# On Model-Based Development: A Pattern for strong Interfaces 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. This article briefly recollects the shortcomings of SIMULINK and presents a solution in the form of a generic pattern.**

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

## I. Introduction

Model-based development of embedded control systems has been evaluated in the automotive industry for a couple of years now. This was driven by the hope that the use of interactive graphical modelling environments with the capability for simulation and code generation could lead to a seamless and completely model-based development process. However, the steps from textual programming to models that can be simulated and further on to using them to generate production code not only require new tools but also changes to the models and the way they are built. Interfaces play an important role in this context. This article will use the SIMULINK tool-family from The MathWorks as an example to discuss the needs and possiblitities for implementing interfaces in graphical models.

## 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. The 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.

However, the tools still have some shortcomings which reflect their origin in rapid-prototyping. An overview of these shortcomings can be found in [Rau00]. One way to address or work-around these shortcomings are patterns. To this end, this article presents a pattern for modelling strong interfaces in SIMULINK. A set of patterns for STATEFLOW was presented in [BR01].

## III. SIMULINK Capabilities and Limitations

As discussed in a previous article by the author, SIMULINK offers some limited support for functional decomposition through the concept of subsystems. However, these subsystems are mostly a graphical convenience and have no heavyweight semantics attached. Most importantly, the interface of a subsystem merely consists of a number of signals and is not usually strictly defined in terms of datastructure or datatype. The possibility to specify the types of subsystem inputs by assigning a type to each so-called inport is limited to the built-in types offered by SIMULINK. No type can be assigned to the so-called outports of a subsystem to specify its outputs.

Furthermore, there is no concept of data-abstraction other than vectors (arrays) and no user-defined compound datatypes. The bus concept of SIMULINK resembles a structured type, but does not allow an abstract definition or its reuse in different contexts. The compatibility of datatypes is decided by layout and not by semantics, i.e. from SIMULINK's point of view it would be perfectly okay to add a four element vector of rpms to a four element vector of wheel velocities.

These limitiations make it difficult to define interfaces in SIMULINK. However, as pointed out in [Rau01a], interfaces are important for several reasons, especially for large systems. An interface describes the inputs and outputs of a module and is an integral part of its implementation. But when a subsystem is driven by a bus in SIMULINK, the structure of this bus cannot be determined by looking inside the module, where its elements might be used in many different places, but outside the module, at the place where it is built. So the minimum requirement for the pattern that is presented in the next section was to allow the complete specification of a subsystem interface *as part of this subsystem*. For a strong interface, adherence to this specification has to be *enforced*, so this was an additional requirement. Please note, that in absence of dynamic-binding, the type and structure of data are static properties and can be checked without a penalty for runtime performance. The final requirement for the pattern was to facilitate the creation and enforcing of *minimal* interfaces. This is necessary because it is very common in SIMULATION models to pass huge busses between subsystems, with only a fraction of their elements being actually used. This violates the principle of information-hiding and can lead to all sorts of problems when the bus structure changes.

## IV. A Pattern for SIMULINK Interfaces

### A. A simple example program

The limitations for interface modelling in SIMULINK can be overcome through a combination of existing capabilities in a pattern. This pattern is based on an analogy between textual programming and SIMULINK models. While such an analogy can provide many valuable insights, its in-depth discussion is beyond the scope of this paper. The analogy can be illustrated by a simple example shown in figure 1 (see next page).

The example shows a generic C-language function `compute_velocity` with an explicit interface definition consisting of typed parameters `omega`, `radius` and a typed return value `v`. It expects its parameters to be in standard SI-units. When the function is used in `main`, the wheel properties first have to be converted to the right units. Please note, that in the example, the specification of the units is a comment and the conversion is implemented manually. In an advanced modelling environment like SIMULINK, this could be supported by a datadictionary for the signal and automatic conversions.

As shown in figure 2 (also next page), the C-compiler implicitly substitutes variables from the local context for the formal parameters, calls the function and assigns the return value to a variable in the calling context.

