On Model-Based Development: Decomposition and Data Abstraction in SIMULINK

Andreas Rau
Systems Engineer
STZ Softwaretechnik
73730 Esslingen, Germany
andreas.rau@stz-softwaretechnik.de

Abstract— **The automotive industry is in the process of broadly adopting a new model-based approach for embedded systems development. However, the available tools still have a number of deficiencies, particularly with respect to the modelling of large and complex systems. The article discusses the present support for decomposition and data-abstraction in SIMULINK and outlines future needs.**

Keywords— **model-based development, decomposition, data abstraction, interfaces, SIMULINK**

I. Introduction

The automotive industry is currently in the process of adopting a model-based approach for the development of embedded control systems. This approach is founded on rapid-prototyping environments that allow graphical modelling of control-systems using standard control-design block diagrams and subsequent simulation of the design for validation. Through the included simulation capability these environments offer significant advantages over plain drawing tools: In early phases, the simulation provides quick feedback about the behaviour of the model under construction. Later, the completed model can be used as an executable specification of the system to be (manually) implemented.

In the meantime, the tools have evolved further and now also support automatic production code generation from the model. With this capability, they finally bring the vision of a seamless and completely tool-supported, model-based development process within reach. However, the step from plain drawings to models that can be simulated and further on to using them to generate production code not only requires new tools but also changes to the models and the way they are built.

Simulation models were already used in the past for rapid-prototyping of important subfunctions of complex systems. The resulting models were usually neither very large nor complex and only required a minimum of different elements. Therefore, the risk of "spaghetti models" was limited and detailed documentation was often not required. Furthermore, resources were plentiful on the workstations used for simulation, so efficiency or memory usage were not much of a concern. Finally, aspects like realtime behavior, initialization and startup/shutdown procedures or system configurations (tasks, intertask communication/synchronization) were not usually modelled because they are hard to capture and simulate.

For an embedded system however, all these things matter. Therefore, the models (and modelling notations) have to be enriched with more details before they can be used to generate production-code. This particularly applies to safety-critical systems, because for such systems everything should be explicitly specified in the model. That is, no decisions should be left to (defaults in) the code generator and compiler. Since the resulting models are increasingly large and complex, they have to be well-structured right from the beginning. This article will use the SIMULINK tool-family from The MathWorks as an example to discuss the structuring of large models and the required notation.

II. SIMULINK

SIMULINK is part of the MATLAB tool family and implements a dataflow-oriented, graphical language consisting of diagrams with blocks representing data transformations and connecting lines representing data signals. This dataflow(=functional) notation lends itself very well to modelling mathematical equations and SIMULINK offers a powerful library of basic function blocks for this purpose. With this library, the characteristic differential equations of feedback control systems can very easily be modelled. On the other hand, feedforward control systems or control flow algorithms are somewhat awkward to model using the dataflow paradigm, even with the recently added control-flow subsystems for if-else, switch-case and loops. Therefore, the application domain of tools like SIMULINK is not so much arbitrary systems or algorithms but control systems and filters in general and feedback control systems in particular.

As indicated by its name, SIMULINK cannot only be used to design models, but also to simulate them. Furthermore, its capabilities can be extended by a whole family of so-called toolboxes. One such toolbox, the Realtime-Workshop (RTW), provides the capability to generate code (ANSI-C or Ada) from the simulation model. Another toolbox, STATEFLOW, allows modelling of control-flow by means of flow-diagrams and statecharts that can be integrated into SIMULINK. In combination, these tools offer considerable expressive power and the potential to innovate the way how hybrid control systems are built. They are currently used in the automotive industry for a variety of applications such as

1. simulation
2. rapid prototyping
3. executable specifications for handcoding
4. executable specifications for code generation

Each of these applications has slightly different requirements to the development process and the underlying tools. However, the tools have yet to catch up with some of these requirements and still have some shortcomings which reflect their origin in rapid-prototyping. An overview of these shortcomings is given in [Rau00]. This article covers the shortcomings with respect to hierarchical decomposition and interfaces, based on the use of SIMULINK for creating executable specifications of automotive systems for autocode generation. One way to address or work around these shortcomings are patterns. A set of patterns for STATEFLOW was presented in [BR01]. Patterns for interface modelling in SIMULINK will be described in a forthcoming article.

