Institute for Software Integrated Systems
Vanderbilt University
Nashville,
Tennessee 37235

TECHNICAL REPORT

TR #: ISIS-04-505

Title: Embedded Control Systems Language for Distributed Processing (ECSL-DP)

Author: Sandeep Neema, Gabor Karsai

Abstract

Embedded Automotive systems are becoming increasingly complex, and as such difficult to design and develop. Model-based approaches are gaining foothold in this area, and increasingly the system design and development is being conducted with model-based tools, most notably Matlab® Simulink® and Stateflow® from Mathworks Inc., among others. However, these tools are addressing only a limited aspect of the system design. Moreover, there is a lack of integration between these tools, which makes overall system design and development cumbersome and error-prone. Motivated by these shortcomings we have developed an approach, based on Model-Integrated Computing, a technology matured over a decade of research at ISIS, Vanderbilt University. The center-piece of this approach is a graphical modeling language, Embedded Control Systems Language for Distributed Processing. A suite of translators and tools have been developed that facilitate the integration of ECSL-DP with industry standard Simulink and Stateflow tools, and open the possibility for integration of other tools, by providing convenient and extensible interfaces. A code generator has been developed that synthesizes implementation code, configuration and firmware glue-code from models. The approach has been prototyped and evaluated with a medium scale example. The results demonstrate the promise of the approach, and points to interesting directions for further research.

Table of Contents

1. Introduction

Embedded automotive systems are becoming notoriously difficult to design and develop. Over the past years there has been an explosion in the scale and complexity of these systems, owing to a push towards drive-by-wire technologies, increasing feature levels, and increasing capabilities in the embedded computing platforms. In order to address this level of complexity, the automotive industry has in general embraced the model-based approach for embedded systems development, however the approach is confined to only the functional aspects of the system design and restricted to a limited suite of tools, most notably the Mathworks [3] family of Matlab®, Simulink® (SL), Stateflow® (SF) tools. Undeniably Simulink and Stateflow are very powerful, graphical system design tools for modeling and simulating, continuous and discrete event-based behavior of a dynamical system. However, these tools by no means cover the entire gamut of embedded systems development. Functional design, howsoever difficult, is only one aspect of embedded systems development. There are several other complex activities such as requirement specification, verification, mapping on to a distributed platform, scheduling, performance analysis, and synthesis, among others in the embedded systems development process. The bright side is that there are tools which individually support one or more of these other developmental activities. The down side is off-course a lack of integration among these tools and the Mathworks family of tools, which makes it extremely difficult to maintain a consistent view of the system as the design progresses through the development process, and also requires significant manual efforts in creating different representations of the same system.

Motivated by this severe shortcoming in the embedded automotive systems development process, cooperation was initiated between the Institute for Software Integrated Systems (ISIS) at Vanderbilt University and DaimlerChrysler (DC) AG in 2002 to investigate and develop an approach that specifically addresses the deficiencies in the embedded systems development process pertaining specifically to distributed embedded systems. The cooperation intended to leverage research already underway in this direction at ISIS, and other research institutes. Specifically, the intent was to built upon Embedded Control Systems Language (ECSL), a graphical modeling language developed at ISIS earlier. This language provides the ability to import existing SL/SF models and make them available via open and extensible interfaces. However, ECSL addresses only the functional aspects of embedded systems design. Thus, the specific purpose of the DC and ISIS cooperation is to extend this language to address other aspects of embedded system design mentioned earlier. This cooperation being limited in duration and resources, does not purport to cover the entire range of the embedded systems development process, however it endeavors to design an open and extensible solution that could be enhanced with further research.

This report describes our approach, prototyped as part of this cooperation, based on Model-Integrated Computing (MIC) [1], a mature technology developed at ISIS, Vanderbilt University over a decade of research. The approach that we present is not designed to replace the individual tools in the development process, but complement these tools as an integrator, by facilitating interchange between the different tools, and providing convenient and open interfaces with which it is possible to integrate new tools with relatively modest effort. The key ingredient of our approach is an extensible graphical modeling language that we call Embedded Control Systems Language for Distributed Processing (ECSL-DP). We surround this language with a suite of translators and tools that facilitate the integration of this language in the embedded automotive developments process, starting from functional specification down to synthesis of executable code for the distributed platform. We demonstrate the tool-chain thus created with an example – Rear Window Defroster – that is complex enough to exercise the key capabilities of the tool-chain, and yet modest enough to be able to evaluate it reasonably within the duration of our project.

This report consists of the following sections: Section 2 provides an overview of the embedded systems development process, and highlights the specific activities in the development process that will be supported by ECSL-DP and the associated tool-chain. Section 3 describes the modeling language ECSL-

DP. Section 4 describes the tasks performed and the technical design of the code generator component. Section 5 describes in brief the Rear Window Defroster example, and the application of the ECSL-DP tool-chain to this example. Section 6 concludes this report and offers suggestions for future work. An Appendix introduces the meta-modeling concepts used in GME, which are employed in the ECSL-DP meta-models.

2. Embedded System Development using ECSL-DP

Figure 1 below depicts the conceptual view of activities in an automotive embedded systems development process. Each rounded block denotes a particular activity within the development process. Arrows indicate the workflow between different activities in the sense that information generated during a certain activity will be necessary or has an impact on another activity (ending arrow). For example, the mapping of software components to ECUs requires information about the existing ECUs regarding the hardware-topology and component design. Activities can roughly be grouped in three blocks: 'Hardware Design', 'Software Design', and 'Mapping'. 'Requirements Engineering' is not within the scope of this project, although it is the basis for most of the modeling and development activities. 'Code Generation' is shown in the conceptual development process assuming a model-based approach that allows for synthesis of functional code, in the absence of which this step can be viewed as manual 'Code Implementation'.

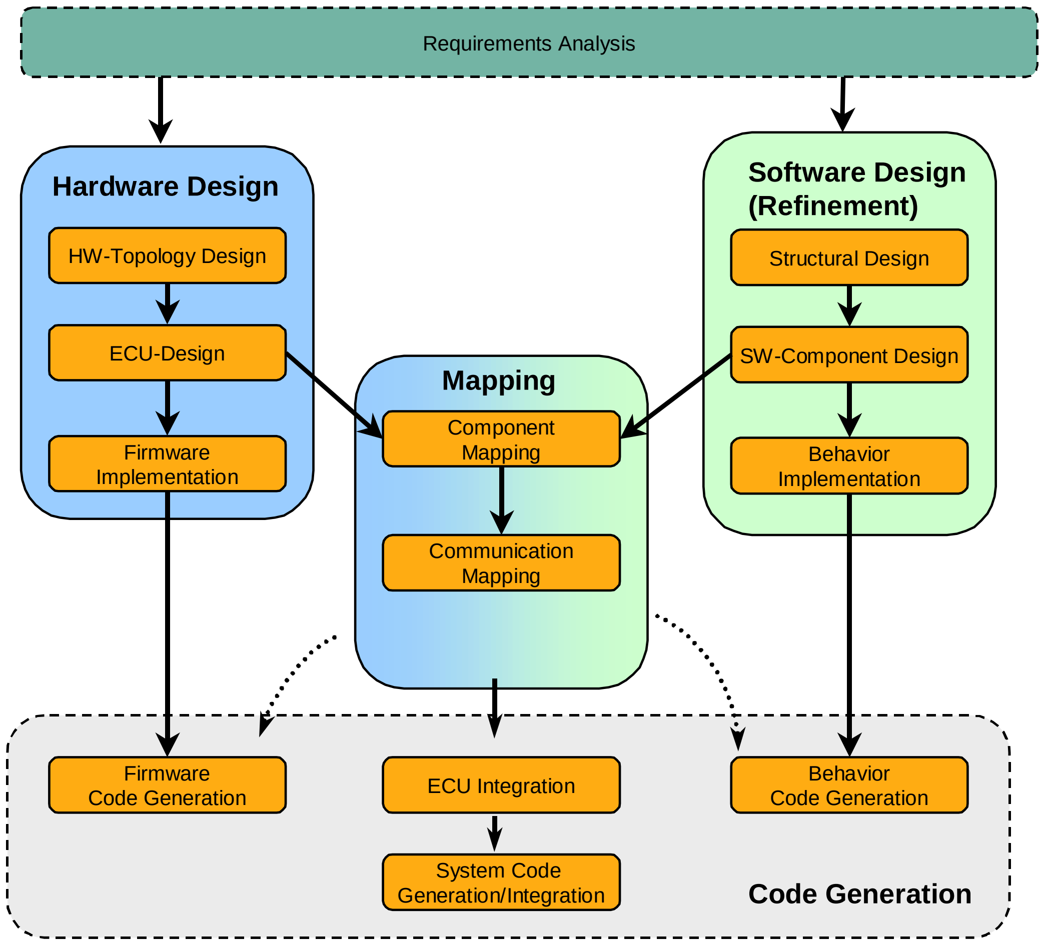


Figure 1: Activities in modeling distributed systems — Conceptual

The intent in the development process is to enable a hardware independent design of the functional software in order to support maximum reuse of software entities. Therefore, it is a major goal for hardware and software design to be developed in parallel and mostly independently of each other.

We briefly elaborate upon the key developmental activities below:

- **Software Design** – deals with: a) structural design, b) component design, and c) functional/behavioral design. The Software design in this view follows a top-down design approach. Structural design refers to the hierarchical decomposition of the embedded system into subsystems and sub-subsystems, from a functional viewpoint. SW Component design is another form of decomposition which is not independent of the functional decomposition, however, brings into bear more of the classical embedded software concerns such as real-time requirements, real-time tasks, periodicity, deadline, scheduling, etc., and the component design is performed with primarily these considerations. The functional design or behavior implementation refers to the elaboration of the leaf elements of the hierarchical structural design in terms of a synthesizable realization.
- **Hardware Design** – includes the specification of ECU-s in a net-work and their connections with busses, defining an architectural topology of the distributed embedded platform. Refinements of this activity include design of individual ECU-s, selecting the processors, determining the memory and I/O requirements. If custom hardware elements are used this activity may also require implementation of elements of the Firmware for the ECU.
- **Mapping** – includes activities involving both software and hardware objects, for example, decisions regarding the deployment of certain complete or partial functions to hardware nodes which are part of the network and the assignment of signals to bus messages.
- **Code Generation/Implementation –** involves creation of low-level coding artifacts, which include RTOS configuration, firmware configuration code, behavioral implementation of the components, etc.

The conceptual diagram captures an abstraction of the design process. To turn this abstraction into reality, a number of tools are needed that support one or more of the activities shown on the conceptual diagram. Figure 2 below overlays the conceptual developmental process with these tools. Note that this diagram does not show all the tools used in design process, only the ones relevant for this discussion. These supporting tools include:

- **GME/ECSL-DP**: This is the Generic Modeling Environment [2], a meta-programmable modeling tool developed at ISIS, Vanderbilt University, tailored to support the ECSL-DP modeling language. Note that tailoring here does not refer to source code modification or creation of a specific version of GME, but simply loading a paradigm definition file (specified with meta-models), in a running instance of GME.
- **ECSL-DP/CG**: A specialized code generator that produces various production artifacts (e.g. source code, configuration files, etc) from ECSL-DP models.
- **ML2ECSL**: Import translators that allow importing Simulink and Stateflow models into the ECSL-DP modeling environment.
- **SL/SF**: These are the Simulink and Stateflow tools.

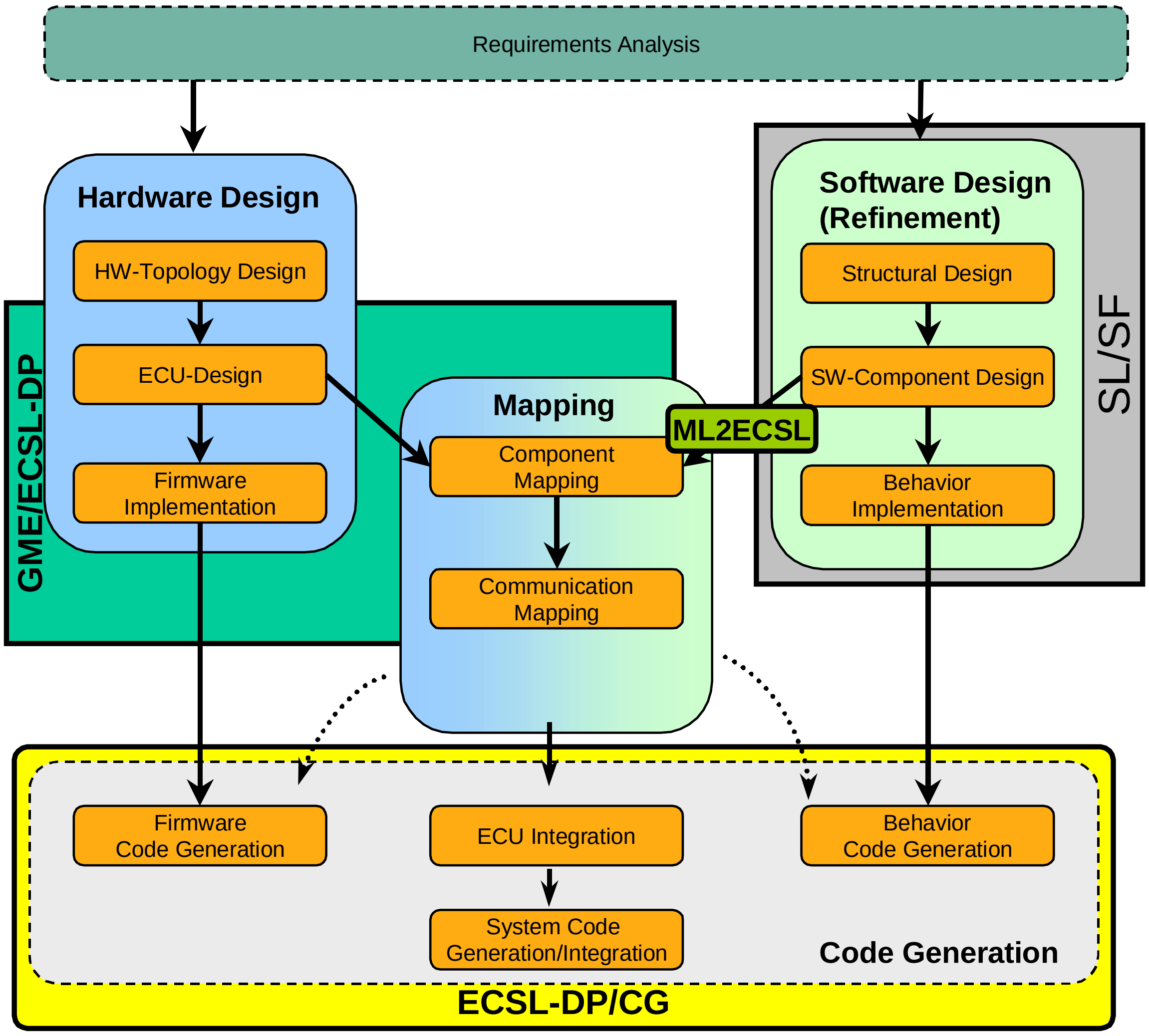


Figure 2: Activities in modeling distributed systems — With Tools

In the above figure, the supporting tools listed above provide support for the developmental activities as shown in the table below:

Tool	Activity
SL/SF	Simulink and Stateflow (*COTS*) are used for the initial construction of the functional design, and behavior implementation. Note that the ECSL-DP has semantic and syntactic constructs that allows the creation of functional design natively within the GME/ECSL-DP environment, however, the intended usage of the tool-suite is such that SL/SF is used for the functional design. This is consistent with the stated objective of ECSL-DP playing the role of integrator.
ML2ECSL	This is an information/data interchange tool (*model transformer*) for importing SL/SF functional design models into the GME/ECSL-DP environment. This tool facilitates easy integration of ECSL-DP in the existing development process that involves SL/SF.

