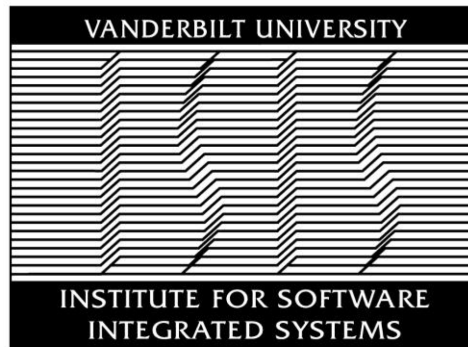

Institute for Software-Integrated Systems

Technical Report

TR#: ISIS-15-107

Title: Software Design and Implementation in the META Toolchain

Authors: Sandeep Neema, Ted Bapty and Daniel Balasubramanian

This research is supported by the Defense Advanced Research Project Agency (DARPA)'s AVM META program under award #HR0011-13-C-0041.

Copyright (C) ISIS/Vanderbilt University, 2015

Table of Contents

List of Figures iii

1. Introduction 1
2. Conceptual Software Design Flow 2
3. Cyber Design Tool Architecture 4
   - 3.1. Software Design and Implementation Toolchain 4
   - 3.2. Hybrid Simulation and Verification Toolchain 7
4. Modeling Languages 9
   - 4.1. Behavior Modeling with Dataflow Diagrams 9
   - 4.2. Behavior Modeling with Stateflow Diagrams 10
   - 4.3. Software Component Modeling 11
   - 4.4. Hardware Platform Modeling 13
   - 4.5. Platform Deployment Modeling 15
5. Code Generation 18
   - 5.1. Behavior code generation 18
     - 5.1.1. Simulink code generation 18
     - 5.1.2. Stateflow code generation 19
   - 5.2. Platform code generation 23
     - 5.2.1. Communication Database (DBC) Generation 24
     - 5.2.2. OIL file generation 25
     - 5.2.3. Signal Definition generation 25
     - 5.2.4. Task Procedure generation 26
6. Scheduling 28
   - 6.1. TT Schedule Generation 28
7. Interface to Physical Dynamics 29
8. Examples 31
   - 8.1. Cyber Component Model 31
   - 8.2. System Model 32
   - 8.3. Cyber Platform Model 33
   - 8.4. Dynamics Evaluation 35
   - 8.5. Platform Software Synthesis 37

9. Future Work 38
Bibliography 39

List of Figures

Figure 1: Software Controller Design and Synthesis Workflow 2
Figure 2: Software Design Tool Architecture for Platform Code Synthesis..... 4
Figure 3: Software Design Tool Architecture for Hybrid System Simulation and Verification..... 7
Figure 4: ESMoL Dataflow Language (Simulink Equivalent) 10
Figure 5: ESMoL Hierarchical State Machine Language (Stateflow Equivalent)..... 11
Figure 6: ESMoL Component Modeling 13
Figure 7: ESMoL Hardware Modeling 15
Figure 8: ESMoL Deployment Modeling 16
Figure 9: Meta-Model of State-Flow C (SFC)..... 20
Figure 10: Top-Level ESMoL to SFC Transformation Rule 21
Figure 11: PopulateFxn Rule in the ESMoL to SFC Transformation..... 21
Figure 12: PopulateExecFxn Rule in the ESMoL to SFC Transformation..... 22
Figure 13: DuringAction Rule in the ESMoL to SFC Transformation..... 22
Figure 14: Platform Code Generation Artifacts..... 23
Figure 15: Assigning Scheduling Information to Components..... 28
Figure 16: The Time-Triggered Scheduling Attributes of the TTExecInfo1 Object in Figure 15..... 28
Figure 17: Modelica Wrapper Code generated for Cyber Controller 29
Figure 18 : Stateflow Model of Shift Controller..... 31
Figure 19: CyPhy Model of the Shift Controller Component..... 32
Figure 20: Torque Control Unit Subsystem in Drivetrain Model 32
Figure 21: Subset of the Drivetrain Assembly Model 33
Figure 22: Platform Model for Cyber controllers of the Drivetrain System..... 33
Figure 23: Mapping and Deployment Model of the Shift Controller Drivetrain System Controller Platform 34
Figure 24: Execution view of the Shift Controller with Task and Timing Association..... 34
Figure 25: Dynamics Test Bench for Full Speed Test 35
Figure 26: Modelica Model generated from Dynamics Test Bench for Full Speed Test 36
Figure 27: Generated Cyber Behavioral and Wrapper code 36
Figure 28: Simulation Results for Full Speed Test showing gear selected, engine RPM and transmission RPM..... 37
Figure 29: Generated Platform Code – OIL Files..... 38

1. Introduction

The need for higher efficiency, performance, and adaptability in systems is increasingly driving the trend towards replacing functions traditionally implemented with purely physical design (for ex. suspension), with a software-controlled design that uses a combination of physical actuators, sensors, and sophisticated control strategies implemented with software. The industry and research literature is replete with examples and statistics of the increasing role of software in such systems (e.g., number of processors and software lines of code in a modern automobile). Indeed, the coinage of the term Cyber Physical Systems is a recognition of this trend, and a recognition of a host of challenges concomitant with the tight coupling between computational and physical processes.