Please note, that the conversion to the necessary type is done implicitly by the compiler. For good style, explicit type-conversions are preferrable and a programming language should support it by enforcing explicit type-conversion. This should at least be available as an option.

### B. Applying the Pattern

Using the pattern, a function is represented by a subsystem with a single input and output bus representing the arguments list and return values, respectively. Following this notion, the `compute_velocity` function and its invocation are modelled with the following three blocks:

*INMAP* (left)
  prepares the "call" of the module
*MODULE* (middle)
  implements the module itself
*OUTMAP* (right)
  handles the "return" of the results.

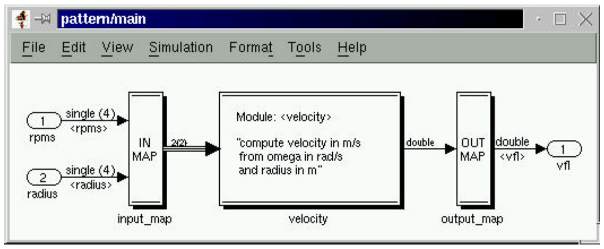


Fig. 3
Interface Pattern, Top-Level

Figure 3 shows the top-level of the translation of the example program into SIMULINK with the pattern. It corresponds to the function `main`. Please note, how the array of structs from the example program is modelled as two separate vectors, because SIMULINK does not support structs.

### C. The INMAP-Block

In the textual example, we saw the assignment of the actual arguments to the formal function arguments, including implicit type conversions. As shown in figure 4, this is exactly what the INMAP block does. It (a) selects the arguments from signals or busses in the calling context,

```
/* compute velocity in m/s from omega in rad/s and radius in m */
double compute_velocity(double omega, double radius) {
  double v;
  v = omega * radius;
  return v;
}

/* compound type for wheel properties */
typedef struct {
  single rpm;           /* in 1/min */
  single radius;        /* in cm    */
  ...
} wheel_t;

int main (void) {
  wheel_t w[4]; /* wheels from front_left thru rear_right */
  single vfl;   /* velocity in km/h                       */
  ...
  vfl = 3.6 * compute_velocity(w[0].rpm * 2 * pi / 60, w[0].radius / 100);
  ...
}
```

Fig. 1
EXAMPLE PROGRAM

```
omega = (double) w[0].rpm * 2*pi / 60;      /* evaluate and assign arguments */
radius = (double) w[0].radius / 100;        /* to prepare the function-call  */
call compute_velocity;                      /* perform the function-call     */
v = result;                                 /* assign return value           */
```

Fig. 2
COMPILER ACTIONS

(b) explicitly performs required datatype conversions, (c) renames the arguments to match the formal argument names, (d) builds a new bus to be used as an argument-list and (e) may be used to perform other kinds of conversions, e.g. unit conversion, to match the definition of the input interface.

The function of the INMAP is only adaptive. It should **not** be used to perform any other kind of calculation. The INMAP-block *collects* only the required signals and implements the *export* of data from the calling context to the module. By doing so, it minimizes the coupling between the module and its context. In practice, this is the most important function of the INMAP-block.

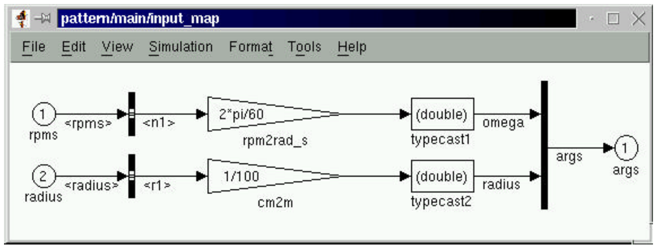


Fig. 4
INSIDE THE INMAP-BLOCK

The conversions can usually be avoided by implementing a signal properly at its source. They are mostly needed for external signals and to reuse generic modules. Likewise, the renaming of arguments when building the bus for the "argument-list" is actually an option and is mainly used with modules that are reused in different contexts, e.g. filters. System-specific modules that are only used in one place will typically receive their input signals with their names unchanged.