GME/ECSL-DP	The key element of the tool-chain (*modeling environment*) facilitates various modeling activities, including: 1. Annotation of structural design, SW-component design, and behavior implementation to supply information needed by the code generator. 2. Creation of HW-topology design models, ECU-design models, and firmware implementation design models. 3. Creation of deployment models that capture component and communication mapping. The activities are performed in a model-based, graphical way using the capabilities of GME.
ECSL-DP/CG	The *code generation tool* produces code for firmware, ECU integration, and behavior, from the models. This code is in addition to hand-crafted code that is supplied by the designer, if necessary.

The above discussion can be summarized with the following ECSL-DP tool-chain (see Figure 3), supporting the development process described above.

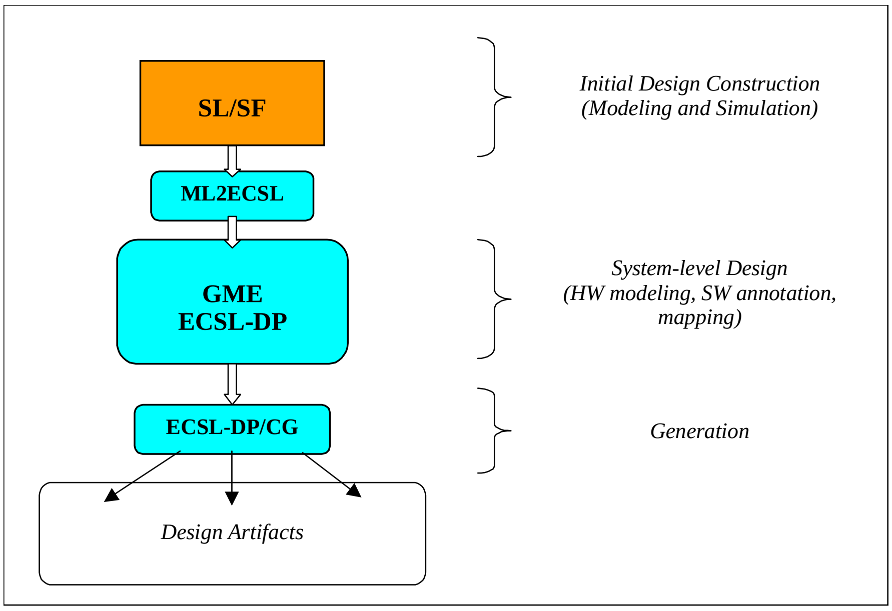

SL/SF
ML2ECSL
GME
ECSL-DP
ECSL-DP/CG
Design Artifacts
Initial Design Construction (Modeling and Simulation)
System-level Design (HW modeling, SW annotation, mapping)
Generation

Figure 3: Tool chain for Embedded Systems using ECSL-DP

The tool chain shown above uses and extends existing tools that have been developed by ISIS earlier, except for SL/SF which is a COTS tool. The ML2ECSL translator consists of the MDL2XML and XML2ECSL tools, ECSL-DP is an extension of the ECSL modeling language, and the ECSL-DP/CG derives from the previously developed ECSL code generator.

3. ECSL-DP Modeling Language

The ECSL-DP modeling language is based on the existing ECSL modeling language as noted earlier. ECSL is a graphical design modeling language, which has design concepts similar to the ones in SL/SF. Specifically, it supports (1) dataflow-diagram oriented modeling of signal flows, and (2) hierarchical state machine diagrams to model finite-state behavior. ECSL was developed in an earlier ISIS research effort [4] to overcome the shortcomings of an SL/SF dominated development process, specifically with the following capabilities:

- Add annotations and additional information, such as timing behavior, memory usage, concrete data-typing, etc. that cannot be represented in SL/SF models natively
- Allow experimenting with code-generators, to provide different optimization, different programming language and different RTOS bindings for the embedded software. The open architecture of GME allows such experimentation, which is very difficult to do in the SL/SF context natively.
- Render SL/SF models/data openly accessible, with an intuitive API that is automatically generated from a precise meta-model of the SL/SF model, for model manipulation
- Allow development of integrated tool-chains that enable analysis and synthesis of code from models, as well as overcome limited SL/SF support for integration

ECSL however falls short on aspects of embedded development process relevant to distributed systems, most notably:

§ ECSL supports modeling only the functional and behavioral aspects of embedded control systems

§ ECSL does not support componentization or task modeling

§ ECSL does not support platform modeling

ECSL-DP leverages off the existing capabilities of ECSL and introduces semantic and syntactic constructs for addressing the above limitations, by extending the ECSL language. In the rest of this section, we first describe the ECSL modeling language, which now forms a subset of the ECSL-DP, and then we describe the extension elements

3.1 The ECSL Modeling Language

ECSL is a graphical language supported by GME. For GME, a modeling language is defined in terms of meta-models that capture the abstract syntax of the language. For illustrative purposes, Figure 4, and Figure 5, below show the (GME-style) meta-models of ECSL[1].

Following the concepts in SL/SF, the ECSL models fall into two model categories: 1) Simulink models, and 2) Stateflow models. The ECSL paradigm uses model containers (or `<<Folder>>`s, in GME terminology) to separately specify these two categories:

- the `Simulink` folder (which contains `System`s corresponding to Simulink models), and
- the `Stateflow` folder (which contains `State`s corresponding to Stateflow models)

[1] In the following discussion the knowledge of the meta-modeling approach used in GME is assumed. The Appendix: Metamodels for Graphical Languages provides a brief summary, while the GME software distribution contains the precise documentation as well as a detailed tutorial.

The two diagrams show the meta-models for the ECSL paradigm. Each diagram represents a portion of the meta-model, which follows the GME conventions: a folder class collects models of the same category in one container. Cross references among the meta-model portions are allowed i.e. if a meta-model element appears on one diagram, in another diagram one refers to it by having a meta-model element of the same name but with a stereotype of the form <<...Proxy>>. For example, in Figure 5, the `BlockRefProxy` (of stereotype `<<ReferenceProxy>>`) meta-model element, refers to, and is the same meta-model element as `BlockRef` (of stereotype `<<Reference>>`) on Figure 4.

In the following discussion we briefly describe both the meta-model portions. On each diagram, one can find a number of classes with various attributes. The attributes follow the GME conventions (i.e. their type comes from the type hierarchy for attributes in GME), and they are indicative of the GUI technique for setting the attribute (instead of the actual type of the data). Internally, most GME attribute types are treated as strings, without further interpretation. Note also that the meta-models capture the abstract syntax for the models. As such, they may contain elements that seem like graphical components (e.g. `Line`), but their semantics is tied to the semantics of the underlying modeling language (i.e. a `Line` means a dataflow connection between ports of blocks). How model elements are actually visualized in GME is determined by their stereotypes, for details see the GME documentation.

3.1.1 Simulink portion

The Simulink portion of the meta-model supports the dataflow-oriented modeling of dynamical systems. The following description elaborates upon the depiction in Figure 4. The toplevel container for Simulink models is the `Simulink` folder. Note that a folder does not have any composition semantics, it is simply a container for organizing models. As such the top-level container of Simulink models with a well-defined composition semantics is really a `System` which is a <<`Model`>> (in the GME terminology) contained in the `Simulink` folder. `System`s are hierarchical as can be observed from the containment relation between the `System` class, and the `Block` class which is an abstract generalization of the `System` class. `System`s are semantically equivalent to the SL concept of SubSystems, and the composition semantics are that of the dataflow model of computation [5]. Thus, a `System` class defines a dataflow relation between the contained `Blocks` (which may be `Systems`, `Primitives`, or `References`), using the `Line` association class, that associates `Ports` of `Blocks`. Note that `Blocks`, `Ports` and `Connectors` are abstract base types (i.e. they cannot be instantiated, thus there are no model elements directly corresponding to them). `Blocks` are subclassed into `Systems`, `References`, and `Primitives.` The `Reference` class (not to be confused with the `<<Reference>>` concept and stereotype of GME) represents an imported block (a library block in SL/SF), while a `Primitive` is a basic block, that has a concrete implementation, and it exists in the local context. `Blocks` also contain `Parameters` and `Annotations`. `Parameters` define configurable properties of a block, for example, the *Gain* parameter of the *Gain* primitive, allows configuration of the gain factor with which the block amplifies the input. `Annotations` are documentation concept that allows a developer to annotate and insert textual comments in an essentially graphical specification. `Annotations` do not have any operational semantics.

`Ports` are subclassed into `EnablePorts`, `TriggerPorts`, `InputPorts`, and `OutputPorts`, each of which corresponds to equivalent modeling concepts in SL/SF and has the same semantics. `Connectors` are sub-classed into `Ports` and `BranchPoints`. The `Connector` abstraction is simply a meta-modeling convenience, which allows abstracting all entities that can participate in a dataflow association, specified with the `Line` association class. Notice that the association class `Line` is stereotyped as a <<`Connection>>` and implies a specific visualization as connecting lines in GME. Thus, `Lines` denote dataflows among `Blocks` within a `System` (via their `Ports` and intermediate `BranchPoints`).

We purposefully, ignore the `BlockRef` class in this description as its role will be clarified in the subsequent description on the Stateflow portion of the ECSL meta-model.

An observation must be made here regarding the `BranchPoint` concept. A `BranchPoint` is an artifact of the Simulink graphical visualization and layout mechanism, and has no operational semantics. An oversight on part of the early ECSL developers led to the inclusion of this concept in ECSL, which could be removed in a future refinement.

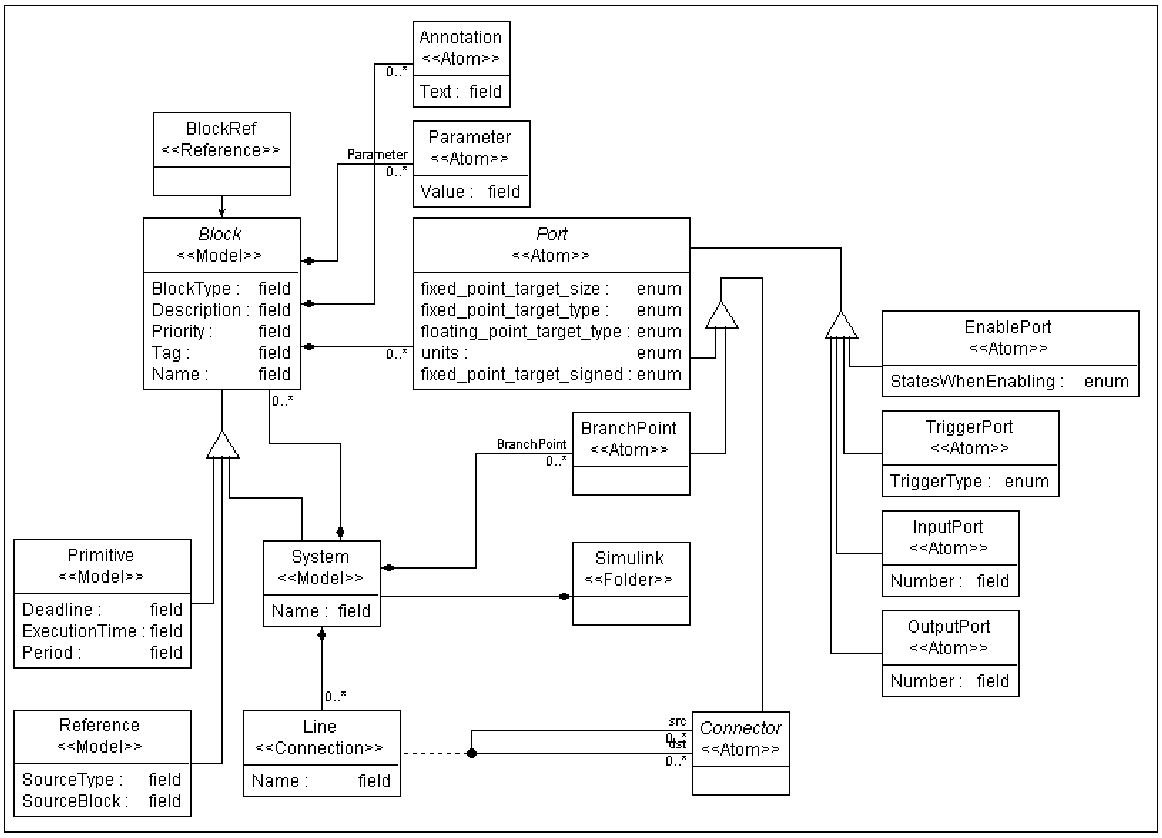


Figure 4: ECSL meta-model - Simulink portion

3.1.2 Stateflow portion

The Stateflow portion of the meta-model supports the statechart-style modeling of hierarchical finite state machines, the semantics of which are described in [6]. Please refer to Figure 5 for the following description. The `Stateflow` folder contains `State` <<`Model`>>s, which are root models for hierarchical state machines, and is equivalent to the State concept in SF. Each `State` can contain a number of `Data`, and `Event` objects – each of which has the same semantics as the equivalent concepts in SF, and subclasses of the (abstract) `TransConnector` (as in "transition" connector) class. The subclasses of `TransConnector` include `Junctions`, `TransInPorts`, `TransOutPorts`, `TransStart` (as in "transition" input and output ports and starting points), `History`, and `ConnectorRefs` (which are <<`Reference`>>s pointing to objects derived from the `TransConnector` base class). `States` contain `Transition` <<`Connection`>>s. These connections connect two objects (derived from the `TransConnector` class), and represent the state transition concepts of the hierarchical finite state machine. The operational semantics of transition is the same as those of transitions in SF; however, the graphical representation differs. In SF, transitions are visualized as a line between the participating states. In ECSL however, a transition is between `TransOutPort` and `TransInPort` of the participating states. The introduction of these ports is an artifact owing to a limitation in an early version of GME that did not allow connections directly to models. The current version of GME allows making such connections, and therefore it should be possible to remove these ports in a future refinement of ECSL and consequently ECSL-DP. A second distinction exists in the graphical representation of cross-hierarchy transitions. SF allows connections cutting across hierarchy, since the SF visualization of hierarchical finite state machines is a flattened diagram. In GME however, this is not feasible since there are no graphical means of depicting connections between objects that are not contained in the same parent. Therefore, ECSL relies on the use of references, which are effectively pointers to objects that exist elsewhere. In order to represent a cross-hierarchy transition in ECSL, a developer must create a reference to the `TransInPort` of the destination state in the parent state of the source state, and then make a transition connection between the `TransOutPort` of the source state, and the referred `TransInPort`.