Towards the broader goals of AVM (stated earlier), this trend entails that, a) Software is increasingly a system complexity and schedule driver, and b) Correctness-by-construction is intrinsically coupled with the behavioral correctness of the software and timing correctness of the implementation on a computational platform.

The software design and implementation tools, integrated into the OpenMETA tools, were driven by the implications above, and designed to fulfill the following requirements:

1. Reduction in Software Complexity
    a. Provide higher-abstraction formalisms for modeling software and computational platforms
    b. Provide automated tools for behavioral and platform code synthesis
2. Correctness by construction
    a. Provide automated tools for co-simulation of software and physical dynamics
    b. Provide automated tools for verification of hybrid state-space formed by software and physical

In fulfilling these requirements, we had to be cognizant of the broader industry and research community trends in terms of tools, methodologies, practices, and tangible assets. Software design in industry (especially embedded and controller software design) is increasingly shifting towards commercial model-based tools such as Simulink/Stateflow by Mathworks, Rhapsody, etc., and there is a growing community of practitioners and engineers using these tools as well a large asset base of models and toolboxes created within this tools. In keeping with our model integration approach articulated earlier, we chose a Simulink and Stateflow-like abstraction as our primary formalism for representation of software components. Our prior work [1] in this area has already developed some of the tools necessary for integrating with Simulink and Stateflow that is leveraged in the OpenMETA tools.

The purpose of this chapter is to introduce the software design, implementation, and verification tools of the OpenMETA toolchain. The rest of this chapter is organized as follows: section 1 introduces the conceptual workflow for (controller) software design, implementation, and verification as a sequence of activities; section 2 introduces the tool architecture including the

tools used and the interfaces as well as information flows between the tools; section 3 provides a description of the modeling languages used in the cyber tools; section 4 provides a description of the code generators (developed in OpenMETA or extended from prior work);

2. Conceptual Software Design Flow

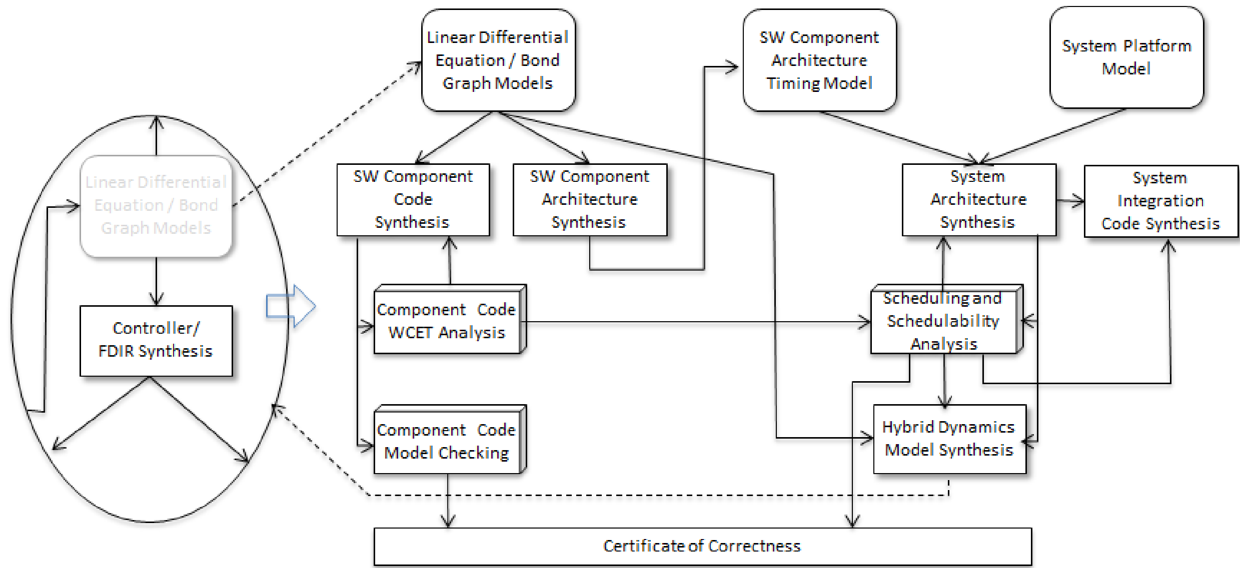

Figure 1: Software Controller Design and Synthesis Workflow

The conceptual design flow for software controller design and synthesis is depicted in Figure 1. The depicted design flow is a refinement of the overall META design flow (described in Chapter 2 – design flow evaluation), for software components. Software components together with sensors and actuators are responsible for realizing physical dynamics that couple with the rest of the physical components and subsystems, which in OpenMETA is described with Modelica. The logical software design flow progresses as follows:

1. *Linear Differential Equations/Bond Graph Models*: The logical software design flow begins with the causalization of the physical dynamics, identifying the input (sensed) and output (actuated) variables. Causalization of dynamics is required as the physical dynamics of the system described with Modelica are typically acausal. The continuous physical dynamics are then approximated with discrete dynamics along with selecting sampling rates for the individual blocks in the controller subsystem. These activities are typically performed in tools like Simulink and Stateflow, which results in a functional block architecture of the controller.
2. *Software Architecture Synthesis:* This functional architecture is then further refined and partitioned into a set of software components and a software component architecture. This activity is performed in the ESMoL modeling suite (described later).

3. *Software Code Synthesis*: The software components, which are defined with Simulink and Stateflow models, are translated into code using code synthesis tools, either commercial such as Targetlink or Realtime Workshop, or code generators such as Stateflow-to-C and Simulink-to-C provided with the ESMoL toolsuite.
4. *Component Code WCET Analysis*: These generated codes are subjected to timing analysis to estimate the worst case execution time (WCET),
5. *Component Code Model Checking*: Generated component codes are model checked, using Stateflow model checker as well as CBMC - a C-code model checker. The result is an instrumented and individually verified code for software components.
6. *System Platform Model*: In parallel, a computational platform model is defined that includes processors, communication buses, and networks.
7. *Software Component Architecture Timing Model*: The software component architecture model, annotated with timing regarding the worst-case execution times of software component code, as well as the periodicity of execution based on the sampling rates leads towards the system architecture model.
8. *System Architecture Synthesis*: In the system architecture model, software components are deployed on the platform model and communication between components is mapped to messages on buses.
9. *Scheduling and Schedulability Analysis*: The resultant model is then subjected to schedulability analysis and schedule synthesis (using scheduling tools provided with ESMoL).
10. *System Integration Code Synthesis*: The fully resolved system architecture model with schedule information is then processed with system integration code generators that produce code for configuration and integration of software components with operating system tasks, as well as configuration of operating system and platform services for scheduling and dispatch of bus messages.
11. *Hybrid Dynamics Model Synthesis*: Finally, the software code and platform that represent a discretized implementation of physical dynamics is re-integrated with the physical dynamics of the rest of the system, and subject to hybrid systems verification (using tools such as HybridSAL). The verification can check for functional correctness of the combined cyber-physical system, against properties specified with temporal logic.

3. Cyber Design Tool Architecture

3.1. Software Design and Implementation Toolchain

Figure 2 depicts the architecture of the software design toolchain for behavioral and platform code synthesis. The tool architecture can be best explained by partitioning into the following core set of activities: 1) Cyber (Component) Behavior Modeling, 2) System Design Space Authoring (Modeling), 3) Cyber Platform Modeling, and 4) Software Manufacturing (Code Generation).

