descriptions of the ROM, NV-RAM, RAM, processor, and I/O, and (ii) by the detailed description of all basic software modules.

In addition, the ECU configuration is defined. This comprises the configuration of all software modules on one ECU. Examples are:

- Definition of tasks
- Mapping of software components - or parts of them - onto tasks
- Definition of counters, alarms, and messages
- The configuration of the communication stack, e.g., I-PDUs and the bus controller configuration.
- The configuration of I/O drivers
- Values for calibration parameters
- The configuration of ECU services such as mode management, NV-RAM, or diagnostic services

Algorithmic models, again used on the lowest level of modularization, are extended by adding information such as fixed-point configurations (e.g., scaling information) or code generation settings (e.g., variable naming schemas or variable classes). For example, Simulink models are transferred to TargetLink models which include this information.

C code generated from models on the implementation level is in many cases platform-dependent, i.e., the C code is optimized for one specific ECU platform. Again, the models can be verified using the models from the system level as a reference.

Other overviews of architecture-driven development processes can be found in [2, 3], for example.

EXTENSIONS TO BEHAVIOR MODELING AND CODE GENERATION TOOLS

Most existing behavior modeling and code generation tools were developed before architecture-driven development processes were established. Extensions are necessary to integrate them into such processes.

As an example, the principles mentioned above have been applied to dSPACE’s production code generator TargetLink. TargetLink is seamlessly integrated into MATLAB®/Simulink®/Stateflow® and provides reliable conversion of software designs available as Simulink/Stateflow models into highly efficient C code. More details about TargetLink can be found in [4, 5].

TargetLink has been enabled to generate software components according to the AUTOSAR specification. The extensions to TargetLink reflect the fact that TargetLink is used on the implementation level, and on the deepest layer of modularization on that level. That is, in Figure 2 TargetLink covers the “algorithmic model” concern for the “functional implementation design” step on the implementation level. Thus TargetLink models represent the leaves of the modularization hierarchies, and their main purpose is to generate software components which comprise C code and the corresponding AUTOSAR description files.

Additional modeling guidelines may apply at the logical or system level. For example, AUTOSAR has published a non-mandatory recommendation for applying AUTOSAR to Simulink models which mainly covers the logical and the system level, i.e., the recommendation was drawn up from a modeling and simulation point of view [6]. The implementation-level modeling means introduced here do not contradict the recommendation, but focus on the implementation level, i.e., they have been developed from a code generation perspective.

In the following paragraphs, the features of architecture-driven development methodology described in the
Introduction are applied to existing behavior modeling tools and code generators. The necessary adaptations of such tools to this new methodology are sketched, using TargetLink as an example.

**SUPPORTING THE SEPARATION OF CONCERNS**

Most behavior modeling and code generation tools, e.g., TargetLink, have so far mixed at least three different concerns: algorithmic models, function networks, and ECU configuration. Since such tools now focus on the “algorithmic model” concern in an architecture-driven development process, some modifications become necessary.

*Code Generators and the Algorithmic Models Concern.*

This is of course the main concern of these tools; algorithmic models can be designed, simulated, and converted to production code. Thus, TargetLink's algorithmic modeling means, i.e., a well-established blockset, remain unchanged.

*Code Generators and the Functional Networks Concern.*

Some aspects of functional networks must also be modeled in a behavior modeling tool. This mainly involves the interfaces of functionalities which must be modeled in the algorithmic model:

First of all, it is not primarily functionalities (called software components here, since we are on the implementation level) that are modeled graphically, but runnables. Runnables are those parts of software components that can be mapped onto tasks later on. That is, tasks are aggregations of runnables. All the behavior of a software component is modeled in such runnables.

The separation of behavior into tasks is necessary, since parts of one software component may be triggered with different sample times or in different modes of the vehicles, e.g., only at startup or flashing time. Figure 5 shows an example: The software component comprises three runnables which are mapped onto three different tasks - one startup task, a 10ms and a 100ms task. Two of the tasks also contain runnables from other software components.

On the logical and on the system level, graphical modeling of the software components themselves may be necessary. Users can utilize a dedicated subsystem for this. Such approaches can be used together with the runnable-oriented modeling style introduced here; they reflect different modeling necessities at different phases of the development process.