III. The Principle of Abstraction

Abstraction is the key principle for mastering complex problems in science and engineering. It works by removing irrelevant detail and combining lower-level concepts to more abstract, higher level ones. Abstraction can be applied both in analysis and synthesis: When analyzing a complex problem, we first use abstraction to understand and hierachically decompose it into parts. Based on our understanding, we then design an architecture of partial solutions and implement them one by one to obtain the total solution. Different levels of abstraction during analysis result in a hierarchy of subproblems which is reflected in the hierarchy of partial solutions. In classical software engineering, these parts are referred to as tasks, threads, modules/objects, subroutines/methods and statements. In model-based development, they are called models, modules and blocks.

Also data is classified using type-systems and arranged into data structures like records, lists or trees. Consequently, in [Wir71] programming is described as a creative activity consisting of "a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into datastructures" where "refinement of the description of program and data structures should proceed in parallel". The requirement to properly support hierachical decomposition for *both code and data* is essential for any language or notation intended to solve or describe problems of real world complexity. This also applies to the new model-based approach.

IV. Abstraction and Models

Abstraction can be applied to models in different ways: First, there is clustering, the view of basic elements as a functional group. An electrical engineer for example, will quickly identify functional groups (e.g. cascaded transistors implementing an amplifier) in a circuit diagram with many components. Mentally isolating such *white-box clusters* within a larger group of visible items is a powerful technique already, but an even more powerful way to reduce complexity is the aggregation of elements into *black-box components* that hide their elements and can themselves be clustered or aggregated again. Such a component is much better suited for reuse than a cluster.

A black-box implements the principle of information hiding described in [Par72] by making its contents independent from their periphery *and vice versa*. This is particularly important for large and complex models: They are usually split into many parts, each of which can be of substantial complexity. The extent to which the work on these parts can be shared between many developers, i.e. the extent to which they can be independently implemented and tested, largely depends on their interfaces.

Interfaces are created whenever something is decomposed into parts. It is one key lesson from software engineering experience that it is important to pay attention to them because of their importance for the development process and the final product:

> From a developer's perspective, an interface is all that is visible of an abstracted functionality. That is, the interface "is" the component.
>
> From a systems point of view, interfaces are the glue for putting modules together: if the glue is bad, the system will break.
>
> From a process point of view, interfaces are a contract that makes developers independent of each other, and regulates the integration of their results.

The characteristics of an interface depend on where the decomposition is made. If the point is well-chosen, complexity is reduced as intended, whereas if it is ill-chosen, complexity might even increase. A discussion of decomposition strategies for system analysis can be found in [MP86]). However, there is no universal strategy for optimal decomposition. So basically, we must rely on a number of basic principles, some rules of thumb, experience and common sense when it comes to finding the right place to cut. By now, these principles are well known and have been discussed in the literature for many years. Some interesting concepts can be found in [Par72], [Mye75], [Mey88], [LH89], [Bro95]. A summary of criteria for good interfaces was presented in [Rau01].

Decomposition and interfaces are essential for managing and reducing complexity in textual programming. This does not change as we move on to models. However, to be able to use decomposition and interfaces in this new context, the models and modelling tools have to properly support these basic concepts first. With respect to interfaces, this includes the need to be able to properly define the data exchanged through them.

V. SIMULINK Capabilities and Limitations

A. Functional Decomposition

SIMULINK supports functional decomposition via the subsystem block. This block can be used to encapsulate a group of other blocks to form a *black-box component* (see figure 1). Decomposing a system into subsystems leads to a strict tree structure. Subsystems can optionally be masked with a GUI to hide their contents and collect all parameters in one place for easier reuse. This also includes a masktype to distinguish different reusable subsystems.

There is currently no other support for modelling different kinds of components (such as tasks, modules and functions) and their properties and mapping them to different implementations. Also at the time of this writing there is no formal support for explicitly marking and describing *white-box clusters*, e.g. with dashed boxes, so informal spacing and annotations must be used instead.

Furthermore, the grouping achieved by a subsystem is only visual (or in SIMULINK terminology: "virtual"): Putting blocks inside a subsystem does not make any difference for simulation or code generation, i.e. subsystems are flattened (removed, inlined) for execution. However, subsystems can be marked as "atomic" and treated as a function. But even such atomic subsystems do not have a formally defined interface: The interface description of all subsystems is limited to a number of named inports and outports where external signals (which could be viewed as function arguments) can be attached. Both plain and