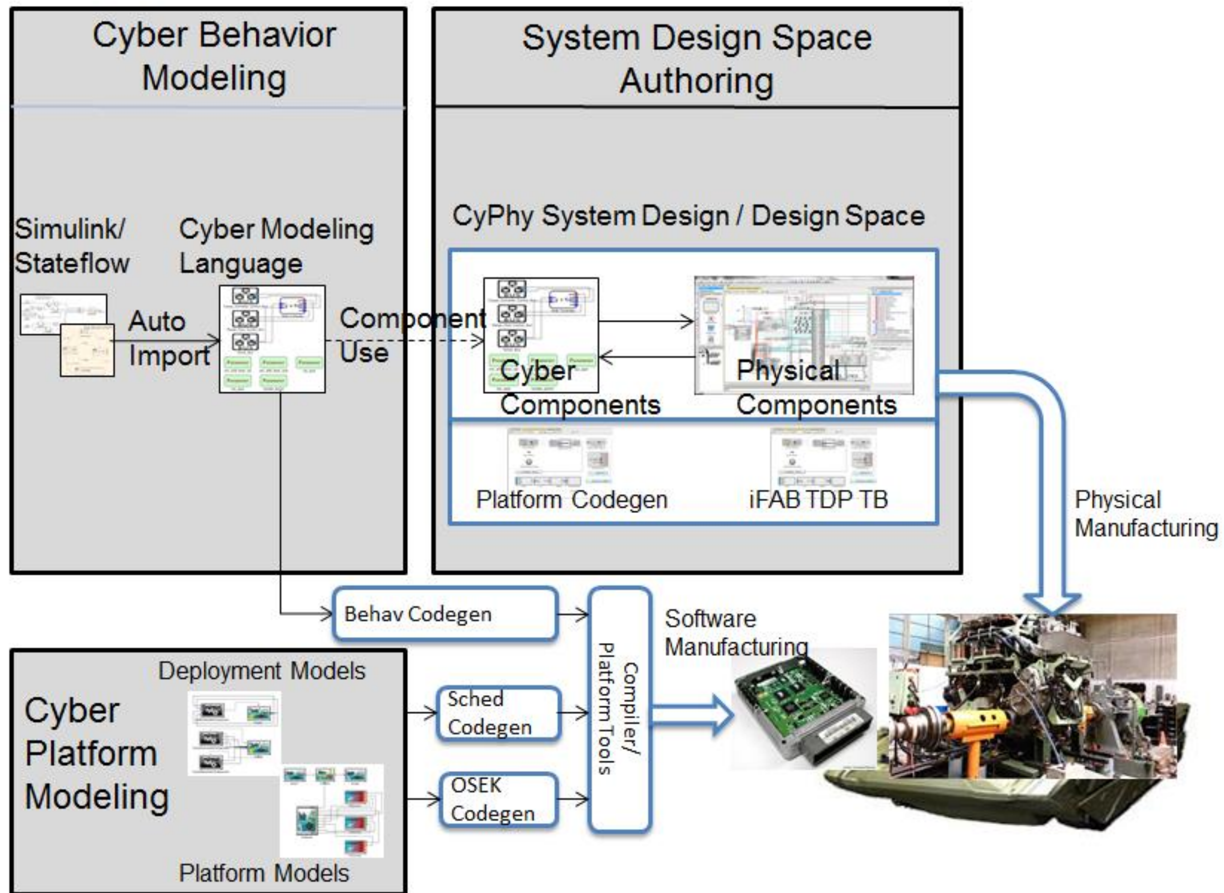

Figure 2: Software Design Tool Architecture for Platform Code Synthesis

Cyber (Component) Behavior Modeling: is performed primarily with Simulink/Stateflow and additionally with the Cyber Modeling Language (inherited from the ESMoL toolsuite). The Cyber Modeling Language (described later) contains constructs equivalent to Simulink Stateflow's integrated dataflow and state machine formalisms, and supports automated import from Simulink/Stateflow models. The Simulink/Stateflow models constitute the core behavioral representation of the controller, however, are agnostic in terms of their software componentization and deployment on a software component framework. Software componentization refers to the packaging of software functions into components, with timed execution threads, and life-cycle management. There is a variety of component frameworks, ranging from complex and comprehensive such as AutoSAR, to emerging ones such as FACE

(defined for Naval Avionics). ESMoL's Cyber Modeling Language, in addition to the dataflow/stateflow behavior formalisms, includes a simple component model that equates a software component to a Real-Time Operating System (RTOS) process, triggered by periodic timed or event-driven alarms, and executes one or more software function associated with the alarm triggers. Thus, after importing controller behavior models from Simulink/Stateflow, engineers partition Simulink/Stateflow behavior models into a set of Software Components. The outcome of the Cyber Behavior Modeling is a collection of Software Components with appropriate causal (input/output) interfaces. These Software components can be packaged as an AVM Component Model, with an ACM descriptor, and are ready to be used in CyPhy for System Design.

System Design Space Authoring (Modeling): is performed in the main CyPhy environment. In this environment, software components are interfaced with physical components or other software components using CyPhy's design and design space representation modeling constructs. The architectural approach from the system design to manufacturing is through Test Benches. A physical manufacturing Test Bench defines an interface to the manufacturing foundry (iFAB) and generates artifacts necessary for iFAB to conduct build activities. Conceptually similar, a software manufacturing Test Bench defines an interface to the Cyber Platform Modeling and Code Generation tools (described below), that trigger the software build activities. The outcome of the System Design Space tools and Test Benches is an AVM Design Model, with an ADM descriptor, that defines the system design, the components used in the design, and their interactions.

Cyber Platform Modeling: is performed in the ESMoL modeling suite. The Platform Modeling Language of ESMoL is used to define the processors, buses, and network topology. The processors and buses are modeled for their core performance characteristics (CPU speed, Bus data rate, etc.) and resource limits (Static/Dynamic Memory size, Stack size, Message packet size, etc.). The Software components are imported from the Cyber Behavior and System Design models. The software components are mapped to RTOS processes (tasks), associated with time and event-triggered alarms, on different processors in the platform model. The interactions between software components for non-collocated components are mapped over to bus messages associated with appropriate communication bus in the network. The outcome of these tools is an integrated platform architecture model that can be used by Code Generation tools.

Software Manufacturing (Code Generation): is performed by three distinct types of code generators: a) behavioral code generation, b) schedule code generation, c) OSEK platform code generation.