### *D. The OUTMAP-Block*

The OUTMAP-block is the INMAP-block's equivalent for the return-end of the module. As shown in figure 5, it assigns the return values to signals in the calling context. Precisely speaking, the purpose of the OUTMAP-block is to (a) decompose the return bus, (b) select the results relevant for the calling context, (c) explicitly implement datatype conversions and (d) assign the results to context signals, possibly building new busses.

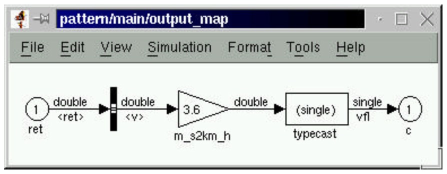


Fig. 5
INSIDE THE OUTMAP-BLOCK

Like the INMAP-block, the OUTMAP block is part of the module's context. It implements the *import* of results from the module and *distributes* them in the calling context. By hiding the necessary adaptions, the INMAP-block and OUTMAP-block support abstraction in the higher-level diagram. This is equivalent to the hiding of the MODULE details in the MODULE-block. To show their purpose on the higher-level diagram, both MAP-blocks and the MODULE-blocks have descriptive masks.

### *E. The MODULE-Block*

Even though the INMAP-block builds the argument list, it does not contain the actual definition of the input interface. It must not, because the definition of the input-interface belongs to the module itself, but the INMAP-block is part of the context (it will look differently every time the module is used in another context). Therefore, the characteristic interface properties of the module are defined inside of it, using the three blocks shown in figure 6:

*INPUTS* (left)
  defines the input-interface of the module
*RETMAP* (middle)
  defines the results of the module
*OUTPUTS* (right)
  defines the output-interface of the module

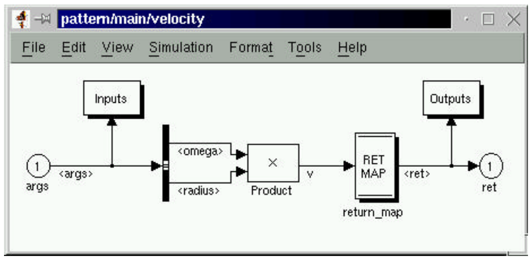


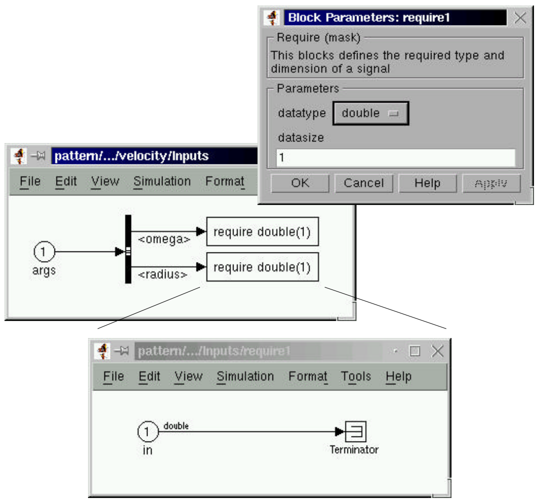
Fig. 6
INSIDE THE MODULE-BLOCK

All these blocks are masked subsystems constructed from standard SIMULINK blocks. A complete interface documentation contains at least the name, datatype and dimension of all signals and their arrangement. As discussed in the previous section, the name, datatype and dimension of a signal can be specified in an inport[1], but it is not possible to specify the structure of a bus there. However, unfolding the structure of the arguments-bus inside the INPUTS-block as shown in figure 7 achieves the same effect. For a nested bus, this has to be done one level at a time through a cascade of BusSelectors or else the structure of the intermediate levels will not be fully captured. As in most practical cases, the bus in the example is flat. Other than in the example however, nested busses with more than 50 signals are not uncommon for interfaces of real-world top-level modules. Quite often, these busses carry a lot of extra signals that violate the principle of keeping interfaces minimal. This is where the INMAP-block comes in again.