A `State` model also contains a `BlockRef <<Reference>>`, which points to a `Block` (contained in a `System`, described above). This mechanism provides the linkage between a Stateflow model and a Simulink model. Within the Simulink hierarchy a state machine is represented as a `System` that has `Ports`. These `Ports` have the same name as the input and output `Data` variables in the state machine model. This `System` object contains a `Primitive` S-Function Block, which is referred in the `State`, thus denoting the correspondence.

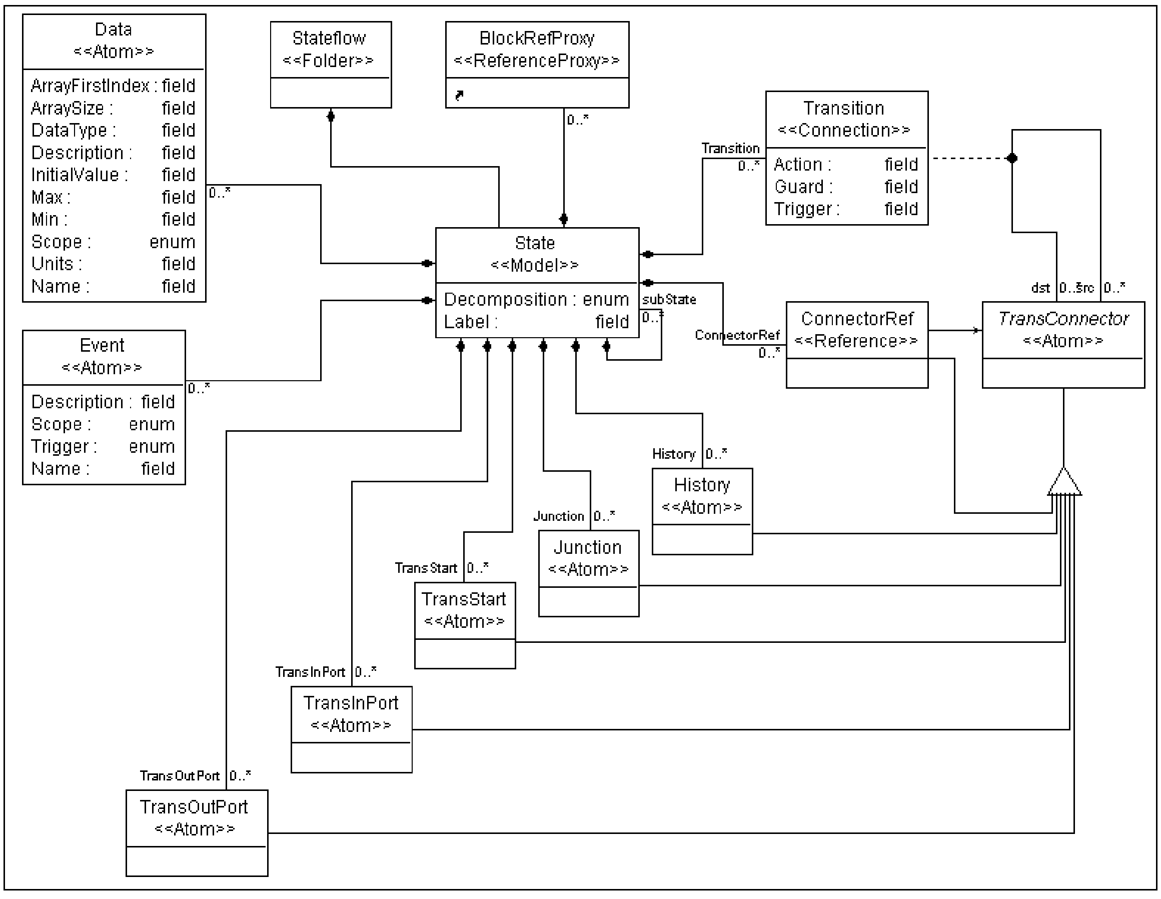


Figure 5: ECSL metamodel - Stateflow portion

3.2 ECSL-DP Extensions

In order to address the limitations outlined earlier, the following extensions have been introduced in ECSL-DP:

1. Component Modeling – A component modeling view has been added (as a folder, in GME terminology). This view combined with the existing ECSL capabilities allows software modeling in two stages: (1) containing models that were imported from SL/SF, and (2) allowing their componentization
2. Hardware Topology Modeling – A hardware modeling view has been added (also as a GME folder). This view allows modeling the topology of the distributed platform including ECU-s, Buses, their physical ports, and their connectivity.
3. Deployment (Mapping) Modeling – A software component to hardware mapping view has been added (as an aspect, in GME terminology of the hardware modeling). This view allows the deployment of components on ECU-s, including association with RTOS tasks, and mapping of component ports on to physical communication conduits (sensors, actuators, and bus messages)

The sections below detail the design of these new ingredients of the modeling language ECSL-DP. Each extension is described by a meta-model and these are in addition to the (baseline) ECSL meta-models introduced above. Each extension can be considered as a new sublanguage of ECSL-DP.

3.2.1 Component Modeling

ECSL-DP components are created from existing ECSL `System` models by encapsulating them in a component. The notational extension introduces the following capabilities:

- Componentization. The designer is able to create components from blocks. A component is a portion of the software model, which is deployed as a unit. The componentization is done using the GME containment and reference capabilities. Specifically, components are GME models that contain references to elements of the functional model (imported from SL). The designer can create Components and specify which sub-trees of the hierarchical dataflow diagram (i.e. the Simulink model) are contained in that component by setting a reference to the root of the subtree within the component. One component in ESCL-DP may contain precisely one System block. This rule is enforced by a constraint.
- Component ports. Components are deployed on processors (ECU-s), and communication between components is facilitated using run-time platform services (for co-located components) and buses (for components located on different processors), while interfaces to physical devices: sensors and actuators is also modeled. This necessitates the introduction of component ports. These component ports should be connected to the system block's ports.
- Signal property definition (scaling, data type, bit width, etc.). Component ports have attributes that allow capturing the required properties. The properties are used in the ECSL-DP/CG to create compact code, which does not rely on the existence of floating-point libraries on the target platform.
- Real-time constraints on software models. Components have atomic elements that allow the designer to specify the real-time constraints (e.g. latency) between selected ports of the component.
- Software dataflow. In ECSL-DP, two-levels of dataflow have to be considered: inter-component, and intra-component. The intra-component dataflow exists within the functional models and it is imported from SL/SF. The inter-component dataflow is introduced by the designer after creating

components from the imported SL/SF models. This step also requires mapping of ports of elements in functional models (referenced in Component models) to ports of component models as discussed above. In the generated code, the inter-component dataflow is implemented using run-time platform services as discussed above, whereas intra-component dataflow is implemented with shared variables, local to a component.

- Port and Signal naming. The names of ports and signals within the functional model are imported as is from SL/SF. The names of component ports are determined by the modeler. In code-generation ECSL-DP follows the convention of using the name of the source port of a signal, when there is a mismatch between the names of the connecting ports and signals.

In summary, ECSL-DP components encapsulate SL/SF `System`s, support the definition of ports (and their association with the ports of the encapsulated `System`), specification of signal properties and real-time constraints. The execution time semantics of an ECSL-DP `Component` is the same as that of the encapsulated `System` model.

Figure 6 shows the Component modeling portion of the ECSL-DP modeling language. An elaboration of the meta-model follows:

- `ComponentModels<<Folder>>` is a container for the `ComponentSheet<<Model>>`-s. A GME Folder is exclusively an organizational concept and has no composition semantics. A designer can create one or more `ComponentModels` folders in a Root Folder (not shown on the meta-model), which is the unique root container in a GME project.
- A `ComponentSheet<<Model>>` is a container for `Component<<Model>>`-s, as well as for component interactions which are modeled with `Signal<<Connection>>`-s. A designer can create `Component` models within a `ComponentSheet`, and model their interactions by creating `Signal` connections between Component ports. For reasons of scalability, and avoiding visual clutter, ECSL-DP allows a designer to create multiple `ComponentSheet` models and distribute `Components` over these. When there is a need to model an interaction between Components that are not located on the same `ComponentSheet`, a designer must create a `ComponentShortcut<<Reference>>` in the `ComponentSheet` where he wants to make the connection.
- A `Component<<Model>>` represents software components. In GME, every modeling object has a name. GME does not impose any restrictions on the naming. However, C code-generation requires that component names form a valid C identifier. The `CName` attribute has been introduced to overcome this restriction. This allows the designer to use a descriptive free-form name for a component, which is displayed in the models, and provide a separate valid C-identifier name in the `CName` attribute. Components contain `SystemRef<<Reference>>`, which is a reference to a `System<<Model>>` (see ECSL: Simulink portion). Notice, the cardinality of the `SystemRef` containment which is set to 0..1. This prevents the user from creating more than one System references within a component. Note however, that this allows creating Component-s that have no System references. In a distributed automotive application, there are situations when Component-s relying on certain Sensor inputs (or generating Actuator outputs) are deployed on an ECU remote from the ECU that is connected directly to the specific Sensor. In such situations forwarder components are required that can forward the Sensor data. In ECSL-DP Component-s that have no Simulink System references, are considered forwarder components.
- A `CPort<<Atom>>`-s, is an abstract class, concretized as `CInPort<<Atom>>`-s and `COutPort<<Atom>>`-s. These represent component ports and define the input and output interface of a component. The `CName` attributes of `CPort` defines a symbolic name for the port that is used in code-generation (similar to the `CName` attribute of `Component`). The `DataType`

attribute is an enumeration of data-types of the signal (Integer, Single, Double), and the `DataSign` attribute specifies if the data-type is signed or unsigned. The `DataSize` specifies the size of the data-type representation as number of bits. The `DataInit` attribute specifies the initial value of the signal associated with the port. The `DataOffset` and the `DataScale` attribute specifies the offset and scaling when converting from the Simulink signal data-type to the concrete data-type specified on component port. The `Max` and `Min` attribute specify the upper and lower bound on the values that the physical signal associated with the port can take.

- A `Signal<<Connection>>` is an association class that represents connections between component ports. The connections originate from `COutPort` and terminate in `CInPort`.
- `InPortMapping<<Connection>>` and `OutPortMapping<<Connection>>` are association classes that represent mapping of Simulink System ports to component ports.
- An `RTConstraint<<Atom>>` allows capturing real-time constraints over component ports. The `Latency` attribute specifies the desired real-time constraint, over when an input is received on an associated `CInPort` (associated via `RTCIn<<Connection>>`), and when the output is generated on the corresponding `COutPort` (associated via `RTCOut<<Connection>>`).

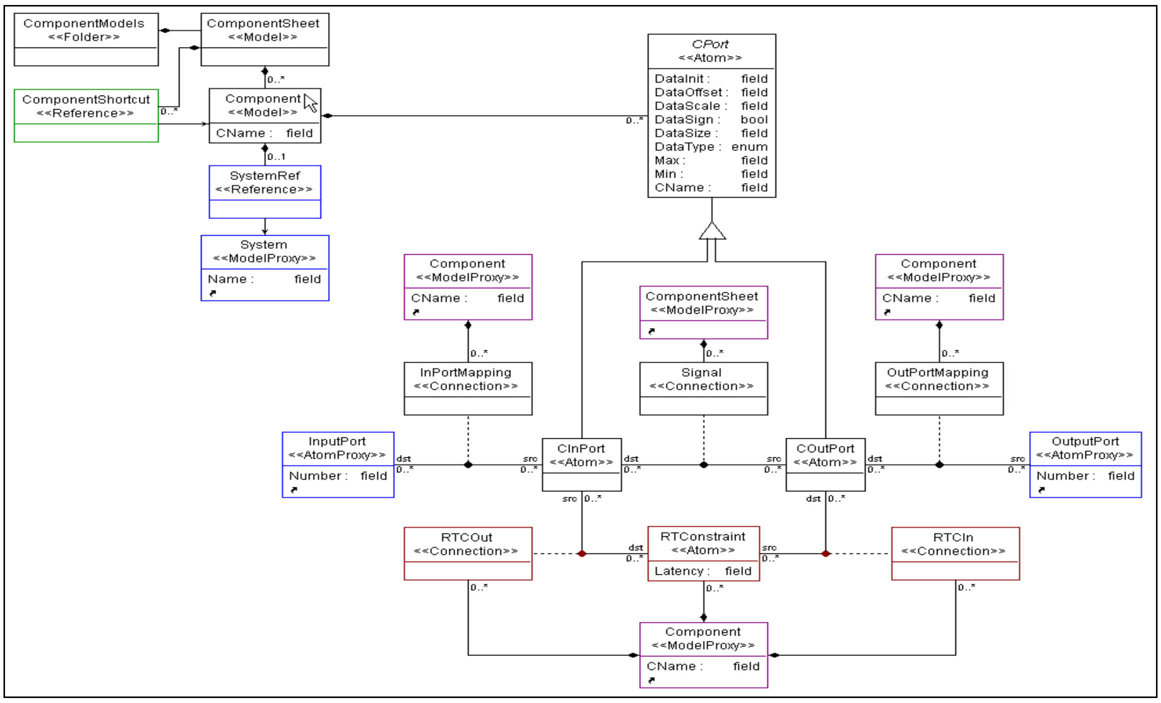


Figure 6: ECSL-DP meta-model - Component Modeling

3.2.2 ECSL-DP: Hardware Modeling

The hardware modeling sublanguage of ECSL-DP allows the designer to specify the hardware topology, including the processors and communication links between the processors. These models introduce new model types: ECUs (which are processors hosting the components), busses (that establish the communication links between the processors, and thus the software components).The details of these models are as follows.

3.2.2.1 ECU Models

ECU models represent specific processors in the system. An ECU is equipped with hardware I/O channels and bus connections, and has a number of other attributes. ECU-s are represented as `<<Model>>-`s in GME, which are ported objects. An ECU model has two kinds of ports (for representing the I/O channels and the bus connections), and (textual) attributes capturing all the other attributes. The specifics of the firmware are captured here as attributes. I/O channel ports come in two variants: sensor ports and actuator ports. As these are separate design objects within the ECU model, they have their own attributes that capture other, relevant properties (e.g. firmware element associated with a sensor).

3.2.2.2 Bus Models

Bus models represent communication pathways used to connect ECUs. Busses are expressed as GME `<<Atom>>`-s and their attributes specify various properties of the physical communication system (e.g. bit rates). Busses connect two or more ECU-s through their bus channels (which are the bus-related connection ports of the ECU-s).

3.2.2.3 Hardware models

Figure 7 shows the Hardware Modeling portion of the ECSL-DP meta-model. An elaboration of the meta-model follows:

§ HardwareModels<<Folder>> is a container for HardwareSheet<<Model>>-s. A designer can create one or more `HardwareModels` folders in a Root Folder.

§ A HardwareSheet<<Model>> represents the hardware topology that is composed with ECU-s, Bus-es, and connections between those. It contains HWElement<<FCO>> which is an abstract class, concretized as ECU<<Model>>, Bus<<Atom>>, and BusConnector<<Connection>>.