The *behavioral code generation* produces C code files from the Simulink and Stateflow models representing the controller behavior. There are two separate code generators included in the ESMoL toolsuite that accomplish this task. A formally described (using graph-rewriting transformations) model transformer maps Stateflow models into a formalizable subset of the C language. The model transformation conforms to the (informally described) operational semantics of Stateflow as documented by Mathworks. Another formally described code generator maps Simulink models to a subset of the C language. This transformation flattens a hierarchical dataflow diagram (Simulink formalism), and performs a topological sort of the

objects in the dataflow. The topologically sorted graph is processed with model template processors that emit code for different Simulink native block types (such as differentiators, integrators, adder, multiplier, etc.). The output of the two behavioral code generator is an integrated top-level initialization function that initializes the internal state variables and data structures, as well as a top-level step function that can be called for each invocation (time-driven or event-driven) of the model defined behavior.

The *schedule code generation* produces an input specification for a scheduling tool, and subsequently translates the derived schedule as an input to the subsequent platform code generator. The input specification for the scheduling tool consists of a list of tasks (processes) for each processor, their worst-case execution time (WCET), and their periodicity. A sporadic server approach is assumed for aperiodic tasks for deriving the schedule i.e. a virtual sporadic server task is included with heuristically defined period as well as WCET. The scheduling tool casts the scheduling problem as a constraint satisfaction problem, and uses a finite-domain constraint solver to derive a satisfying schedule. The output of the scheduling tool is the input specification file augmented with a major period for each processor as well as offset of each task into the major period.

The *OSEK platform code generation* produces a set of configuration files that are used by the platform tools for deploying and executing the operational software. These files include an OIL (OSEK implementation language) file and CAN bus configuration file. The OIL file contains specification for tasks, the main entry point for the task, and a set of alarms triggering the task. The CAN bus configuration file contains the bus messages and the size and timing of the bus messages.
The generated code and configuration files are processed by compiler and platform tools to generate binary object files that can be loaded into the platform processors for operation of the cyber physical system.

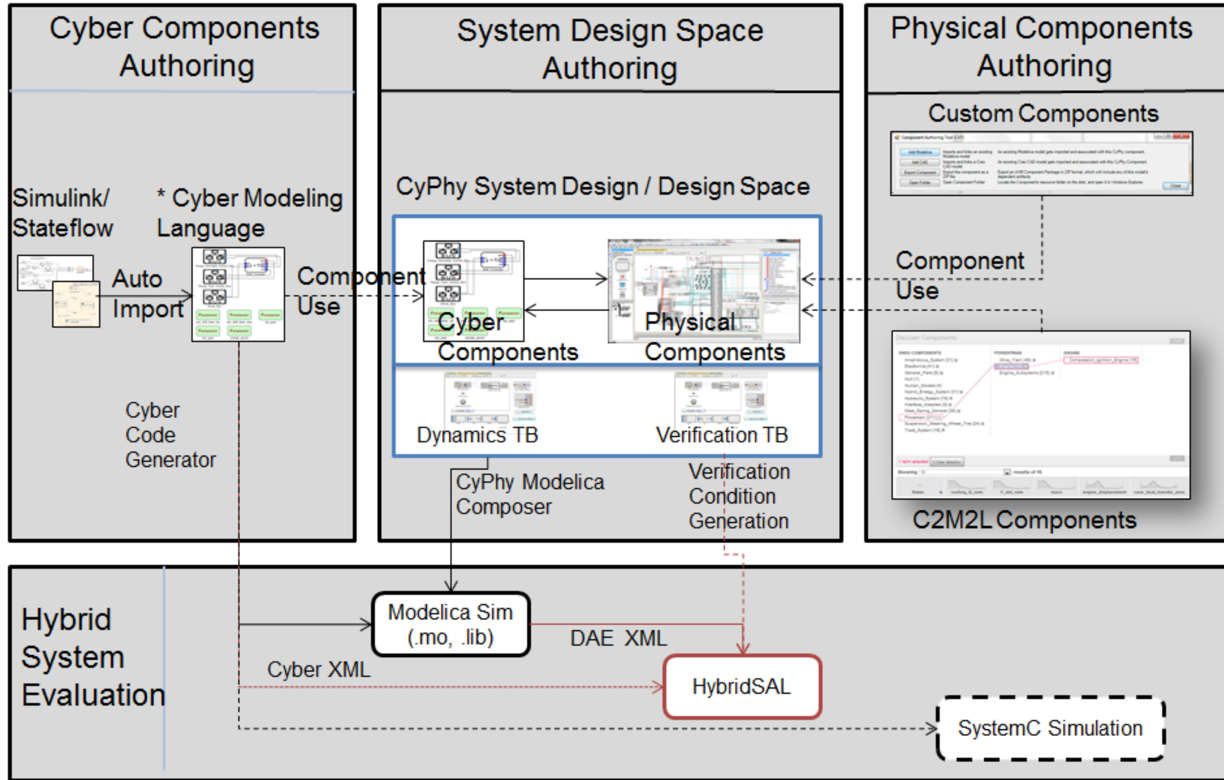

Figure 3: Software Design Tool Architecture for Hybrid System Simulation and Verification