Eventually, every signal in the bus is connected to a special REQUIRE-block. The REQUIRE-block is a simple masked subsystem that is used to specify the datatype and dimension of a scalar or vector signal. This is done by writing the values entered into the mask shown in figure 7 into the inport underneath the mask which will enforce adherence to this specification.

By completely unfolding the bus structure and filling in the REQUIRE-blocks, we have created a separate, full, strongly-typed and explicit specification of the input-interface. This can be done long before the module is actually implemented. Before, the input interface of a module was only implicit and scattered across several ports and BusSelectors in the actual implementation of the function-

[1]The name is not enforced


Block Parameters: require1
Require (mask)
Parameters
datatype
double
datasize
1
OK
Cancel
Help
Apply
pattern/.../velocity/Inputs
File Edit View Simulation Format
args
<omega>
<radius>
require double(1)
require double(1)
pattern/.../Inputs/require1
File Edit View Simulation Format Tools Help
double
in
Terminator

This blocks defines the required type and dimension of a signal

Fig. 7
INSIDE THE INPUTS-BLOCK

ality. Its graphical representation in the pattern is intuitive and human-readable, but it takes some work to build it. However, the specification of the interface is not only for reading, but will also be enforced by the elements used for building it:

the BusSelectors will complain if the unfolded bus has the wrong structure, i.e. if nodes or signals are missing or have the wrong name.
the inports underneath the REQUIRE-blocks will complain if the signals have the wrong datatype or dimension.

These are both static checks that take place at "compile-time" of the model and do not harm performance. Note though, that the BusSelectors will *not* complain about extra signals in the bus! It is the job of the INMAP-block to select only the required signals. The right-hand side of the INMAP-block must reflect the input-interface.

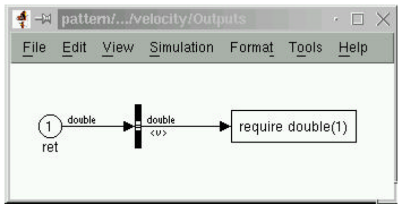

pattern/.../velocity/Outputs
File Edit View Simulation Format Tools Help
1
ret
double
double
<v>
require double(1)

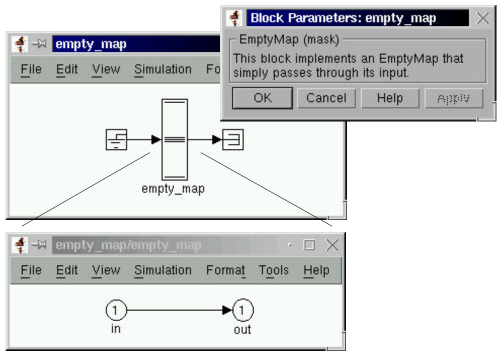
Fig. 8
INSIDE THE OUTPUTS-BLOCK

The outputs of the module are specified inside the OUTPUTS-block in the same way as the inputs in the INPUTS-block (see figure 8). But before that, they must be *collected* by the RETMAP-block which is the OUTMAP's peer inside the module and implements the *export* of results from the module. By doing so, the RETMAP-block hides internal details (signals) of the module from its context. Please note the subtle difference between the RETMAP-block and the OUTPUTS-block: The RETMAP-block collects the results to be returned from the module. It corresponds to the return statement in the C-example. The OUTPUTS-block specifies the arrangement and properties of the results. It corresponds to the declaration of the return type in the C-example.

Since the RETMAP-block has to build the return bus the way it is specified, one might consider merging the two blocks into one. The reason while they are kept separate is, that like all MAP-blocks the RETMAP-block actually contributes to the functionality of the model, whereas both the INPUTS- and OUTPUTS-block contain only static specifications and could be stripped out or disabled before code-generation[2].

## F. Special blocks

The OUTMAP-block in the example is very simple. In practice, the OUTMAP-block will sometimes even be empty. This is possible also for INMAPs and RETMAPs, but much less frequently. For both cases, there is a special EMPTYMAP-block (see figure 9). Instead of leaving the map block out if it is not needed, this block can be used to distinguish between a purposely empty map and an accidentally forgotten one.