§ An ECU<<Model>> represents a physical ECU. The CName attribute of an ECU is similar to CName attribute detailed earlier for Component-s and CPort-s. The CPU attribute specifies the processor family, the RAM and ROM attributes specify the available memory on the CPU, while the Speed attribute specifies the processor speed. The Simulator attribute specifies the name of the simulator used for simulating the ECU.

§ A Bus<<Atom>> represents a physical communication bus. The BitRate attribute defines the transfer speed over the bus, while the FrameSize attribute defines the size of the message frame transmitted over the bus in bytes. The Medium attribute specifies the communication protocol (type) of the bus, such as CAN, or FlexRay, or other. The NM attribute is used by the code-generator to decide if network management code should be generated for the bus.

§ A Channel<<FCO>> is an abstract class, concretized as IChan<<Atom>>, OChan<<Atom>>, and BusChan<<Set>>. IChan-s represent sensor ports, OChan-s represent actuator ports, and BusChan-s represent bus connection ports. Channel-s are contained in ECU-s to represent the physical interface of an ECU.

§ A FirmwareModule<<Atom>> represents a firmware driver that can be attached to a Channel with the FirmwareLink<<Connection>>. The LibraryFile attribute of the FirmwareModule specifies the name of the library in which the driver is contained. If the driver is present in source code form, which should be compiled and linked at build time, then the SourceFile attribute should be filled in to indicate the location of the source code. If the driver is interrupt-driven, then the ISR attribute specifies the name of the interrupt handler. The EventPublished attribute specifies any events that are published by the driver, if it uses events to notify the components. The ReadAccessor attribute specifies the reader API (‘get’ method) provided by the driver, while the WriteAccessor attribute specifies the writer API (‘set’ method).

- § A BusConnector<<Connection>> is an association class representing architectural connections between Bus, and BusChan-s of ECU-s. BusConnectors-s are contained in HardwareSheet models, allowing representation of hardware topologies.
- § COM<<Atom>> and OS<<Atom>> capture OSEK OS and COM attributes. Note that the cardinality of containment is set to 0..1, allowing atmost one instances of each in an ECU. The OS has attributes for Compiler settings, OSEK Conformance class (BCC1, BCC2, ECC1, ECC2, AUTO), Schedule (FULL, NON, MIXED, AUTO), Status (STANDARD, EXTENDED), and TickTime indicating the size of the RTOS clock tick in micro-seconds (this represents the task switching granularity).
- § A BusMessage<<Atom>> represents a physical bus message, a basic unit of communication transported over a bus. BusMessage-s are associated with specific BusChan-s, and the association is represented with the Set membership containment relation. This also explains why BusChan-s are stereotyped as Set-s, different from IChan and OChan. The ID attribute of the bus message specifies a numerical identifier for the Bus Message. The ID also has a priority semantics i.e. attributes with lower ID values are given higher priority over the bus. The Size attribute specifies the size of the message in bytes. The CycleTime attribute specifies the periodicity of a cyclic message.
- § A BusMessageRef<<Reference>> is a reference to a bus message that originates on a remote ECU. The relevance of this is clarified while discussing the deployment.

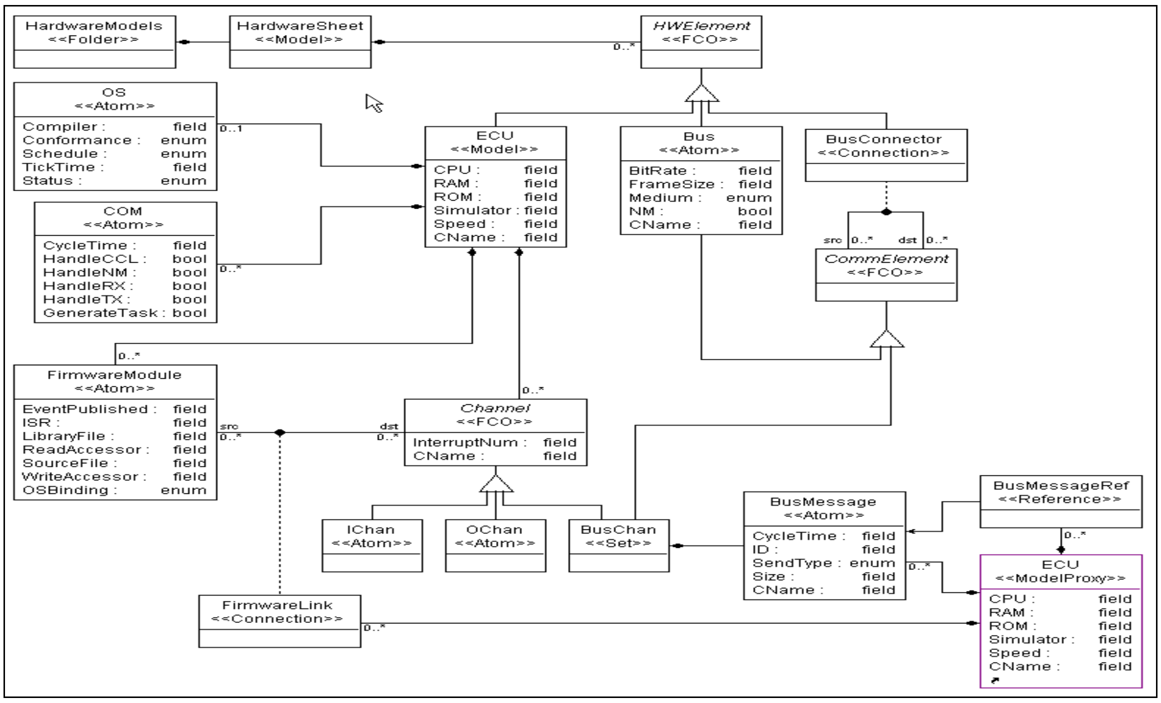


Figure 7: ECSL-DP meta-model - Hardware Modeling

3.2.3 ECSL-DP: Deployment Modeling

The previous two sections described the (software) component modeling and the hardware modeling sublanguages of ECSL-DP. This section describes the third ingredient: deployment modeling, which captures how software components are deployed on the hardware (see Figure 7, from REQ). The deployment models capture the *mapping* (or allocation) of software components onto the hardware architecture. Conceptually, they implement the mapping as shown on **Error! Reference source not found.**. The ECU model has a “deployment aspect” that allows the designer to capture SW component to

ECU mapping using GME's reference concept. In this aspect of the ECU models, references ("pointers") can be placed that indicate that an instance of the component is allocated to the specific ECU. Note that deployment models are separate from software models, thus allowing the reuse of software models in different HW architectures. Furthermore, component ports are connected to ECU ports (sensor, actuators, and bus connections) to indicate how the component software interfaces map to actual sensors, actuators and buses. In the initial version of ECSL-DP all of the connections between component ports to ECU ports and buses will be constructed manually by a modeler. In a later extension some of these connections may be introduced automatically by developing GME plug-ins[2].

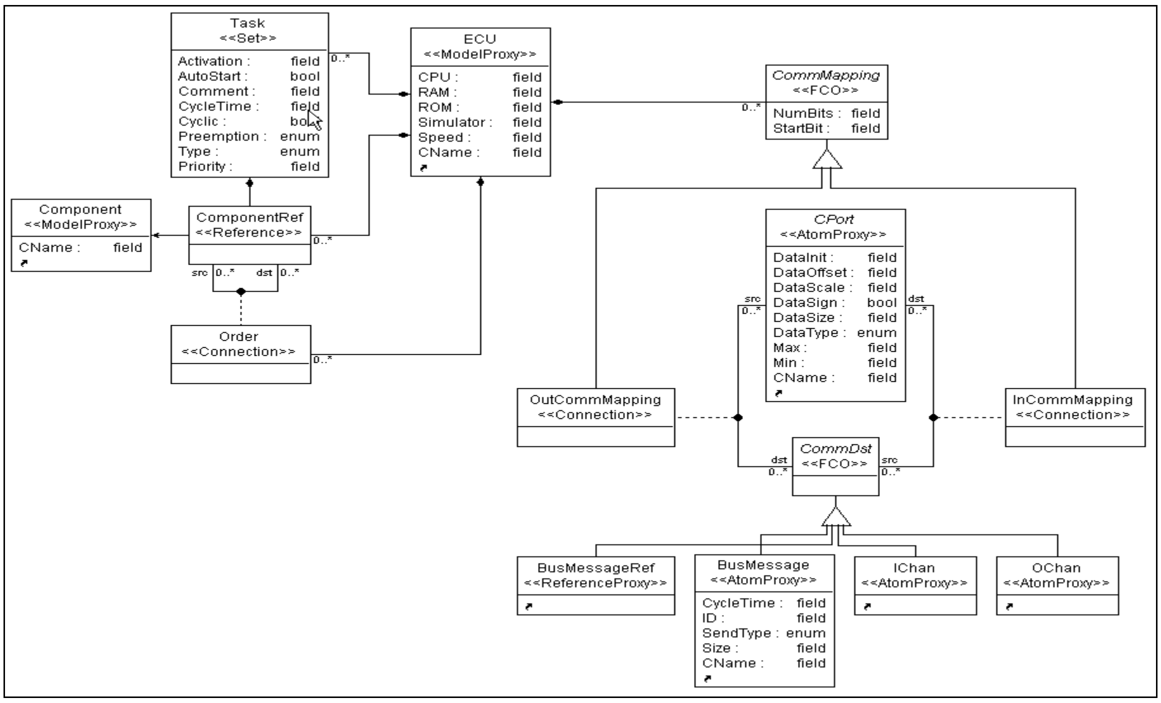


Figure 8:ECSL-DP Metamodel - Deployment Modeling

Figure 8 shows the Mapping (deployment) modeling portion of the ECSL-DP meta-model. Note that this metamodel describes an aspect of the (previously defined) ECU model and thus it does not define a new `<<Model>>` kind. An elaboration of the figure is as follows:

§ A ComponentRef<<Reference>> is a reference to a Component described earlier. ComponentRef-s can be contained in ECU-s to indicate the mapping of components to ECU-s. Furthermore, ComponentRef-s are associated to Task<<Set>>-s with the set membership containment relation. Task-s are stereotyped as <<Set>>-s because GME <<Set>>-s are container where the contained objects are has the same parent as the container. The requirement for same parent container is imminent since we need ComponentRef-s to be immediate children of the ECU for them to participiate in mapping relations with Bus Messages and I/O channels contained in ECU-s, as noted below. Also, note that we could have equivalently represented the mapping of Components to Tasks with Connections (Association). However, the choice was driven by graphical considerations, since multiple Connections running across Tasks and Components increases the visual clutter, whereas Set has a cleaner visualization that does not require introduction of any graphical structures, and is visible only

[2] Plug-ins are small utility programs that provide extra functionality to GME users. They are useful for extending GME's capabilities with new features.

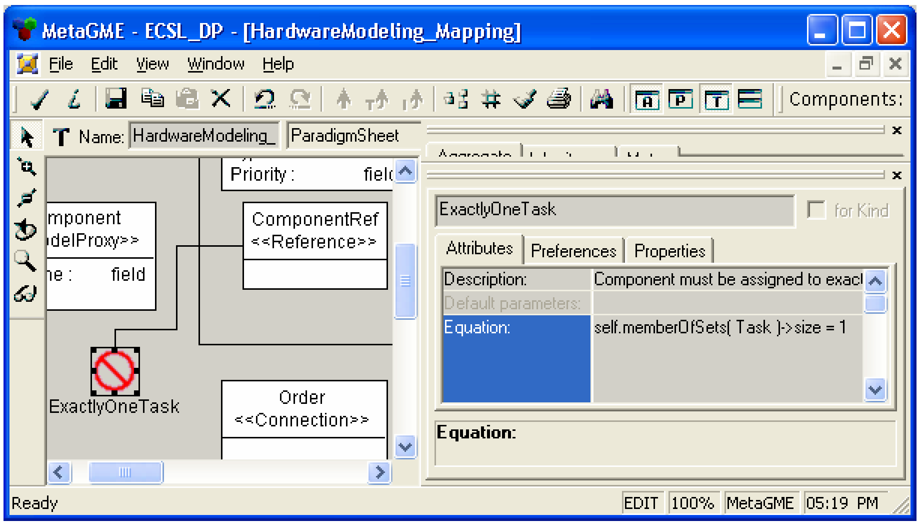
in the Set mode visualization in the GME editor. The containment represents mapping of a Component to a Task. A Task can contain multiple ComponentRef-s, however a ComponentRef must be contained in exactly one Task as a set member. This rule is enforced with the GME/OCL constraint shown in Figure 9 (Equation box on the right). The constraint specifies that the size of the Task<<Set>> must be exactly 1. The constraint is checked by GME, and the modeler user will get a constraint violation message if the model does not satisfy it.

§ Order<<Connection>> is an association class, which represents the ordering of component invocations when multiple components are associated with a single task. The ordering semantics are such that the source component has a higher order than the destination component, when an Order connection is present between components.

§ A Task<<Set>> represents an OSEK task. Various OSEK specific attributes configure the task. The membership containment of ComponentRef indicates the assignment of a Component to a task.

§ InCommMapping<<Connection>>, and OutCommMapping<<Connection>>, is an association class (CPort to/from CommDst), which represents mapping of component ports to hardware channels. Noticeably CPorts are not directly associated to a BusChan, but to a BusMessage. Multiple component ports can be multiplexed over a single BusMessage. The NumBits, and the StartBit attribute of the mapping connection assigns the location of a component port signal within a bus message. A BusMesssage is a first-class entity in the underlying bus communication firmware. Once defined in the communication database, the firmware allocates memory, and provides methods and macros to access the bus-messages. In fact macros are provided that allows access to individual component ports, which are multiplexed over a bus message.

Mapping modeling relies on a GME visualization technique that allows for attaching multiple views (referred to as `Aspects` in GME terminology) to a model, and enabling selective visualization of different parts of a model. The ECU models have two aspects (not visible in the portion of the meta-model shown above): (1) Topology aspect, and (2) Mapping aspect. The topology aspect visualizes topological elements of the hardware platform, while the mapping aspect visualizes mapping elements, notably component references, tasks, and the mapping of component ports to hardware channels. Note that this approach implements the semantics implied by Figure 10.