3.2. Hybrid Simulation and Verification Toolchain

Figure 3 depicts the architecture of the software design toolchain for hybrid simulation and verification. The tool architecture can be best explained by partitioning into the following core set of activities: 1) Cyber (Component) Behavior Modeling, 2) System Design Space Authoring (Modeling), 3) Physical Components Authoring (Modeling), and 4) Hybrid System Evaluation (Simulation and Verification). The Cyber Behavior Modeling, and System Design Space Modeling capabilities and roles are similar to the software design and implementation tool architecture explained above.

Physical Components Authoring: is performed in CyPhy and component authoring tools developed in C2M2L and related projects. The physical components dynamics modeling is performed using Modelica language. The outcome of these tools are a set of AVM component models, associated with an ACM descriptor, that are made available to System Design Space Modelers for creating system design.

Hybrid Systems Evaluation (Simulation and Verification): involves two distinct flows with shared design artifacts.

Hybrid Simulation Flow - The CyPhy Dynamics model composer, generates a Modelica model for the System under Test specified through the Dynamics Test Bench. As part of the composition process, when the composer visits a Cyber component, it triggers the invocation of the Cyber behavior code generator (described earlier). This code generator produces the behavior code of the Cyber component (initialization, and step function code). The dynamics composer generates a Modelica wrapper for these cyber components, using a Modelica language

functionality to allow inclusion of external “C” code functions. The resultant Modelica model is processed with Modelica compiler and simulation tools that generate a hybrid simulation of composed physical and cyber system. The results of simulation tools are processed similar to other simulation outputs, and used to extract metrics that are evaluated against requirements
Hybrid Verification Flow - The output of the CyPhy Dynamics composer, Modelica model is processed using the Modelica compiler to generate a Differential Algebraic Expression (DAE), XML representation of the Modelica model. The DAE-XML along with the Cyber behavior model (from ESMoL) is provided directly to a translator that generates a HybridSAL representation of the hybrid dynamics of the cyber-physical system. HybridSAL checks the model against a set of verification conditions, which are specified using CTL/LTL formulae, and can check the model for conformance, or find a counterexample in case the model does not conform to the specified verification criteria.

4. Modeling Languages

The Cyber Design Toolchain consists of multiple modeling suites and languages. In this section we will provide an overview of these languages which are integrated into the ESMoL modeling suite.

The ESMoL is a graphical design modeling language suite which consists of the following languages: (1) dataflow-diagram oriented modeling of signal flows, (2) hierarchical state machine diagrams to model finite-state behavior, (3) software component modeling, (4) hardware topology modeling, and (5) deployment modeling. The first three languages relate to Cyber Behavior Modeling, while the last two relate to Cyber Platform Modeling. These languages are described below in terms of meta-models that capture the abstract syntax of the language.

4.1. Behavior Modeling with Dataflow Diagrams

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 top-level 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. Systems 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. Systems 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).

Figure 4: ESMoL Dataflow Language (Simulink Equivalent)

4.2. Behavior Modeling with Stateflow Diagrams

The Stateflow portion of the meta-model supports the statechart-style modeling of hierarchical finite state machines. 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 ESMoL however, a transition is between TransOutPort and TransInPort of the participating states. 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, ESMoL relies on the use of references, which are effectively pointers to objects that exist elsewhere. 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. See Figure 5 for a view of the ESMoL hierarchical state machine language.