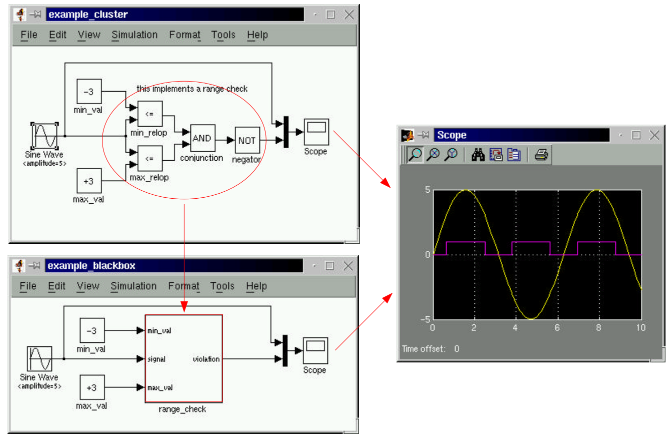

Fig. 1
AN EXAMPLE HIERACHY

atomic subsystems can be reused by putting them in an extra library file and "linking" them into a model.

For achieving a minimum of formality, a common guideline for preventing wiring errors is to ask for the port-names to match the names of the connected signals, both inside and outside of the subsystem (see figure 2). This is not required by SIMULINK and there is also no built-in check to enforce this.

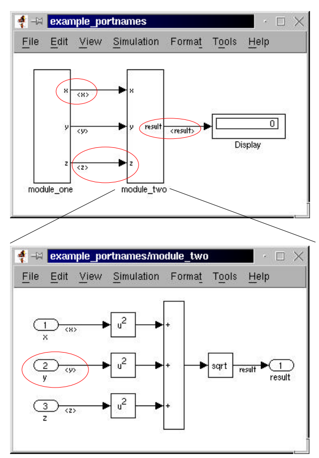

Fig. 2
AN EXAMPLE FOR MATCHING PORTS AND SIGNALS

It is possible to associate a subsystem port with a type and dimension (but not with any other information, such as upper and lower limits or a physical unit). However, this feature is rarely used in practice with the exception of top-level inputs, where the type and dimension *have* to be specified for code generation. The reasons for this are lack of time and the fact that the feature itself is limited, as we shall see below.

B. Data Abstraction

By default, all signals inside a model are of type double. This is fine for prototyping. But for optimizing production code and saving system resources, the use of appropriate datatypes is elementary. And when doing so, well documented, i.e. strongly-typed interfaces are an essential measure to prevent and detect errors and mismatches early on, especially when developing large systems. Therefore, developers should be able to enforce strict datatyping at interfaces as an option[1]. In this way, modelling for production code could be made much safer without making prototyping any more complicated.

The type-system of SIMULINK is rather limited. It consists of a boolean and elementary integer and floating point numeric types. There are no compound datatypes other than vectors, which are equivalent to arrays and can be accessed by index. This means that developers cannot define their own compound datatypes. This limitation for data-abstraction is at best cumbersome, because it requires developers to treat separately what really belongs together. In particular, at an interface it means that a compound argument cannot be implemented as such, which makes the interfaces appear wider and makes it more difficult to understand. For large systems with complex interfaces, this can be fatal.

SIMULINK offers a bus concept to address this problem. However, the implementation of this concept falls short of a complete solution. Busses can be used to group signals. They support hierachy and allow access to their elements by name. This makes them resemble structures or records. But busses are not really compound datatypes. They are virtual like subsystems and their only purpose seems to be to reduce the number of lines and avoid visual clutter in the diagram. In comparison with textual programming languages, the concept which might be closest to a bus is a dictionary (associative array).

The structure and names of signals in a bus are not formally defined. Instead, busses are built ad-hoc at their source by feeding signals into a Mux block. It is also possible to nest busses, i.e. build a bus of busses by feeding a bus into a Mux block. For accessing, they use the names of the enclosed signals, i.e. they require them to have one. Unfortunately, no error is issued if multiple signals have the same name. Instead, SIMULINK adds a suffix to make the names unique.

[1]This is already possible for the SIMULINK/STATEFLOW interface.

The absence of a formal definition means, that the expected structure of a bus cannot be explicitly defined nor enforced at its destination, i.e. it is not possible to specify an expected bus structure in an inport. This makes complete interface definitions impossible. For the same reason, busses cannot be used as top-level inputs of a model. Furthermore, this means that bus definitions cannot be reused but have to be re-created every time, leading to redundancy and maintainance problems.

So with busses, signals are only informally bundled and unbundled, i.e. moved in and out. A record by contrast, has a predefined number of fields with predefined, unique names where values are copied in and out. The lack of formality of busses is justified from a rapid-prototyping perspective, but is not adequate for generating production code.