ExactlyOneTask

Description:	Component must be assigned to exacl
Default parameters:	
Equation:	self.memberOfSets(Task)->size = 1

Equation:

Figure 9: Constraint for enforcing Component Task assignment

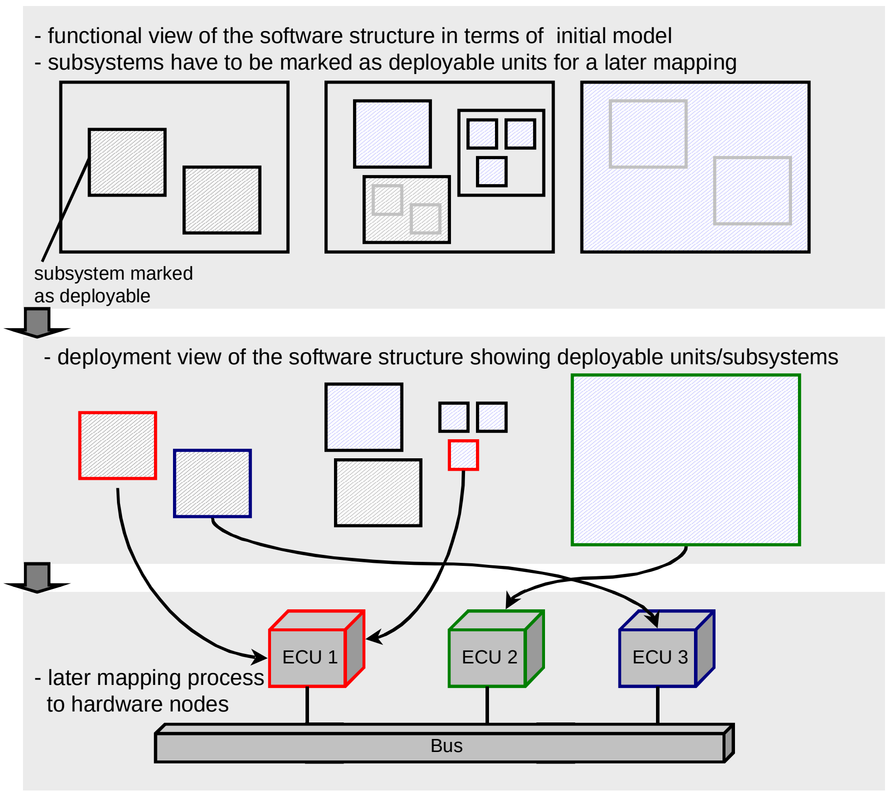

- functional view of the software structure in terms of initial model
- subsystems have to be marked as deployable units for a later mapping
subsystem marked as deployable
- deployment view of the software structure showing deployable units/subsystems
- later mapping process to hardware nodes
ECU 1
ECU 2
ECU 3
Bus

Figure 10: Notional diagram of components and their mapping to HW resources

4. Code Generator Design

The code generator component synthesizes code artifacts necessary for system implementation. Figure 11 shows the artifacts that are generated by the ECSL-DP/CG. As can be seen in the figure, fhe following types of files are generated:

- **OSEK oil-File:** For each ECU-node in the network an oil file is generated, that includes a listing of all used OSEK objects and their relations (see OSEK specification).
- **OSEK Tasks & Code:** All tasks are implemented in one or more C code files.
- **Application Behavior Code:** A separate function is generated for each application component that implements the behavior of the component. This function is called out from within a task frame.
- **Glue Code:** The glue code comprises one or more C code/header files that resolve the calls to the CAN driver or the firmware in order to provide access to CAN signals or HW I/O signals.

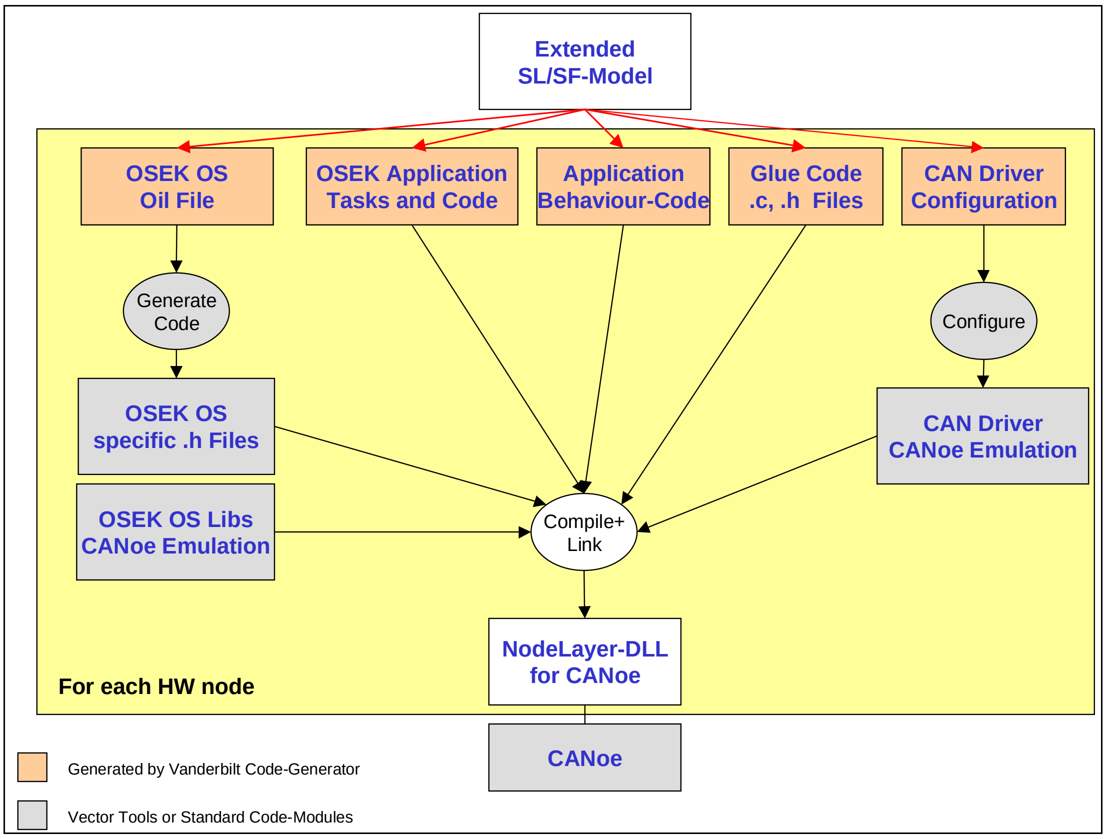

Extended SL/SF-Model
OSEK OS Oil File
OSEK Application Tasks and Code
Application Behaviour-Code
Glue Code .c, .h Files
CAN Driver Configuration
Generate Code
Configure
OSEK OS specific .h Files
OSEK OS Libs CANoe Emulation
CAN Driver CANoe Emulation
Compile+ Link
NodeLayer-DLL for CANoe
For each HW node
CANoe
Generated by Vanderbilt Code-Generator
Vector Tools or Standard Code-Modules

Figure 11: Code Generation artifacts

The code generator is an extension of the existing ECSL/CG package. That package supports the behavioral code generation, but does not support the other requirements. However, the extended modeling language allows capturing the details of the hardware architecture, and the component mapping, and thus the ECSL-DP/CG has enough information to synthesize all the required files.

Note that the ECSL-DP/CG supports only Stateflow, and a specific set of discrete-time Simulink blocks for behavioral code generation. A future extension may support linking to library functions for the unsupported Simulink blocks.

The code generator uses a "traverse-transform-print" strategy in order to gather information from the design models, build intermediate data structures (e.g. tables) as necessary, and then output the resulting code. There are four stages in the code generator each of which involves a multi-pass traversal of the model database. These stages are described in details below.

4.1 Communication Database (DBC) Generation

This stage generates a communication database file that is used for configuring the CAN bus firmware. A DBC file contains specification of bus messages, mapping of signals on to bus messages, and additional CAN bus firmware configuration attributes. We have developed a UML class diagram of the DBC file that describes the "abstract syntax" of a DBC file. The code-generator algorithm traverses the network of ECSL model objects, and builds a corresponding DBC model in terms of objects corresponding to the DBC class diagram. The UDM (Unified Data Model) tool has been used in the implementation. UDM can automatically generate C++ API-s from UML meta-models. Using this generated API, a developer can access and manipulate an object network that is conformant with the meta-model, independent of the underlying persistence mechanism. The details of UDM are described in a conference paper [7].

The DBC data-network thus constructed by the code-generator is subsequently printed as formatted text in a DBC file. The print algorithm follows a simple "traverse-and-print" strategy. Each class in the DBC meta-model has a corresponding Print method, which typically takes the form of emitting text for the host class, and then performing a Print method call on its children.

The "transform" part of the code-generation algorithm involves creating a DBC data-network while traversing an ECSL-DP data-network. The traversal follows the following sequence:

1. At the root level create some a few attributes in the DBC file, and then iterate over all HardwareModels folder, and the contained HardwareSheet models.
2. For each HardwareSheet model, iterate over the contained ECU-s
3. For each ECU, iterate over each BusMessage, and create a corresponding bus object (BO) instance in the DBC data-network.
4. For each BusMessage, traverse all the COutPort-s associated with the bus message with the OutCommMapping connection and determine the startBit, and numBits of the signal in the BusMessage. For each of these create an SG object in the DBC data-network. The attributes of the SG objects, such as name, CName, numBits, and startBits are filled with the corresponding attributes of the CPort. The traversal also determines the ECU where the destination component of the specified communication is located, and populates the destination attribute of the SG object.
5. A second traversal over each ECU, traverses to each IChan and OChan objects, and generates a physical signal element (EV) in the DBC data-network.
6. If the NM attribute of the Bus is enabled, then messages for network management are automatically created.

4.2 OIL file generation

This stage generates an OIL file that is used for configuring the OSEK OS. An OIL file contains specification of tasks, events, alarms, etc. Similar to before we have developed a UML meta-model of the OIL file that describes the meta-data of an OIL file. The code-generator algorithm traverses the ECSL data-network and builds a corresponding OIL model using UDM generated API-s.

The OIL data-network thus constructed by the code-generator is subsequently printed as formatted text in an OIL file. The print algorithm follows a simple “traverse-and-print” strategy. Each class in the OIL meta-model has a corresponding Print method, which typically takes the form of emitting text for the host class, and then performing a Print method call on its children.

The “transform” part of the code-generation algorithm involves creating an OIL data-network while traversing an ECSL-DP data-network. The traversal involves the following key sequences:

1. Iterate over all HardwareModels folder, and the contained HardwareSheet models.
2. For each HardwareSheet model, iterate over the contained ECU-s
3. For each ECU, create an OIL data-network
4. For each OS object (at most one) in the ECU create a corresponding OS object in the OIL data-network, and propagate the attributes. Similarly, for each COM object in the ECU.
5. For each Task in an ECU, create a TASK object in the OIL data-network. Assign the attributes of the Task object, such as cycle time, scheduling, and the Task procedure in the event that the task is an event-driven task. If a task is cyclic then an alarm that is set to trigger every cycle, and an event that is published when the alarm triggers are created in the OIL data network.
6. If a COM object is present in the ECU, and its GenerateTask attribute is set to true, then a Communication Task is created in the OIL Data-network. This task si bound with Alarm-triggered Events that are associated with Network Management messages, CCL, Receive and Transmit.

4.3 Task generation

4.3.1 Signal Definition generation

This stage generates signal definition files. A signal definition file is a C-header file (sigdefs.h) that contains macros to access physical signals i.e. bus signals, and sensors and actuators signals. The macros hide the firmware details thereby facilitating development of portable component code. The code-generator algorithm traverses the ECSL data-network using UDM generated API, and prints a signal definition file for each ECU.

The traversal involves the following key sequences:

1. Iterate over all HardwareModels folder, and the contained HardwareSheet models.
2. For each HardwareSheet model, iterate over the contained ECU-s
3. For each ECU, create a signal definition file
4. For each component reference contained in the ECU, traverse to the referenced component
5. For each CInPort of the Component, determine the associated physical channel connected with the InPortMapping connection. Generate a macro definition in the signal definition file, the signature of which is patterned as “get_$CName()”, where $CName refers to the CName attribute of the CInPort. If the CInPort is associated with a BusMessage, then this macro is defined to a “dbk$CName” call, whereas if the CInPort is associated with a IChan, then the macro is defined to as ‘simGet(“$CName”)’.
6. For each COutPort of the Component, similarly navigate to the associate physical channel with the OutPortMapping connection. This is similar to above except that the generated macro is a “put” macro and takes a value as an argument

4.3.2 Task Procedure generation

This stage generates task procedure code in C-source file which are named as $TaskName_proc.c. A signal definition file contains macros to access physical signals i.e. bus signals, and sensors and actuators signals. The traversal sequence for this stage is defined below:

1. Iterate over all HardwareModels folder, and the contained HardwareSheet models.
2. For each HardwareSheet model, iterate over the contained ECU-s
3. For each ECU, iterate over the contained Task-s
4. For each Task, create a $Task_proc.c file, and generate a void function definition code. The signature of this function is "void $Task_proc(void)", where $Task refers to the name of the task.
5. For each component reference that is a member of the Task set, traverse to the referred Component
6. For each CInPort and COutPort emit a declaration of a local variable. The data-type of this variable is determined using the attributes of the CPort while the name of the local variable is the same as the name of the port.
7. For each CInPort, emit code to perform a get operation, using the macros defined earlier to read the value of the variable. Also emit the code to perform an offset and scaling operation on the values that are read.
8. If there is a reference to a Simulink subsystem, then invoke the Simulink code generator, and emit code to call the generated function for the Simulink subsystem. This requires iterating over the InputPort-s of Simulink subsystem, determining the associated Component ports, and passing the local variable corresponding to those ports in the emitted function call. Subsequently there is also a need to iterate over OutputPort-s, to pass the output parameters.
9. For each COutPort, emit code to perform a put operation, using macros defined earlier to write the value of the corresponding local variable to the physical channels. The necessary inverse offset and scaling code is also emitted. The value of the variable is computed by the code generated for the Simulink subsystem

4.4 Behavior code generation

As noted earlier, functional design of the components is specified in Simulink/Stateflow models, which is represented in the Simulink/Stateflow sublanguage of ECSL-DP. This stage deals with synthesizing implementation from Simulink and Stateflow sublanguage of ECSL-DP.

4.4.1 Simulink code generation

The output of the Simulink code generation stage is a C file that contains implementation functions for Simulink systems and sub-systems. Again, we follow an approach similar to other code generation stages described earlier. We defined a simplified data-model for the output as a UML meta-model that we call SLC. The code-generator algorithm traverses the ECSL data-network and builds an SLC data-network using UDM generated API-s.

The "transform" part of the code-generation algorithm involves constructing an SLC data-network while traversing an ECSL-DP data-network. The traversal involves the following key sequences:

1. Iterate over all ComponentModels folder, and the contained Component models.
2. For each Component model, iterate over the contained System reference-s, note that there is at most one System reference in a Component model.

3. For each System reference, navigate to the referred System. This is the top-level System for the subsequent steps in the transformation algorithm. Create an SLCFile object in the SLC data network

4. Starting from the top-level System, traverse down the hierarchy and create data-type objects (SLScalar or SLStruct) in the output data-network, based on the typing information associated with input and output ports of the ECSL blocks. This step creates SLStruct data-types and populates its members for handling signal busses.

5. In a second pass starting from the top-level System, traverse down the hierarchy and create SLComp or SLPrim objects in the output data-network, based on whether the traversed ECSL block is a System or a Primitive or Reference. This step also creates SLIn or SLOut in the constructed SLComp or SLPrim object corresponding to input and output ports in the ECSL block. This step performs a topological on the contained blocks before traversing further in order to ensure a valid execution order in the generated code. For the constructed SLPrim object, this step also constructs SLParam objects corresponding to the Parameter objects in the ECSL network.