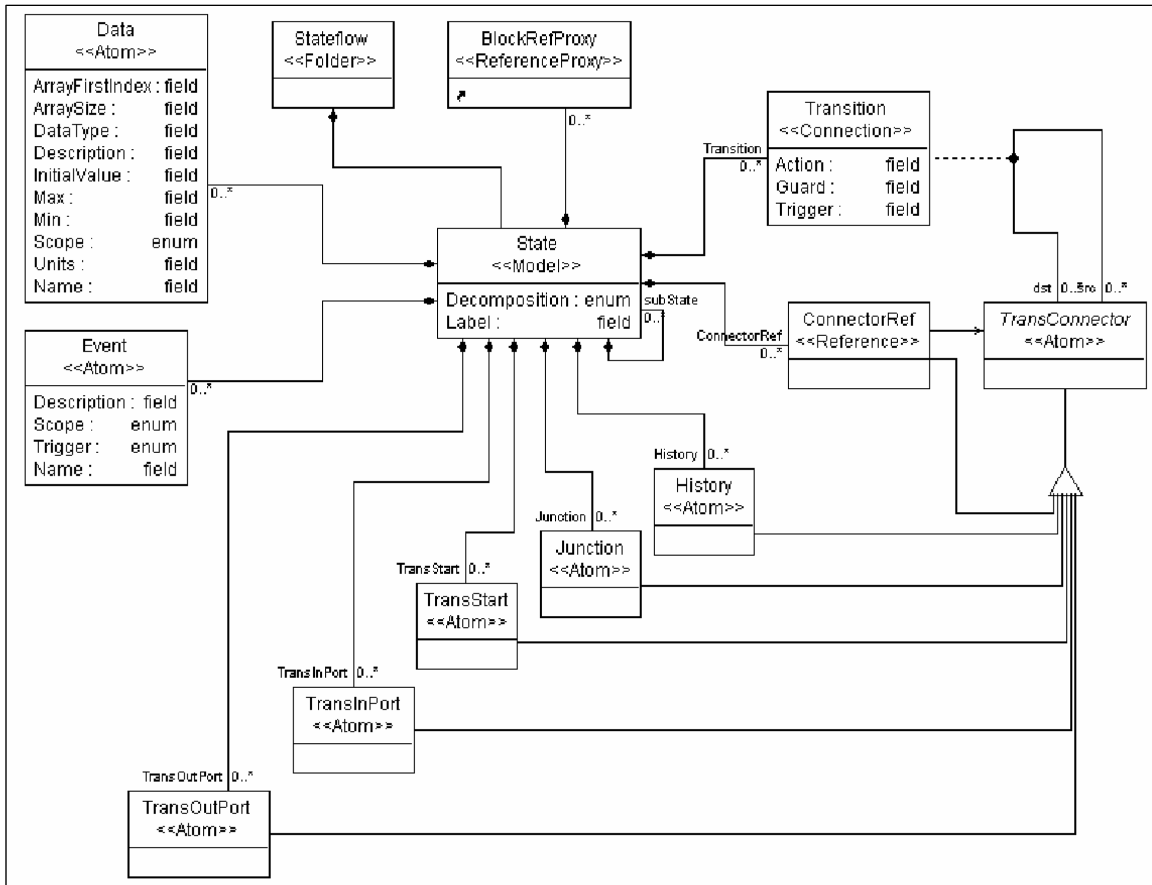

Figure 5: ESMoL Hierarchical State Machine Language (Stateflow Equivalent)

4.3. Software Component Modeling

ESMoL components encapsulate SL/SF Systems (blocks), 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 ESMoL Component is the same as that of the encapsulated System model. Figure 6 shows the Component modeling portion of the ESMoL 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, ESMoL 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 ESMoL: 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 ESMoL 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>>).