Block Parameters: empty_map
EmptyMap (mask)
OK
Cancel
Help
Apply
empty_map
File Edit View Simulation Fo
empty_map
empty_map/empty_map
File Edit View Simulation Format Tools Help
1
in
1
out

This block implements an EmptyMap that simply passes through its input.

Fig. 9
THE EMPTYMAP-BLOCK

Please note, that renaming in a dataflow-environment is equivalent to assignment in textual programming. In MAP-blocks, this usually takes place directly behind a built-in block, such as a gain or datatype-conversion. Even

[2]However, they are virtual and will not generate any code anyway.

with an EMPTYMAP-block, we can rename the result bus as it comes out. When no other built-in block is needed, we could use a gain(1) to rename signals after a BusSelector. However, using a specialized block for this purpose is much more readable. The RENAME-block shown in figure 10 is such a block. It is yet another masked subsystem block consisting of an empty subsystem to break the line on the current level and be able to give the signal a new name as it comes out. In contrast to the frequently used gain(1), this will also preserve the bus structure.

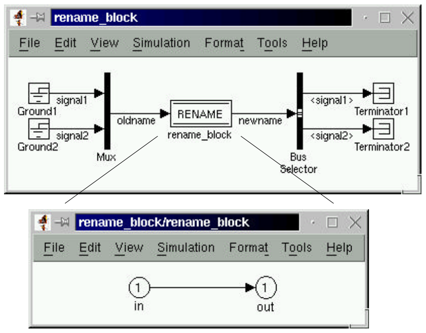


Fig. 10
THE RENAME-BLOCK

## G. Possible Extensions

The basic pattern presented so far satisfies the minimum requirements for an interface specification. It could be extended by a datadictionary to specify additional signal properties, such as their physical unit, as well as additional conditions for the inputs and outputs, such as their range or relation to each other. Doing so would closely resemble the "design-by-contract" approach described in [Mey92]. The additional conditions could be placed either in the INPUTS/OUTPUTS-blocks themselves or in two new parallel PRECONDITIONS/POSTCONDITIONS-blocks (see figure 11) and could be used for diagnostics during development or watchdogs during runtime. For the latter, the PRECONDITIONS/POSTCONDITIONS blocks would need an outport for the results.

Please note, though, that datatype and size can be enforced by a static check while other conditions might require dynamic checks at runtime and thus require extra performance. Therefore, it would be desirable to be able to selectively turn them on and off for diagnostics-on-demand or to minimize the overhead in the generated code without having to remove the checks from the model. A set of custom blocks to implement such optional condition checks is described in [Rau01b]. When pure SIMULINK blocks are used to implement the conditions, they must

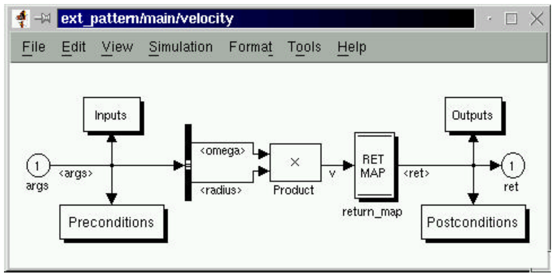


Fig. 11
THE EXTENSION OF THE PATTERN

be put in an extra subsystem (PRECONDITION-block) to switch them on and off with SIMULINK's enable feature or automatically strip them before production codegeneration.

## H. A final example

The interface specifications that are part of the pattern are also helpful for reuse. The `compute_velocity` function used in the example could also be a generic function. Such functions could be placed in libraries and should have their own masktype to distinguish them from other blocks. However in addition to the technical specification of the interface, a semantic documentation (i.e. for signal units) is still needed. A concept to integrate a datadictionary with this kind of information is currently being used in several projects and has been filed for patent ([Rau]). It may be presented in a future article.

Figures 12 and 13 show a larger example implemented with the pattern and with plain subsystems. Both models compute the velocities of the two front wheels of a car and calculate their average.