6. A third pass starting from the top-level System, traverses down the hierarchy and constructs SLSig objects in the object data-network which associates the SLIn and SLOut objects, thus mapping the ECSL connections. An SLSig object in the SLC data-network has a single SLOut object "feeding" it, however there can be multiple SLIn object "feeding" from it.

The SLC data-network thus constructed by the code-generator is subsequently printed as formatted text in a .C file. Each class (SLComp, SLPrim, SLIn, SLOut, SLSig) in the SLC meta-model has two Print methods 1) PrintDef, and 2) PrintUse, which correspond to printing the declaration code of a variable or a function, and printing the invocation code of a variable or a function. Moreover, there are a number of overloaded PrintUse functions for the SLPrim class which correspond to different SL block types, for example, PrintUseAbs, PrintUseConstant, PrintUseSum, etc. These functions emit the code for the Primitive SL blocks. The print algorithm follows a simple "traverse-and-print" strategy.

A remark must be made here regarding the integration of the Simulink and Stateflow code generations. In the SL/SF model, an SF block appears as a SL primitive block of S-Function type. In our code generator, the transformation algorithm described above determines if an SL primitive corresponds to an SF block, in which case the code-generator invokes the Stateflow code generator described in the next section. The Stateflow code generator produces code in a C file that implements the logic of the state-machine, and also emits code for a top-level function that serves as the interface between the Simulink code and Stateflow code. In the print stage of the Simulink code generator, there is a PrintUseS-Function method, that simply emits a call to the Stateflow generated top-level function. The two code-generators follow a convention regarding the name of the top-level function, which is $prefix_main, where the $prefix is an argument passed by the Simulink code generation to the Stateflow code generation.

4.4.2 Stateflow code generation

The Stateflow code generation is similar to the previous code generation stages in following a transform and print strategy; however, it is uniquely different from the other stages in the implementation of the transformation. The transformation algorithm of the Stateflow code generation is developed using a Graph Rewriting technique, implemented in the 'GReaT' tool developed at ISIS, Vanderbilt University [ref]. We consider this code generation to be a valuable contribution of this project.

The output of the code generator is a C program that implements the logic of the state-machine. The generated C code is a stylized subset of C, and we have created a UML meta-model of this stylized C, which we call SFC (see Figure 12 below). The key entities in this meta-model and what they represent are described below:

- SFFile – the top-level file object
- InitFxn – initialization function that must be invoked by the generated Simulink code once to initialize the state machine
- RootFxn – the main interface function that is invoked by the generated Simulink code
- SFData/SFEvent – the data, event variables within the state-machine that are the interface to the Simulink code. These variables form the input and output argument list of the root function, note the association between RootFxn and DE, the abstract base class of SFData and SFEvent
- Enter,Exit,Exec – these are the entry, exit, and step function corresponding to each compound state in the state-machine. Fxn is the abstract base class representing a function.
- SFState – these represent the states in the state machine, an enumeration list is printed in the generated code.
- ActiveSubStates – this singleton array variable represents the current list of active sub-state for each compound state in the state machine. The enumeration value of the compound state is used to index into this array to determine the current active sub-state in the generated code.
- Statement – this abstract base class represent code blocks in the generated code. Statements are sub-classed into CompoundStatements, and PrimitiveStatements. CompoundStatements are code blocks that include other Statements. These are sub-classed as Switch, Case, If, and Fxn. PrimitiveStatements are FxnCall, Break, Return, ArgComp, Activate, IsInactive, UExpr, etc.

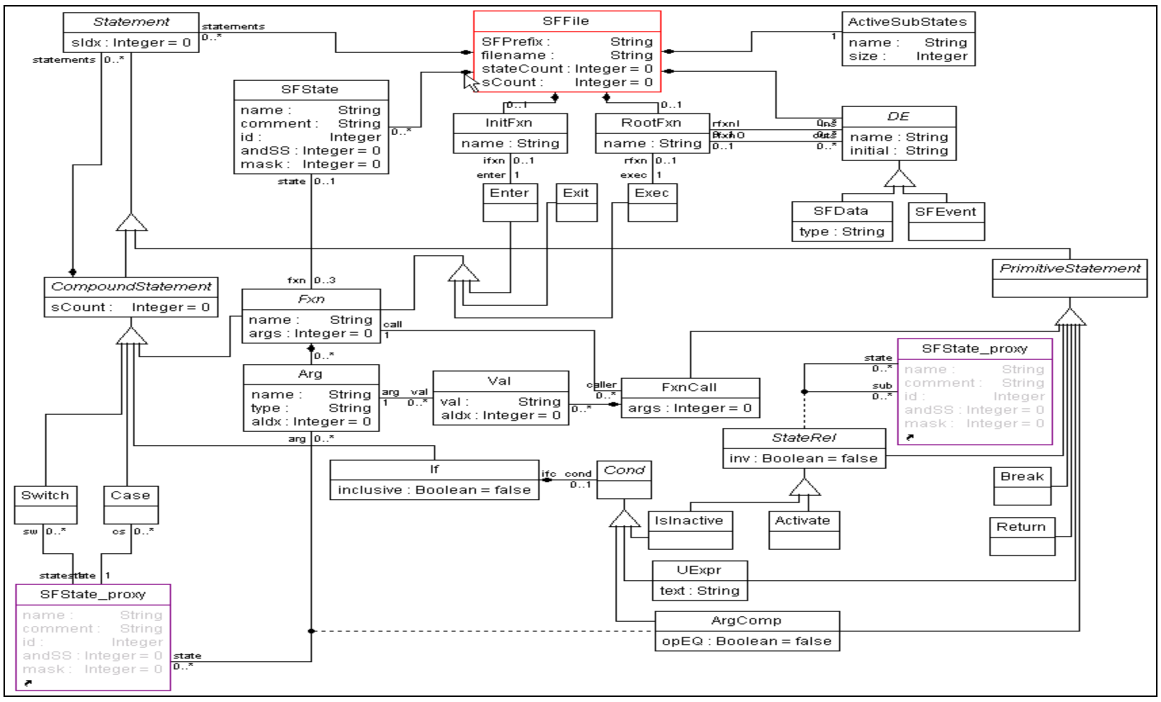


Figure 12: Meta-model of State-Flow C (SFC)

As noted earlier, the transformation in this code generation is implemented as a graph rewriting specification. In the rest of this section we describe the transformation by showing screen-capture of key parts of the transformation specification. It should be noted here that the transformation language implemented by the GReaT tool, has a control flow structure in addition to the graph rewriting instructions. The details of the graph transformation language are reported in a conference paper [8].

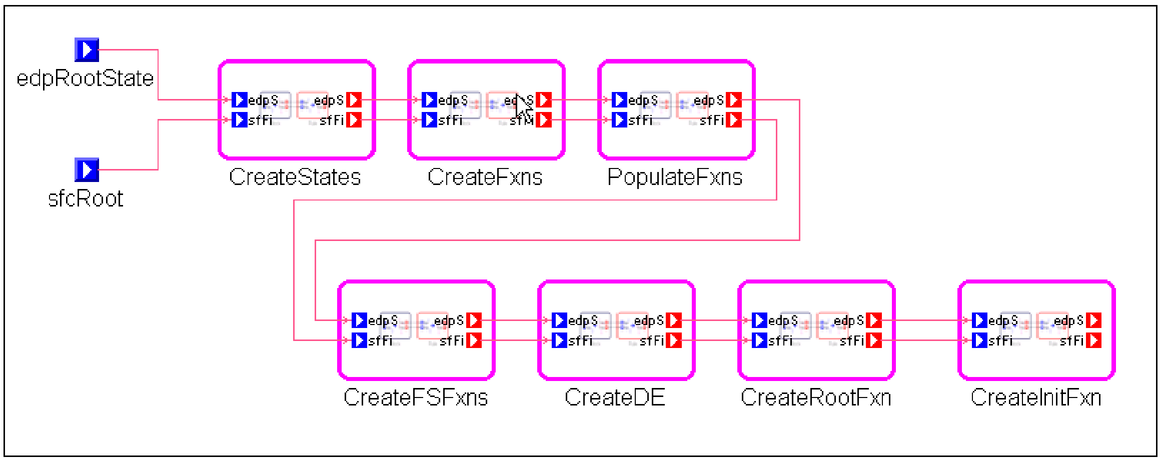


Figure 13: Top-level ECSL-DP to SFC transformation rule

Figure 13 shows the top-level transformation rule. The top-level rule shows the sequencing of sub-rules. Note that the purple colored boxes represent compound rules, and the blue and red colored port objects within this rule boxes represent passing of objects to and from the rules. The ports edpRootState, and sfcRoot in the top-level rule are bound to the top-level state in the ECSL-DP network which is to be transformed, and the root object (a singleton instance of SFFile) in the SFC data-network, respectively. There are seven key steps in the transformation, as shown by the seven sub-rules in the top level rule. The CreateStates rule creates SFState objects in the output data network, whereas the CreateFxns object creates Enter, Exit, and Exec functions. The PopulateFxns rule populates these functions. We navigate down into this rule next.

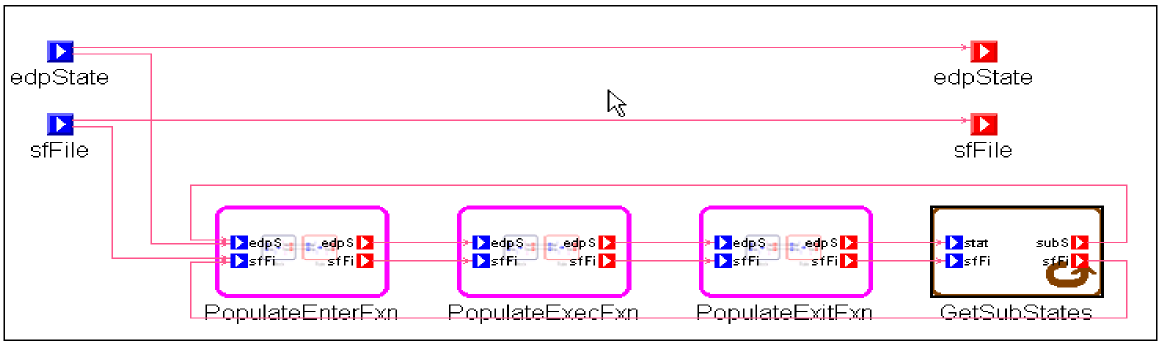


Figure 14: PopulateFxn rule in the ECSL-DP to SFC transformation

Figure 14 shows the PopulateFxn rule. There are three sub-rules in this rule corresponding to the population of Enter, Exit, and Exec function. The fourth rule GetSubStates is visualized differently from the other rules as it is a proxy to a rule defined elsewhere, and demonstrate the ability to reuse rules. Also note the arrows going the GetSubStates rule back to the PopulateEnterFxn rule. This represents a form of recursion - GetSubState rule returns the sub-states of the current state, and the other rules are invoked on the sub-states.

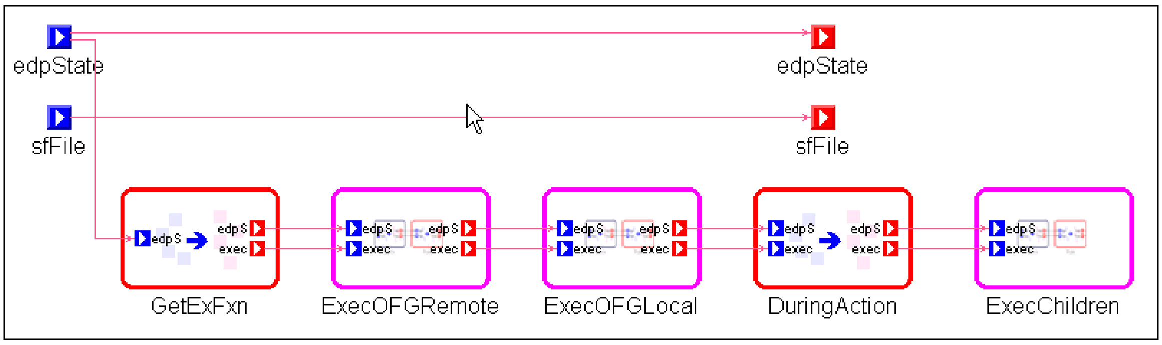


Figure 15: PopulateExecFxn Rule in the ECSL-DP to SFC transformation

Figure 15 shows the PopulateExecFxn rule. This rule generates code for the Exec fuction, which implements a step in a state-machine. The generated code for the step function must check for enabled transitions leading out of this state, and if there is an enabled transition then the transition must be taken which requires a call to the exit function of the source state, performing the transition actions, and invoking the enter function of the destination state in the simplest case. If no transitions are enabled then the during action of the state must be performed, and then the step function must do a step on the sub-states. The ExecOFGRemote, and the ExecOFGLocal sub-rules of this rule emit the code for checking for enabled transitions and performing the transition step. The ExecOFGRemote rule handles remote transitions (source and destination state have different parents), while the ExecOFGLocal rule handles local transitions (source and destination state have the same parent). The ExecOFGRemote rule is invoked prior to the ExecOFGLocal rule since cross-hierarchy transitions have a higher priority than local transitions. The DuringAction is a primitive rule (red-colored box), and we examine it next.

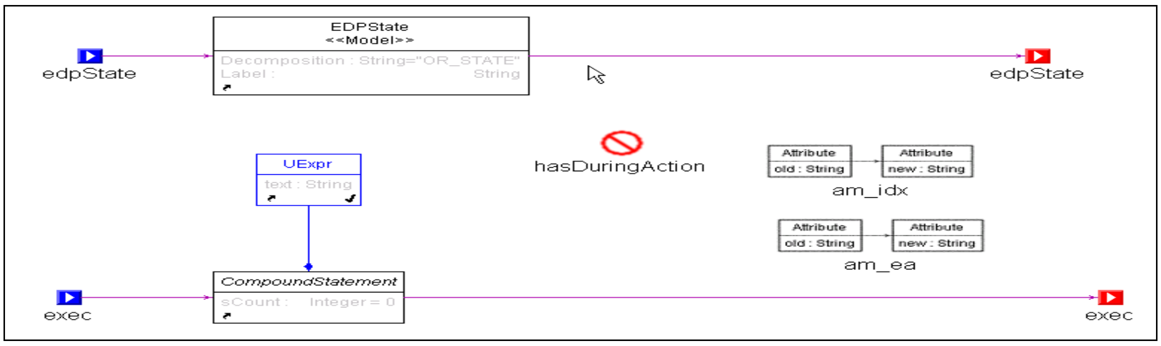


Figure 16: DuringAction Rule in the ECSL-DP to SFC transformation

Figure 16 shows the DuringAction rule. This is a graph rewrite rules, which typically consist of a LHS which represents a pattern to be matched, and RHS which represents the modification in the graph. In this particular rule the pattern is simply an ECSL-DP State, and a CompoundStatement, which are objects passed as input to this rule. The blue-colored class UExpr represents creation of a new object instance of the UExpr class. Also, the blue-colored composition arrow represents creation of a composition relation between the CompoundStatement object and the created UExpr statement. In simple words this rule creates a UExpr object in the output data-network. The boxes labeled am_idx, and am_ea contain attribute mapping specifications. These are code snippets which are executed by the transformation engine when the pattern is matched. The red-circle labeled hasDuringAction is a guard which must be satisfied for the pattern to be matched. In this particular case the guard simply checks that the State has a during action.