For modeling runnables, TargetLink offers a special TargetLink AUTOSAR blockset (see Figure 6) comprising the following blocks:

- A Runnable block to identify a particular subsystem in TargetLink as a runnable. Runnables may be triggered periodically or by external events. During code generation, a C function is generated for the runnable.
- Runnable Imports and Runnable Outports to associate signals in TargetLink with individual data elements of an interface. The Runnable Ports specify the data exchange and the communication mechanisms between individual runnables. TargetLink supports sender/receiver communication as well as synchronous client/server communication, both of which are specified in the AUTOSAR standard. All communication mechanisms translate into special code patterns during code generation later on.
- A Client Port block to model a call of a method - a client/server interface - e.g., a service of the underlying AUTOSAR basic software, like I/O drivers.
- Receiver and Sender Ports for software components. These are optional, and can be used to highlight the connection between signals in TargetLink and ports of a software component.

---

2 Most of these modifications are not specific to AUTOSAR - AUTOSAR is used as a concrete example - but have to do with architecture-driven software development in general.
When modeling runnables, users can now combine the regular TargetLink blockset with the TargetLink AUTOSAR blocks. The algorithmic aspects of a runnable are modeled using the regular TargetLink blocks, and the partitioning of the algorithm into separate runnables and data exchange between them is modeled by adding TargetLink AUTOSAR blocks. Figure 7 shows the design of a simple PI controller for the AUTOSAR use case. Whenever a subsystem is to become an AUTOSAR runnable during code generation, the Runnable block must be dragged into it. For communication with other runnables, the Runnable Inports and Outports have to be inserted to receive and send input and output signals. This reflects the AUTOSAR principle that all communication between runnables is routed through a standardized interface. The algorithmic part of the PI controller, however, is modeled just as in the non-AUTOSAR use case.

The AUTOSAR standard specifies that runnables are activated by events. Simulink provides a comparable concept in the form of function-call-triggered subsystems, whose execution is actuated by events from other blocks. Consequently, runnables are implemented as function-call-triggered subsystems in TargetLink (see Figure 8). If a runnable is time-triggered, it can also be implemented as an ordinary subsystem. A software component can consist of an arbitrary number of runnables executed in a user-defined order. Signal lines specify which data items are exchanged between the individual runnables, and intra-component communication results in particularly efficient code using interrunnable variables and per-instance memory.

Since the modeling of runnables follows a proven workflow, existing models can be easily migrated to AUTOSAR. Users must primarily split the designed functionality into runnable entities in the same way as they would partition it into subsystems or functions. The newly introduced Runnable Ports can also be regarded as the natural adaptation of existing TargetLink ports to the AUTOSAR use case.

Alongside the design process on the block level, TargetLink models are always associated with a data dictionary file. It serves as a central data container for numerous code generation specifications, like data types, scaling formulas, variables etc., defining how individual signals appear in the generated code. Consequently, the data dictionary has also been adapted for the administration of AUTOSAR properties. Elements like runnables, ports, interfaces etc. require a set of associated attributes, which are specified by the AUTOSAR standard. Interfaces, for instance, must specify the data types of their enclosed data elements, while runnables must define the events by which they are activated. These attributes are administered in the data dictionary and referenced at the block level. This is how the appropriate AUTOSAR specifications are assigned to the individual blocks.

Figure 6: Library with special AUTOSAR blocks for use with TargetLink

Figure 7: Designing a PI controller for the AUTOSAR use case by combining items such as Gain, Sum, and Unit Delay blocks with Runnable and Runnable Import and Outport blocks.

Figure 8: Runnables are implemented as function-call-triggered subsystems and activated by events.
For the purpose of simulation, the runnables are included in a special simulation frame realized by a TargetLink subsystem; see Figure 9 for a potential modeling style. The runnables contained in the TargetLink subsystem are activated by events from the Stateflow chart residing outside the software component. Consequently, the latter is used only for simulation purposes and has no bearing on subsequent code generation steps or the software component. This modeling style supports simulations performed in different modes like model-in-the-loop (MIL), software-in-the-loop (SIL) and processor-in-the-loop (PIL). While the MIL mode involves simulation of the controller model based entirely on blocks, the SIL and PIL modes support the simulation of the generated controller code, executed either on the host (SIL) or on an evaluation board containing the target processor (PIL).

In many cases, users currently develop TargetLink models which have - to use the corresponding AUTOSAR notion - several connected software components. Thus, a part of the functional network was also modeled in the form of connected TargetLink subsystems - in TargetLink.