As figures 14 and 15 show, only the MAP-blocks have to be adjusted to reuse the `compute_velocity` function. The `average` function only requires name adaption and thus has a simple INMAP with two RENAME-blocks (figure 16) and an EMPTYMAP at its output. The function itself is very simple. Inside of INPUTS and OUTPUTS, REQUIRE-blocks are used to specify the individual signals as in the previous example.

The plain version uses less blocks than the pattern, but lacks abstraction and explicit interface specifications. In this simple example, the inports could actually be used for this purpose (the example could even be completely flattened). However, the typical use of this pattern is for large subsystems with many signals. They mainly appear at the upper levels of a model, where testing and the sharing of work usually take place. On lower levels, subsys-

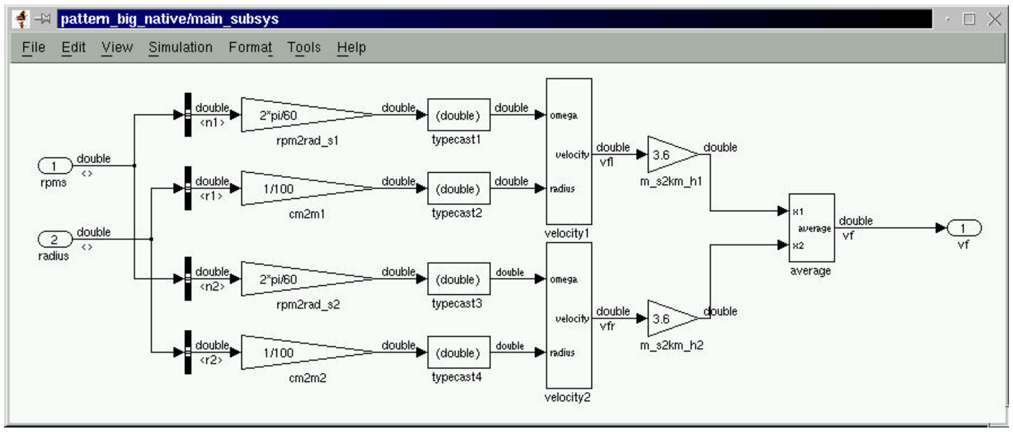


Fig. 12
A LARGER EXAMPLE (CONVENTIONAL SUBSYSTEMS)

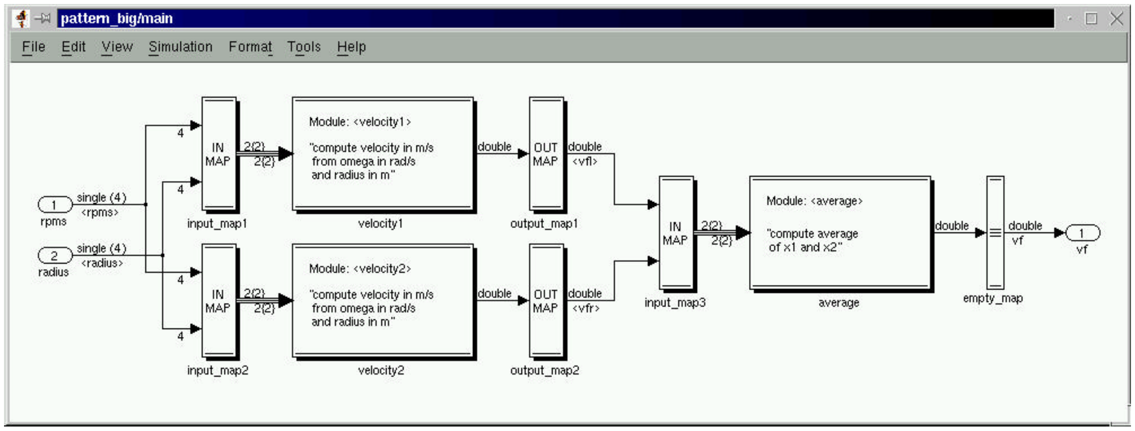


Fig. 13
A LARGER EXAMPLE (PROPOSED PATTERN)