There are additional rules in this transformation specification, however, a description of all the rules is outside the scope of this report.

The GReaT tool, compiles these transformation specification into compilable code. The code is compiled and linked with the other code generation stages to build the complete ECSL-DP CG.

5. Case Study: Rear Window Defroster

This section presents a case study in the application of the ECSL-DP tool-suite. A Rear Window Defroster (RWD) was chosen, by Daimler-Chrysler sponsors for evaluating the tool-suite, as it is a distributed embedded automotive system sufficiently complex to exercise various capabilities of the tool-suite, and yet manageable enough to be presentable in a short discourse on the use of the tools. We first give a short overview of the example, and then describe steps through the design flow.

5.1 RWD overview

The RWD system is responsible for defrosting the rear window of an automobile. In addition to the defrosting control, the system is also responsible for updating a display indicating the status, and monitoring the battery voltage levels. The controller processes the temperature sensors information, and generates actuation signals for the heater unit. The RWD system is implemented on multiple ECU-s, since the actuators and sensors are shared by other systems within the automobile.

5.2 RWD Functional Design

The functional design of the RWD system was conducted by DaimlerChrysler engineers using the SL/SF tools. The three main functions of the RWD system: 1) Defrost controller, 2) Display controller, and 3) Voltage surveillance, were designed as SF models. This model of the RWD system is stored in an mdl file (hhs_hk_2.mdl)

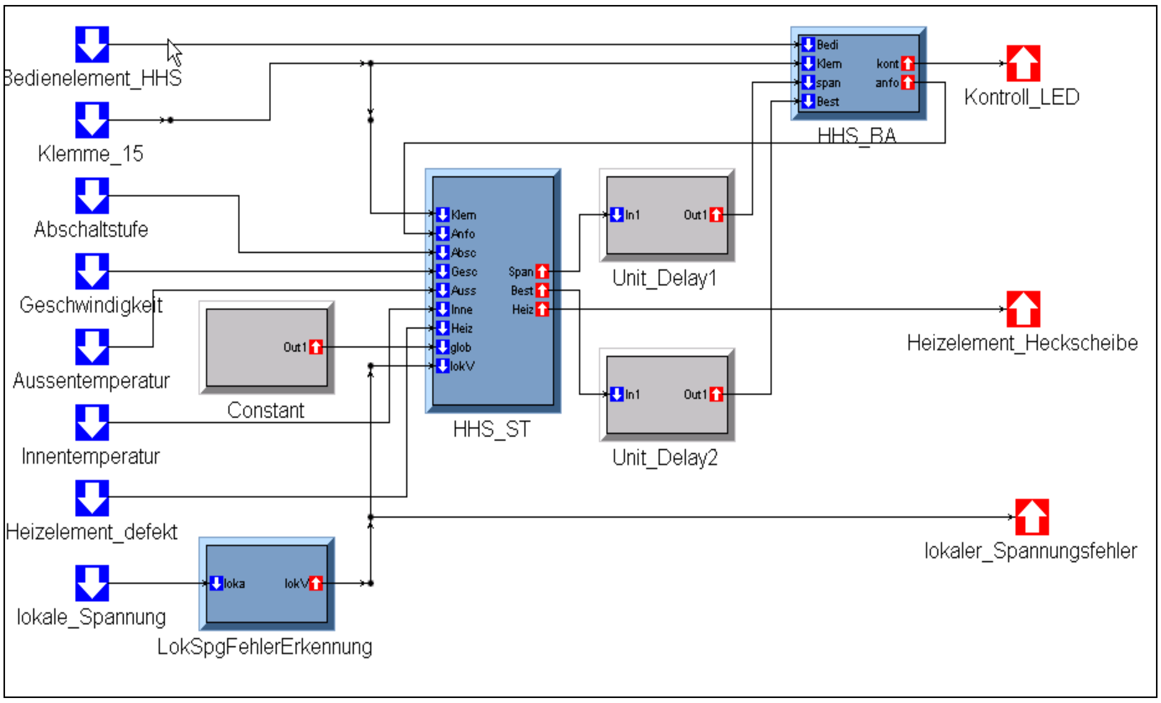


Figure 17: Top-level Simulink model of RWD in ECSL-DP

This SL/SF model was imported into ECSL-DP using the ML2ECSL utilities. Figure 17 shows the top-level Simulink model after importing in ECSL-DP. The three blue colored boxes in this figure represent the three SF models (HHS_ST, HHS_BA, and LokSpgFehlerErkennung). Figure 18 shows the HHS_ST Stateflow model in ECSL-DP. The two sub-states of HHS_ST – HHS_NICHT_BEREIT, and HHS_BEREIT, can be seen in this figure, along with the data and event variables. The names of data variables are color coded to indicate their scope, for example blue is input data, while red is output data, green is constant, while purple is local data.

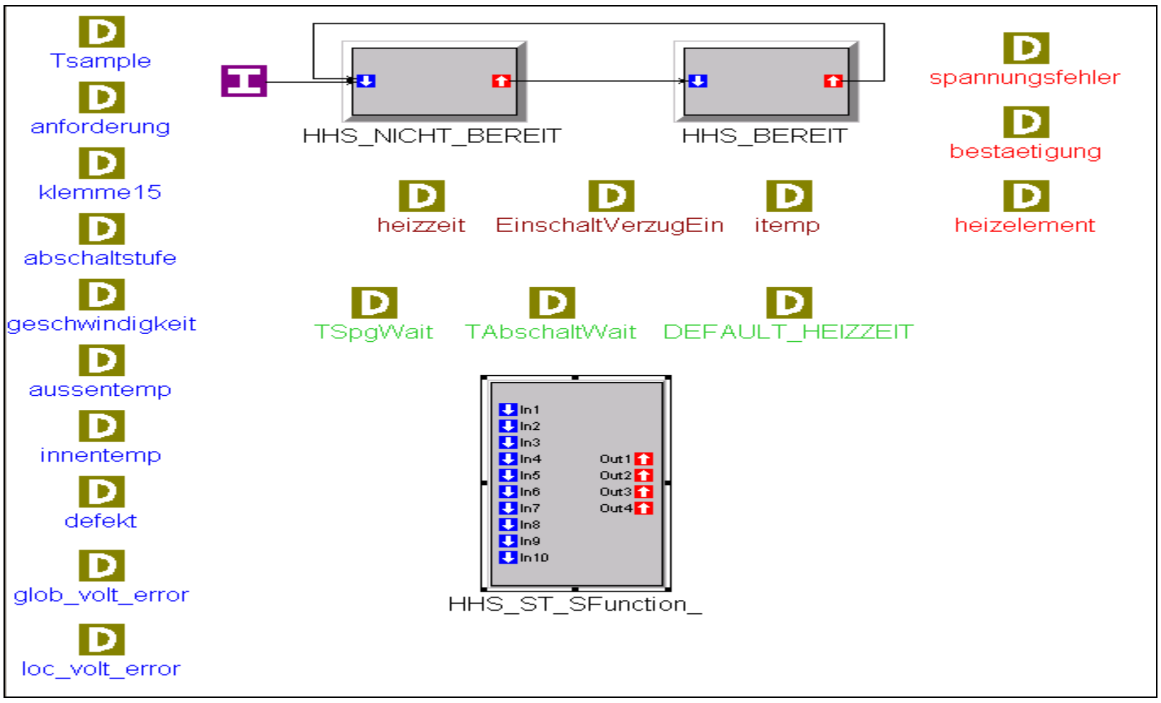


Figure 18: HHS-ST Stateflow model in ECSL-DP

5.3 RWD Component Design

Subsequent to importing the RWD functional design in ECSL-DP, the automotive engineer performs the SW component design in the GME/ECSL-DP modeling environment. Figure 19 shows the component design of the RWD system, which shows the components and their interactions. Five components can be seen, three of which correspond to the three functions of the RWD system listed earlier. The two other components are forwarder components. In a distributed embedded system actuator and sensors are attached to different ECU-s. It is often the case that the data of a sensor attached to a specific ECU is required by SW components on other ECU-s. Forwarder components are special SW components that are responsible for forwarding the sensor data from one ECU to others as bus messages. In the modeled RWD system, there are two such sensor signals: 1) Power Mode signal, and 2) Ignition Clamp signal, that are being forwarded as bus messages.

Note that the view shown in Figure 19 is that of a ComponentSheet<<Model>> (refer to the ECSL-DP Component meta-model Figure 6), and the Components (grey-boxes) are instances of Component<<Model>>-s. The port graphics on the boxes represent the CInPort<<Atom>> and COutPort<<Atom>> objects contained in Component<<Model>>-s.

While performing the component design an automotive engineer is expected to supply concrete data-typing information as attributes of the CInPort and COutPort. Figure 20 shows “drill-down” into the HHS_ST component. The component contains a reference to the HHS_ST System from the functional design. The input and output ports of the System are mapped to the CInPort and COutPort of the Component as depicted with the connections in the figure.

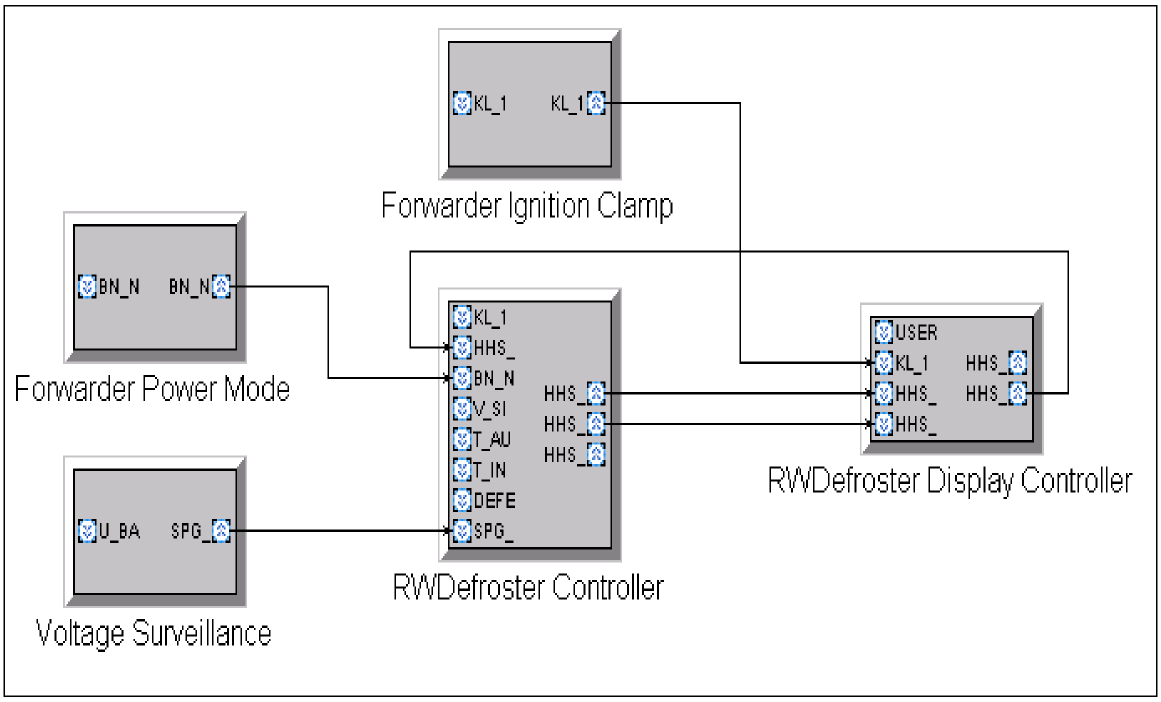


Figure 19: Component design model of RWD in ECSL-DP

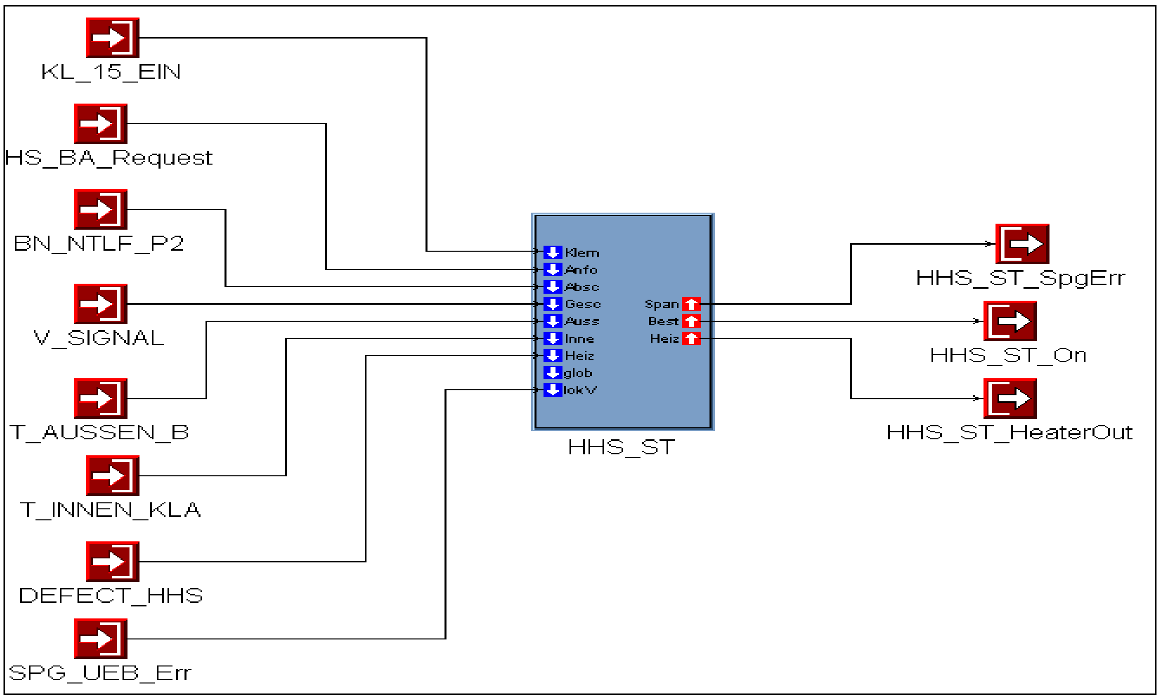


Figure 20: HHS_ST Component

5.4 RWD Platform Design

In parallel with the SW component design, the HW platform design is performed. This involves modeling the ECU-s, Bus-s, Sensor-s, Actuator-s, Bus port-s of ECU-s and interconnects. Figure 21 shows the platform model of the RWD system, with three ECU-s and a CAN Bus. The ECU-s are connected to the Bus with BusConnector<<Connection>> between BusChan ports of ECU-s. The platform design also involves setting the attributes of the ECU-s dealing with processor speed, available memory, and defining the specifics of the OS. Bus Messages are also defined as part of this exercise, and the attributes of the Bus Messages are configured to define the size of the message, its priority, and the frequency of its transmission over the bus. Additional attributes related to communication over the bus are defined as the attributes of the COM object.

