Models *of elegance*

Domain specific modelling promises fast software with fewer errors. By **Duncan Leslie**.

A method of developing software that increases the reliability of code and the speed at which it is written is gaining ground in automotive system development.

Domain specific modelling (DSM) uses a graphical programming language to represent the various facets of a system. It can include automatic code generation, which means faster creation of source code than manual coding and a significant reduction in defects in the code. Because the languages used support higher level abstractions, rather than general purpose modelling languages, it requires less effort and fewer low level details to specify a given system.

Whilst code generation has been used in the past – notably in the CASE tools of the 1980s and UML tools of the 1990s – the code generators and modelling languages were built by tool vendors. With DSM, this process is more likely to occur within an organisation. A few expert developers create the modelling language and generators, which are then used by other developers.

Because these tools are built by the organisation that will use them, they can be tailored to the exact domain and needs. This approach also reduces the time needed for developers to learn the language, since it can use familiar terms and concepts. And, since only one organisation's requirements need to be considered, it is easier for the modelling language to be adapted to change.

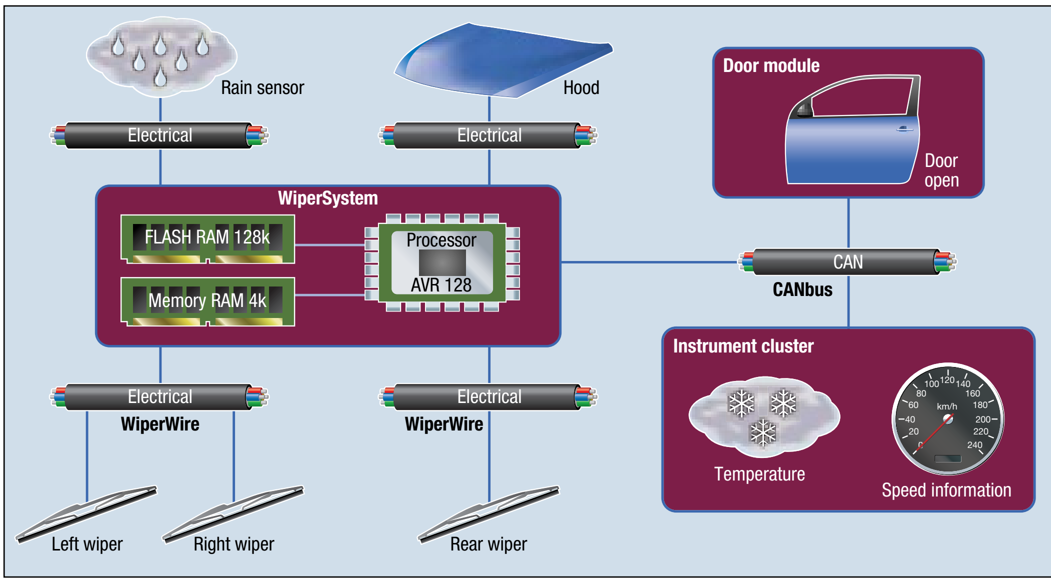
An example of model based code generation from domain specific languages for the automotive sector was presented by Dr Juha-Pekka Tolvanen of MetaCase and Cord Giese of Delta Software at the recent Software im Automobil conference. They described the use of DSM to develop a control unit for windscreen wipers.

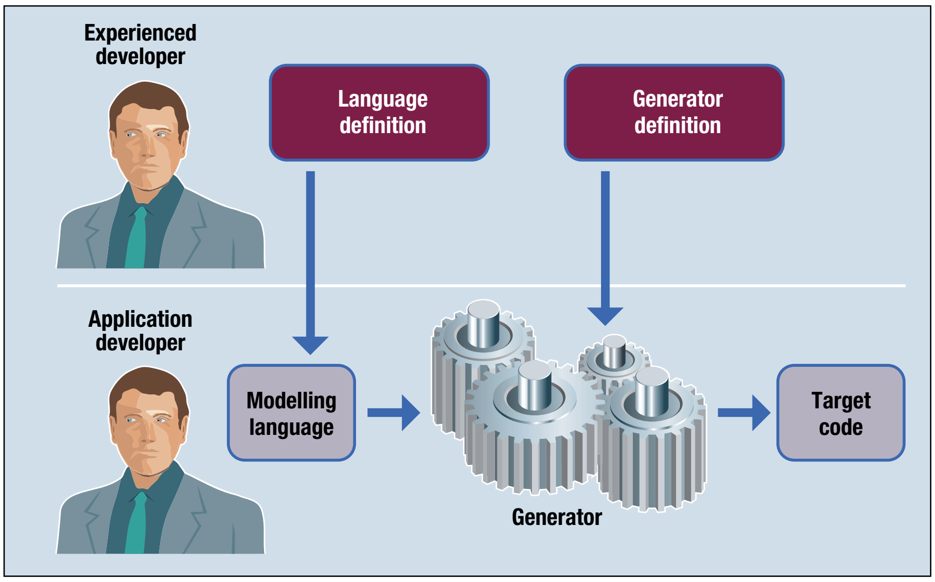
Dr Tolvanen and Giese believe a key element of DSM's success is the focus of the models on the application domain, rather than abstractions of software construction mechanisms. Since the rules of the domain can be incorporated into the language as constraints, the possibility of specifying illegal or unwanted design models can be eliminated. The presenters say companies that apply this approach on top of a platform or framework can generate complete final products automatically from high level specification models.