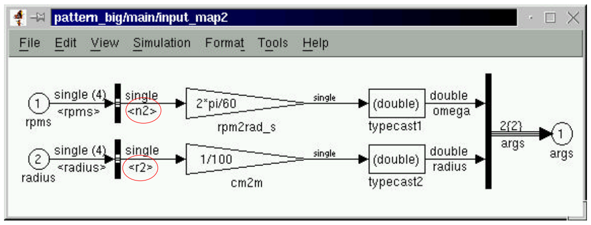


Fig. 14
THE ADJUSTED INMAP-BLOCK

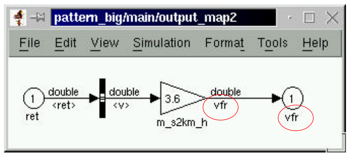

pattern_big/main/output_map2
File Edit View Simulation Format Tools Help
ret
<v>
3.6
m_s2km_h
vfr

Fig. 15
THE ADJUSTED OUTMAP-BLOCK

tems become smaller and eventually too small to justify the effort for the pattern. But since they typically only use a few signals, using individual ports is often sufficient for describing them. Unfortunately, SIMULINK does not provide different kinds of subsystems to reflect these concepts. Therefore, we use the terms “module” and “cluster” to refer to subsystems with full interfaces and normal subsystems, respectively, and use the term “subsystem” only to refer to the underlying SIMULINK concept.

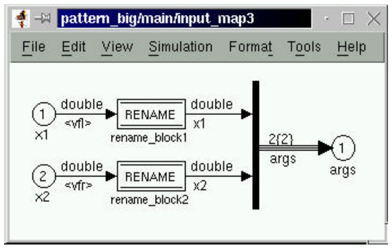


Fig. 16
THE INMAP FOR THE AVERAGE FUNCTION

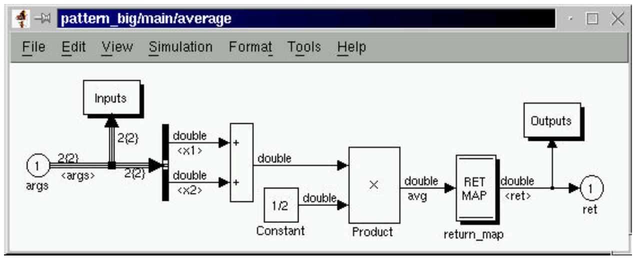


Fig. 17
INSIDE THE AVERAGE FUNCTION

## V. CONCLUSION

This paper presented a pattern for explicitly modelling and enforcing strongly-typed interfaces in SIMULINK, using only built-in capabilities. By using this pattern, it is possible to fully specify and enforce strong interfaces as part of a subsystem and early in the development process without disturbing the implementation of the module or causing runtime penalties.

The different MAP-blocks used for adapting to these interfaces facilitate information hiding and minimal interfaces. Furthermore, they support reuse by providing a single place of adaption to different contexts. Together, this leads to interfaces that are stable, well-documented and adhered to. In addition, the new special blocks introduced with this pattern provide more detail, clarity and a higher level of abstraction for SIMULINK models.

However, it takes extra work to follow the pattern. Even though this effort pays in the long run, eventually the tool should provide the means to specify signals and interfaces and check them without having to add extra blocks.

The pattern shown in this paper is currently being successfully used in projects in the automotive industry. Still, it is but one example of the insights to be gained from comparing models to textual programs as we advance towards using them for implementing software. Future research of this analogy is therefore necessary.

## 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.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[Rau] Andreas Rau. Verfahren zur Entwicklung einer technischen Komponente. Patent Pending, filed 29.06.2001 by DaimlerChrysler, German Patent and Trade Mark Office, reference no. 101 314 438.8-53.

[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.

[Rau01a] 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.

[Rau01b] Andreas Rau. Zusicherungen und Laufzeit-Überwachungen in der modellbasierten Entwicklung (assertions and watchdogs in model-based development). In Klaus Panreck and Frank Dörrscheidt, editors, *Simulationstechnik - 15. Symposium in Paderborn (ASIM2001)*, 2001.

## 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.