To make things worse, while there are different blocks for the decomposition of vectors (Demux) and busses (BusSelector), both are created using the same Mux block. Whether the output of a Mux block is treated as a vector or a bus is determined automagically by SIMULINK from its use and the properties of the incoming signals (e.g. whether they have names), but the way this magic works is not always clear to the user. Furthermore, if (and only if) all signals in the bus have the same datatype vectors with named signals may be treated as busses and a bus may be treated as a vector - both intentionally and by accident. As a result, developers cannot clearly express their intention and the automagical classification cannot raise an error, if a bus is accidentally used as a vector (see figure 3). This can lead to problems with the practical use of busses. For example, a vector in the top-level (output) interface may be accidentally unbundled (as a bus) during code-generation because its elements happen to have names.

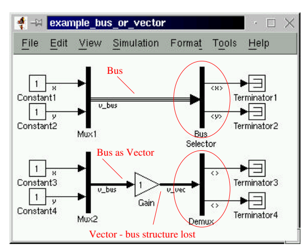

Fig. 3
AMBIGUITY BETWEEN BUS AND VECTOR

Another limitation becomes apparent when looking at built-in blocks: They are only designed for scalars and vectors, so busses must not be directly connected to them, because. For some blocks however, namely switches and delays, it would actually be desirable and make sense to attach a bus.

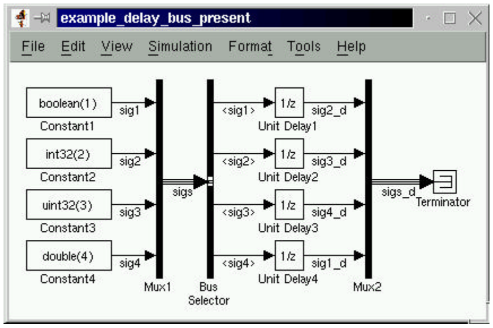

Fig. 4
DELAYING A BUS: WORKAROUND

To work around this limitation, a bus must be decomposed and its elements passed into individual blocks, as illustrated in figure 4 . This need for explicit decomposition limits the usability of busses. The array of blocks in the workaround needs a lot more room and is presumably harder to optimize by a code generator than the single block in the desired solution shown in figure 5. However, that solution does not work with SIMULINK, because the bus contains hybrid datatypes. It would seem to work for a bus with elements of equal type, because it can be treated as a vector. owever, this is no alternative to the workaround because the bus would also come out as a vector and the bus information would be lost. This can still happen by accident, though.

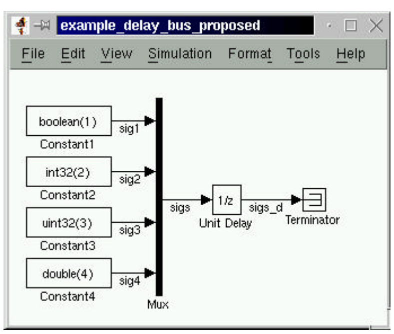

Fig. 5
DELAYING A BUS: DESIRED SOLUTION

The decomposition of the bus shown in the workaround necessarily unfolds a redundant description of the bus structure that has to be adjusted whenever the original bus changes. Forgetting to do so is particularly dangerous, when signals are added: Without adjustment, a signal that is added to the original bus will become stripped out at the point of bus rebuilding. When it is eventually reported missing at the point of its use, it is tedious to backtrack through a chain of such transformations to find the error. For removed or renamed signals, the BusSelector block used for decomposition will issue a warning.

Finally, the order of bus elements in its vector equivalent is not well-defined, especially for nested busses. As of today, SIMULINK uses the order of elements when building the bus. Since busses are accessed by name, busses with different internal orders can still be equivalent if their element signals have the same names. However, when busses are used as vectors, the order is significant.

VI. A CHRISTMAS WISH LIST

A. *Functional Decomposition*

Functional decomposition of large models requires a clear notion of interfaces as part of a functional component, so this capability should be added to SIMULINK. This could be done by extending the capibility to specify types in inports to user-defined types (see Data Abstraction below). Furthermore, it would be desirable to be able to reuse subsystems without the need for putting them in an extra library by referencing them from inside the model (just like one would call a reusable function from several places inside a program).

In addition to atomic subsystems, new semantics should be introduced to express concepts such as tasks. Please note, though, that the functional decomposition and physical decomposition of a system are not necessarily the same. Therefore, a simple “task subsystem” won’t do. However, a concept for modelling tasks and their priorities is dearly needed to achieve the goal of generating the whole system out of the model and not having to wrap the generated code in hand-coded task-stubs for execution in the target environment.

For “lightweight decomposition”, it would be helpful to be able to mark and briefly describe functional clusters in a larger diagram without having to hide them in a black box. This could be done by some kind of enclosure (e.g. dashed lines) or a “transparent black box”[2].