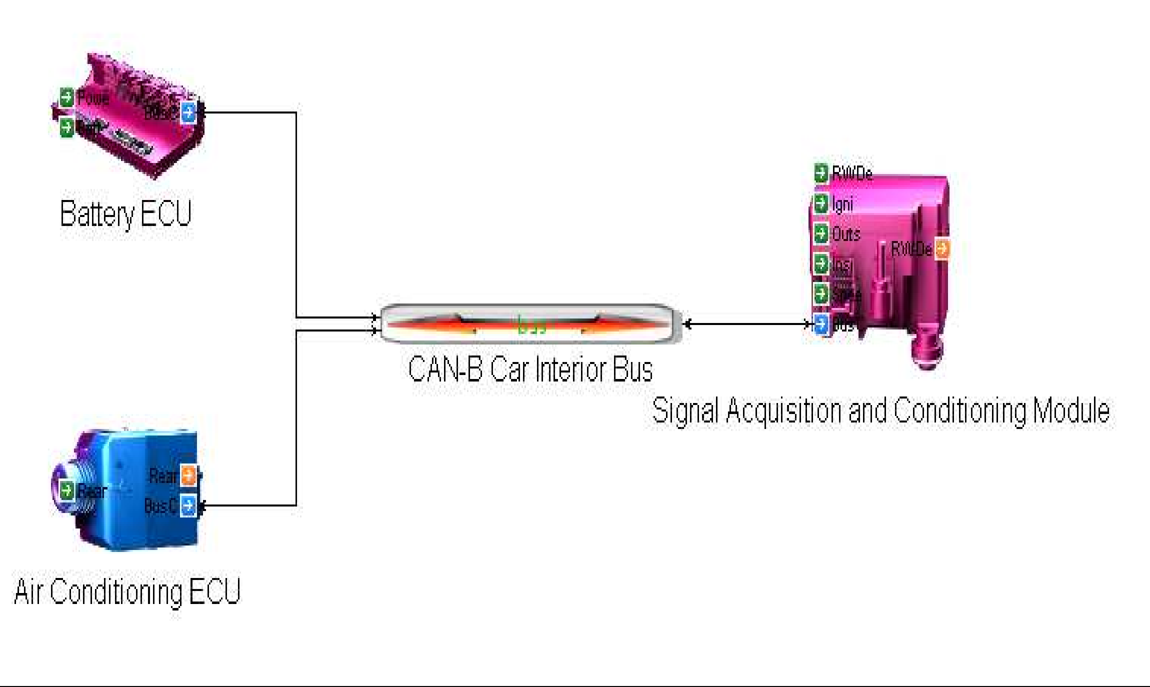


Figure 21: RWD Platform Design model in ECSL-DP

5.5 RWD Component Mapping

The final modeling step involves mapping the SW Components over ECU-s. The mapping also defines the OS Task binding of components, and mapping of component ports to Sensor-s/Actuator-s and Bus messages. Figure 22 shows a mapping view of one of the ECU-s. The shown ECU has two Components (Voltage Surveillance and Forwarder Ignition Clamp) mapped onto it. The component ports are connected to Sensor/Actuator ports or Bus messages. Three BusMessages can be seen, however, only one 'Signal Acquisition and Conditioning Module Message 1' is local to this ECU, the other two messages are references to Bus Messages from other ECU-s. Two OS Tasks can also be observed in the figure. The association of tasks to Components is expressed with set membership relation as explained earlier. This association is viewable only in the Set mode of the GME editor.

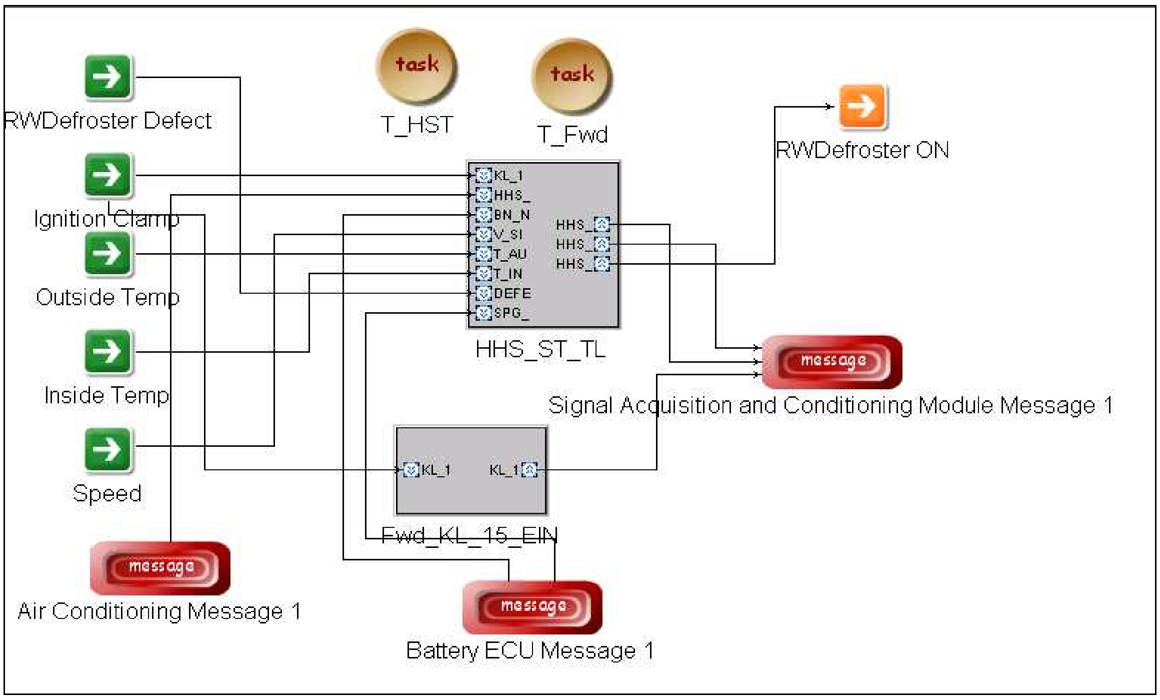


Figure 22: RWD Component mapping in ECSL-DP

5.6 RWD Generated Code

Subsequent to the completion of the modeling step, code generation is performed using the ECSL-DP/CG. This results in creation of several modeling artifacts as described earlier. The code generator creates a separate directory for each ECU, and produces the generated OIL files, and task, and firmware glue code in those directories. A single directory labeled System is also generated, which contains the generated behavior implementation code for individual components. The following table lists the generated files for the RWD example (the top-level directory containing the model is represented with $):

Directory	Files
$	HHS.dbc
$/BSG	BSG.oil, main.c, sigdefs.h, T_Fwd_proc.c, T_SPG_proc.c
$/KLA	KLA.oil, main.c, sigdefs.h, T_HBT_proc.c
$/SAM_H	SAM_H.oil, main.c, sigdefs.h, T_Fwd_proc.c, T_HST_proc.c
$/Systems	HHS_BA.c, HHS_BA_sl.c, HHS_ST.c, HHS_ST_sl.c
	LokSpgFehlerErkennung_sl.c, Spannungs_berwachung.c

6. Conclusions and Future Work

This report described a prototype model-based approach for development of Distributed Embedded Automotive Systems, centered around a modeling language that we call “Embedded Control Systems Language for Distributed Processing” or ECSL-DP, developed as part of a cooperation between ISIS, Vanderbilt University and DaimlerChrysler. We demonstrated with the prototype that it is possible to create tool-chains that integrates industry standard Matlab tool-suite, and provides open interfaces for introducing other tools. The cooperation resulted in several valuable research prototypes, notably the ECSL-DP language, and the ECSL-DP Stateflow C code generator.

We realize that the embedded systems development process is a large and vastly complicated one, and given the limited duration and funding of this project it was not feasible to either cover the entire gamut, or even accomplish great depth in any one of the areas. As such there are several limitations in the prototyped approach that could be addressed in future cooperation and research. There were some shortcomings in the ECSL as noted earlier (BranchPoints, TransInPort and TransOutPort in State-s) which we did not sufficiently address. The tool-chain itself is particularly limited in the absence of any analysis tool.

Some possible suggestions for future work in this area include enhancing the aspects of the modeling language, providing a deeper formal foundation for sublanguages of the ECSL-DP. We would also like to integrate analysis tools such as HSIF, Checkmate, and SAL, into the tool-chains. The real-time constraints that are currently modeled in ECSL-DP are ignored by the code generators. It should be possible to instrument the generated code with real-time constraint checking code. Integrating and generating testing harnesses is another interesting possibility for any future research in this area.

Finally, we conclude this report with an acknowledgement of the funding and support provided by DaimlerChrysler , the sponsor of this cooperation.

Definitions, Acronyms, and Abbreviations

API	Application Programmers Interface
CAN	Controller Area Network
CG	Code Generator
COTS	Commercial off the shelf
CPU	Central Processing Unit
DBC	Communication Database
DC	DaimlerChrysler
ECSL	Embedded Control Systems Language
ECSL-DP	Embedded Control Systems Language for Distributed Processing
ECU	Electronic Control Unit
GME	Generic Modeling Environment
GReaT	Graph Rewriting And Transformation
GUI	Graphical User Interface
ISIS	Institute for Software Integrated Systems
I/O	Input and Output
MIC	Model Integrated Computing
ML	Matlab
NM	Network Management
OIL	OSEK Implementation Language
OS	Operating System
OSEK	Open Systems Executive Kernel
RAM	Random Access Memory
ROM	Read Only Memory
RTOS	Real Time Operating System
RWD	Rear Window Defroster
SL	Matlab Simulink
SF	Matlab Stateflow
SFC	Stateflow C
SLC	Simulink C
UML	Universal Modeling Language
UDM	Unified Data Model

References

[1] Karsai G., Sztipanovits J., Ledeczi A., Bapty T.: Model-Integrated Development of Embedded Software, Proceedings of the IEEE, Vol. 91, Number 1, pp. 145-164, January, 2003.

[2] Generic Modeling Environment: a metaprogrammable modeling environment. For details see: http://www.isis.vanderbilt.edu/projects/GME/ The downloadable package includes detailed documentation and tutorial for modeling and metamodeling in GME.

[3] Mathworks web-site – http://www.mathworks.com

[4] Model-based Synthesis of Generators – http://www.isis.vanderbilt.edu/projects/MoBIES

[5] Lee, E. A. and Messerschmidt, D. G., “Static scheduling of synchronous data flow programs for digital signal processing,” Transactions on Computers, C36 (1):24 --35, January 1987.

[6] Harel, D., “StateCharts: A visual Formalism for Complex Systems”, Science of Computer Programming 8, pp 231-278, 1987.

[7] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages, The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anahiem, California, October 26, 2003.

[8] Agrawal A., Karsai G., Ledeczi A.: “An End-to-End Domain-Driven Software Development Framework”, Domain-Driven Development track, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.

Appendix: Metamodels for Graphical Languages

The Generic Modeling Environment (GME) uses an UML-based approach to define modeling languages. The underlying assumption is that graphical modeling languages have "sentences" formed from objects, i.e. a sentence is a network of objects. A UML class diagram can capture multiple classes, their attributes, and their relationships: inheritance, containment, and general associations. A programmer can instantiate those classes, specify instance attributes, and establish links among objects that correspond to associations in the class diagram. Therefore, a UML class diagram is a finite description of an infinite number of object networks that comply with it, not unlike a context-free grammar is a finite description of a (potentially infinite) language.

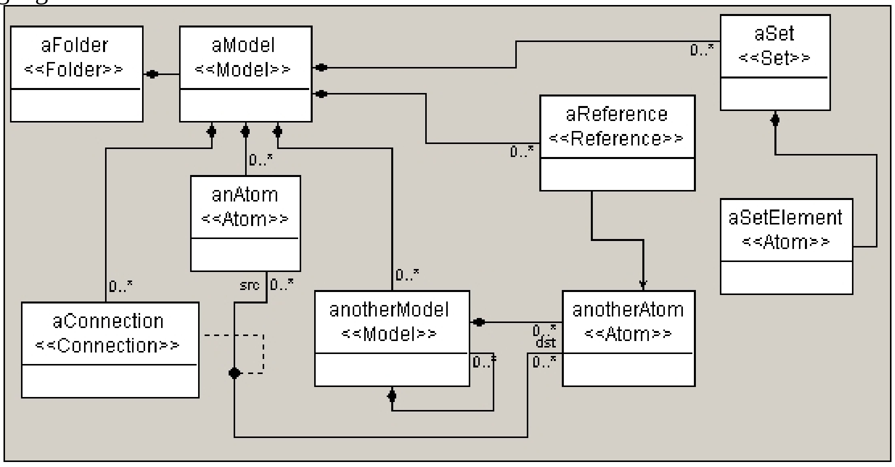


Figure 23: An example meta-model

Unfortunately, pure UML class diagrams are not well suited for the metaprogramming of modeling environments. The reason is that environments tend to support some core modeling concepts (e.g. containers, ported objects, atomic objects, etc.), which are not UML concepts, yet metamodels should contain hints how UML class diagrams should be interpreted in terms of those concepts. A convenient solution to this problem is to use *stereotypes*, which mark classes as belonging to a specific category that is meaningful for (and has a specific semantics in) the modeling environment. Stereotypes are part of the UML standard, but in UML they do not have a specific interpretation —they are simply indicators marking classes as members of some category of classes. In the metaprogrammable Generic Modeling Environment this approach has been chosen. Figure 23 below illustrates how a UML class diagram can be embellished to define a meta-model for GME. The drawing also summarizes the core model organization concepts supported by GME. GME provides the following set of organization concepts: folders (containers), models (ported hierarchical composite objects), atoms (primitive objects), connections (wires), sets (groups of objects), and references (pointers to models, atoms, sets, or other references). The diagram on Figure 23, read as a pure UML diagram, has the following classes: `aFolder`: an untyped container of objects, `aModel`: a typed container with model semantics, `anAtom` and `anotherAtom`: simple objects, `aConnection`: an association class relating the classes `anAtom` and `anotherAtom`, `anotherModel`: a container for `anotherAtoms` and `anotherModels`, `aSet`: yet another container containing `aSetElement`, and `aReference`: associates with ("points to") `anotherAtoms`. The stereotypes map these classes into environment-specific modeling concepts. GME supports `<<Model>>-`s, which are composite objects with ports containing other objects (including other `<<Model>>`-s), `<<Atom>>`-s are primitive objects that have their own graphical icons, `<<Set>>`-s are special containers that contain objects within the same parent `<<Model>>` that also contains the set, `<<References>>` are alias objects which point to (non-local) objects in the object hierarchy, and `<<Connection>>`-s are association objects relating and two (or more) iconic objects. All objects except the `<<Connection>>`-s are iconic. `<<Model>>`-s can have ports on their icons, and `<<Connection>>`-s are visualized as lines. It is not shown on the drawing, but many stereotypes have a corresponding “proxy” stereotype, which is semantically equivalent to the base stereotype. These stereotypes can be recognized by their name, which follows the form `<<...Proxy>>`. A class with name `X` with stereotype `<<S>>` can be referred to on another diagram by a class with name `X` with stereotype `<<SProxy>>`. The metamodel element appearing on the “other” diagram denotes the same metamodel element as `X`. This allows reducing visual clutter.