![Figure 9: Top hierarchy of a model for simulating runnables and entire components. Runnables are included in the TargetLink controller subsystem and activated by events emitted from the Stateflow chart called Chart.](image)

**Code Generators and the ECU Configuration Concern.**

In most behavior modeling and code generation tools, e.g., in TargetLink, users can specify multirate systems, operating system API accesses, and tasks. Task definitions, operating system calls and operating system configuration files such as OSEK's OIL files are created from this information. For an architecture-driven approach, this information must be specified differently: (i) Runnables replace tasks, (ii) Runnable Inports and Outports replace OSEK message definitions. Since the context of a component's usage, i.e., the ECU used and the components connected to the component under development, may be unknown during component development, configuration information such as task definitions or message definitions must be defined externally.

**SUPPORTING SOFTWARE COMPONENTS**

Most of the extensions described above already enable the user to model software components using a behavior modeling tool. However, besides the pure modeling aspect, the generated code also has to follow specific rules. Some important restrictions to code generation are:

- Incoming or outgoing signals - i.e., Runnable Inports and Outports - are modeled in the code using standardized macros. So instead of referring directly to connected software components, which would make the component such that it cannot be reused, these macros refer to the ports of the component. In that sense ports are proxies for the components that will be connected to the respective ports later on. Figure 10 shows a small example: The "controller_runable" runnable reads one variable using the abstract macro `Rte_Read_Controller_RequiredSignals_ref` and writes the result using the macro `Rte_Write_Controller_ProvidedSignal_upi`. The definitions of these macros are generated later on during the system generation phase. AUTOSAR standardizes the syntax of these macros.

- Other macros are generated for calling methods of other components.

- Communication between the runnables of one software component is handled specially using so-called interrunnable variables, as in the macro `Rte_IrvRead_controller_PosController_LinPos`.

- Runnables become functions in the generated code.

```c
void controller_runable(void)
{
    /* call of function:
       controller/Controller_Runnable/RteApiFunction
     */
    Rte_Read_Controller_RequiredSignals_ref(&ref);
    /* Sum: controller/Controller_Runnable/e1 */
    S12_e1 = ((UInt16) ref) - ((UInt16) Rte_IrvRead_controller_PosController_LinPos());
    /* Sum: controller/Controller_Runnable/si1 */
    S12_si1 = ((Int16) Rte_IrvRead_controller_PosController_LinPos());
    /* call of function:
       controller/Controller_Runnable/RteApiFunction1 */
    Rte_Write_Controller_ProvidedSignal_upi((S12_sPI1));
    ....
}
```

Figure 10: Code fragment

---

3 This is necessary to make the component reusable.
All this information can be exported into the standardized AUTOSAR software component description XML format. This XML file can also include information such as runtime performance and memory consumption\(^4\).

**SUPPORTING SYSTEM GENERATION**

As mentioned before, AUTOSAR-compatible application software generated by code generators such as TargetLink must not use any direct calls to basic software layers such as I/O drivers, to other software components, or to the communication stack. Standardized middleware macros must be used instead. Later on the middleware generator replaces these calls with the appropriate platform functions.

In addition, communication between software components must be implemented according to their locations on ECUs and in tasks, i.e., the middleware generator must replace the platform-independent communication patterns used in software component code with the respective communication means such as CAN messages, inter-task messages, or global variables.

The AUTOSAR software component description mentioned before, in the form of an XML file, contains all the information necessary for middleware generation. The process is sketched in Figure 11: The software components, comprising the C code and the XML description, are used as an input for the middleware or RTE generator. This generator also needs descriptions about other parts of the system: (i) the basic software and its configuration, (ii) other components, (iii) the connections of software components to other components and to basic software, and (iv) the hardware platform. This information is provided in the system description. Using these input files, the middleware generator generates the AUTOSAR middleware, i.e. C code defining the macros and the operating system tasks.

\[^4\) This information can be acquired via TargetLink’s processor-in-the-loop simulation, i.e., using evaluation boards.

**CONCLUSION**

This paper has placed behavior modeling tools and function code generators in an architecture-driven development process. The typical features of such processes were outlined. Necessary modifications to existing tools were explained briefly using TargetLink’s AUTOSAR software component export as an example. These software modifications allow software developers to generate reusable and AUTOSAR-compliant software components using the established TargetLink technology.

**REFERENCES**

1. AUTOSAR, Internet Homepage [http://www.autosar.org](http://www.autosar.org)


CONTACT

Oliver Niggemann
Ulrich Eisemann
Michael Beine
Ulrich Kiffmeier
dSPACE GmbH
Technologiepark 25
33100 Paderborn
Germany
E-Mail: {oniggemann, ueisemann, mbeine, ukiffmeier}@dspace.de