B. *Data Abstraction*

For proper data abstration, it should be possible to clearly distinguish between vectors and busses from the time they are built. Today, there are two separate blocks (Demux and BusSelector) to decompose them. Consequently, there should also be separate blocks to assemble them. So in addition to the Mux, a BusCreator is needed. It should not be possible Demux the output of a BusCreator or use a BusSelector on a Mux output. This would allow

[2]Although sounding strange, this is already possible in STATEFLOW

developers to choose between the two concepts according to their purpose as explained below.

Vectors should be used to group signals that only make sense together (mathematical or physical vector) or have the same meaning (array) and for which the use of mathematical vector operations or element-wise operations make sense as a group, e.g. $v \quad v_x\ v_y\ v_z$ (velocity), $n \quad n_{fl}\ n_{fr}\ n_{rl}\ n_{rr}$ (wheel rpms front left through rear right). Vector elements always have the same datatype and very often also the same physical unit. They may or may not share other properties like range (min/max), ...

Busses should be used to group signals that have something in common, but also make sense on their own. Mathematical vector operations and element-wise operations typically do not make sense for busses as a whole. Also, bus elements generally do not have a common datatype or unit. Examples for common properties are same origin, same destination, properties of the same component[3], ...

Furthermore, it should be possible to define the structure of a bus as an abstract type and optionally attach this type to BusCreators and BusSelectors to enforce proper composition and decomposition of busses as well as inports and outports to define interfaces. Such an optional type-binding would allow fortifying code generation models without complicating the creation of prototype models.

Binding model elements to abstract types would also be useful for code generation. For example, each abstract type could be mapped to one or more different physical implementations, e.g. union/struct, to optimize the code and simplify interfacing with legacy code. At interfaces, this could include the decision about the data-passing mechanism, e.g. by value, reference or global variable.

SIMULINK users may argue, that a BusCreator block is already available in SIMULINK 4.x, however it is little more than a masked Mux. Instead of replacing them by BusCreators during `slupdate`[4], SIMULINK still allows the creation of busses using the Mux block, and the output of a BusCreator can nicely be DeMuxed. So the concepts are still not clearly separated although different linestyles at least show what SIMULINK thinks a line represents. Furthermore, the new BusCreators can enforce a predefined bus structure, but unfortunately this definition can not be reused by referencing it in other BusCreators or from Inports. Thus, the problem with interface definitions remains unsolved.

[3]True records would actually be more suitable for this

[4]A command to convert models to new versions of SIMULINK

VII. Conclusion

This paper has motivated the need for hierarchical decomposition and data abstraction for the model-based development of complex automotive control systems and has given an overview of SIMULINK's current capabilities. Some deficiencies were identified and addressed by a wish list of future capabilities. These capabilities have also been discussed with the vendor of tool, The MathWorks, and some of them may be implemented in one of the forthcoming releases.

References

[BR01] Daniel Buck and Andreas Rau. On Modelling Guidelines: Flowchart Patterns for STATEFLOW. *Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends*, 21(2):7–12, August 2001. http://pi.informatik.uni-siegen.de/stt.

[Bro95] Frederick P. Brooks. *The mythical man month: essays on software engineering*. Addison Wesley Publishing Company, Inc., 20th anniversary edition, 1995.

[LH89] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[MP86] Stephen M. McMenamin and John F. Palmer. *Essential Systems Analysis*. Prentice Hall, 1986.

[Mye75] Glenford J. Myers. *Reliable Software Through Composite Design*. Van Nostrand Reinhold Company, New York, 1975.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[Rau00] Andreas Rau. Potential and challenges for model-based development in the automotive industry. *Business Briefing: Global Automotive Manufacturing and Technology*, pages 124–138, September 2000.

[Rau01] Andreas Rau. Decomposition and interfaces revisited. *Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends*, 21(2):19–23, August 2001. http://pi.informatik.uni-siegen.de/stt.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

About the Author

Andreas Rau received his degree in Computer Science from the Fachhochschule für Technik (University of Applied Sciences) in Esslingen (Germany) in 1995 and spent 5 years working for the Steinbeis-Transferzentrum Softwaretechnik in object-oriented software development projects at Alcatel SEL and Deutsche Telekom. He is currently doing research for his Ph.D. in Computer Science with Prof. W. Rosenstiel from the University of Tübingen while working with the Control System Design (CSD) team in advanced development at DaimlerChrysler, Sindelfingen (Germany). He is a member of the German Gesellschaft für Informatik (GI) and the IEEE Computer Society.