This is possible because the modelling language and the code generator are designed to fit only one problem domain and its implementation space. Experienced developers in a company specify the languages and code generators, so the resulting code is better than most application developers write by hand.

As an example, they described a windscreen wiper development process. The microcontroller is an AVR128 8bit risc processor with 128kbyte of flash and 4kbyte of sram. The software running on this controller consists of the PURE operating system (an object oriented system configurable for AVR microcontrollers), a generic hardware abstraction layer and several C++ classes to be generated (the concepts also apply to C).

As well as the microcontroller, the system comprises a combination switch, an ignition key, one motor for each wiper, an optional rotary switch and several optional sensors (rain, door open, hood open, outside temperature, speed). It supports one or two front wiper arms and an optional rear wiper arm. The front wipers have three possi-





ble installation positions (left, centre and right).

In order to generate complete code properly, Dr Tolvanen and Giese had to identify the domain concepts and rules relevant for windscreen wiper control. These are dictated by the basic system architecture and what it is required to do. For example, the service 'ignition off' could lead to the response 'finish wiping', which means the wiper arm returns to its start position. A list of such events includes the preconditions, requirements, in/out states and constraints associated with each event, and these are used to define modelling language objects and rules.

"Code is expected to be of better quality than handwritten code: fewer bugs and easier to detect."

Dr Juha-Pekka Tolvanen, **MetaCase**

These rules constrain the use of the language and enforce correct models. For instance, in a wiper control system, speed variation is only via a bus to the main system, which can be chosen from the list of possible bus types (LIN, MOST, CAN). Another rule might specify that a speed sensor can only have one bus connection.

The language is formalised by defining its metamodel. The metamodel allows the user to define the concepts of the language, properties, legal connections between elements of the language, model hierarchy and correctness rules. Support for reuse and various model integration approaches is also essential. Dr Tolvanen and Giese used MetaCase's MetaEdit+ tool.

After defining the language, Dr Tolvanen and Giese provided a visual representation for it in the form of a diagram, although other forms are possible (matrix, table or plain text).

"A good modelling language uses a notation that closely reflects the actual problem domain so the models become easier to read," says Dr Tolvanen. "Using UML style rectangles for all the different model concepts is analogous to trying to understand a foreign language where the only letter is A, with 20 slight variations of inflection."

Dr Tolvanen and Giese took the notation from the real product appearance or easily understood symbols for the sensors.

With the expert created modelling language in place, other developers design specific features by adding elements to the model. The modelling language guides correct designs and checks the required information is given. For example, whilst adding the connection to specific wipers, their position information is needed and verified. If a single wiper configuration is being designed, the model would know automatically the wiper is expected to be in the central position.

Code on the road

The models created form the input to the code generator. At its simplest, the process means each modelling symbol produces certain fixed code, although the generator can also produce different code depending on the values in the symbol, its relationship with other symbols or other information in the model.

"In our windscreen wiper example, we began the implementation work by programming a specific prototype," says Dr Tolvanen. "From a certain level of complexity, such a prototype is needed to implement a generator because it yields the necessary fragments of source code, enriched by task related knowledge.

"Based on this prototype, we divided the source code modules into generic and non generic modules. Whilst the latter were to be generated, the generic modules formed the hardware abstraction layer. In fact, this layer raised the level of abstraction of the other source code modules."

To implement the generator for the remaining modules, Dr Tolvanen and Giese used Delta's HyperSenses generator development tool to define code generators. HyperSenses was coupled with MetaEdit+ by importing the appropriate metamodel and data.

The target code for the specified model is produced by applying the code generator. Since the model provides the configuration for the code generator, no additional work is required – a single button press generates the working target code.

"DSM only makes sense in combination with domain-specific code generation," Dr Tolvanen concludes. "In addition to the productivity gains through automation, the code is also expected to be of better quality than handwritten code: fewer bugs and easier to detect."