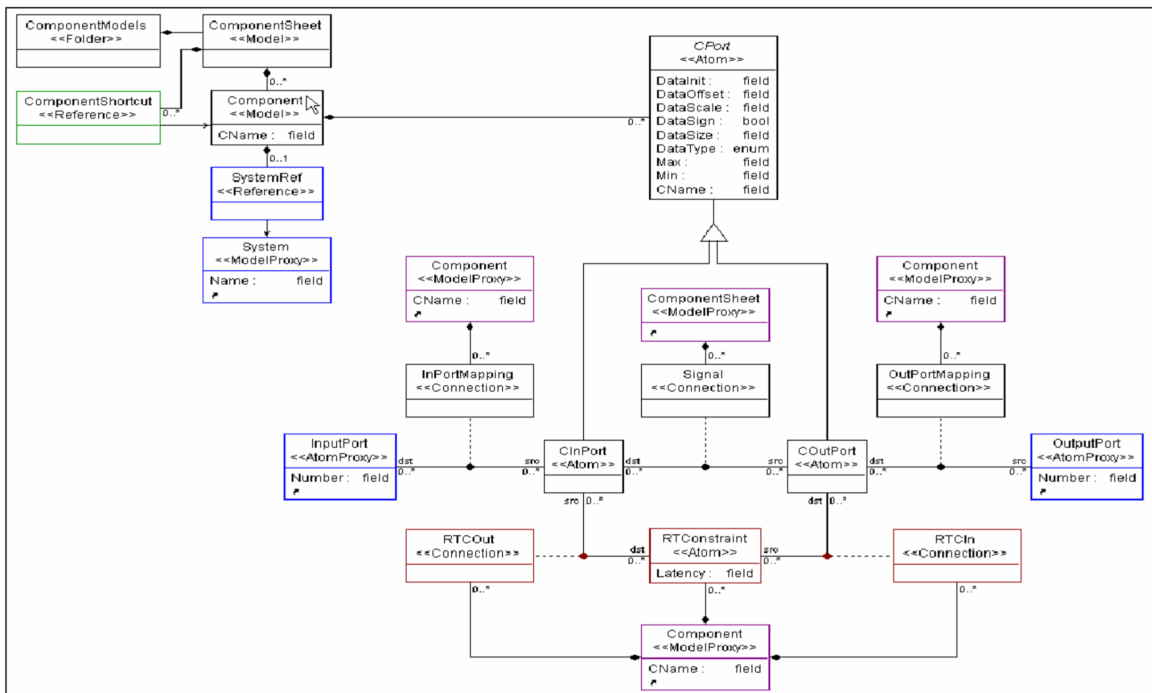

Figure 6: ESMoL Component Modeling

4.4. Hardware Platform Modeling

The hardware modeling sublanguage of ESMoL 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. *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). *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). Figure 7 shows the Hardware Modeling portion of the ESMoL meta-model:

- 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-is, and connections between those. It contains Wheelmen<<FCO>> which is an abstract class, concretized as ECU<<Model>>, Bus<<Atom>>, and Bus Connector<<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 Frame Size 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 Flex Ray, 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 IChat<<Atom>>, Chan<<Atom>>, and Buchan<<Set>>. IChat-s represent sensor ports, Chan-s represent actuator ports, and Buchan-s represent bus connection ports. Channel-s are contained in ECU-s to represent the physical interface of an ECU.
- A Firmware Module<<Atom>> represents a firmware driver that can be attached to a Channel with the Firmware Link<<Connection>>. The Library File attribute of the Firmware Module 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 Source File 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 Event Published 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 Bus Connector<<Connection>> is an association class representing architectural connections between Bus, and Buchan-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 at most 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 Buchan-s, and the association is represented with the Set membership containment relation. This also explains why Buchan-s are stereotyped as Set-s, different from IChat and Chan. 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.

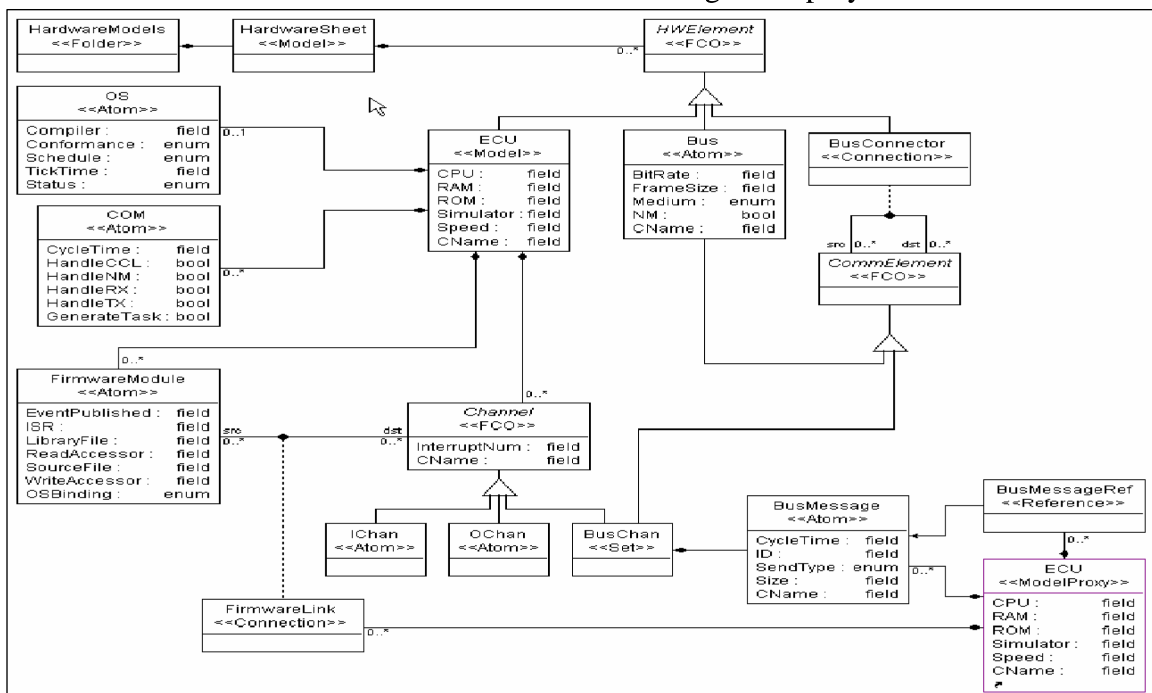

Figure 7: ESMoL Hardware Modeling

4.5. Platform Deployment Modeling

The previous two sections described the (software) component modeling and the hardware modeling sublanguages of ESMoL. This section describes the third ingredient: deployment modeling, which captures how software components are deployed on the hardware. The deployment models capture the mapping (or allocation) of software components onto the

hardware architecture. 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.

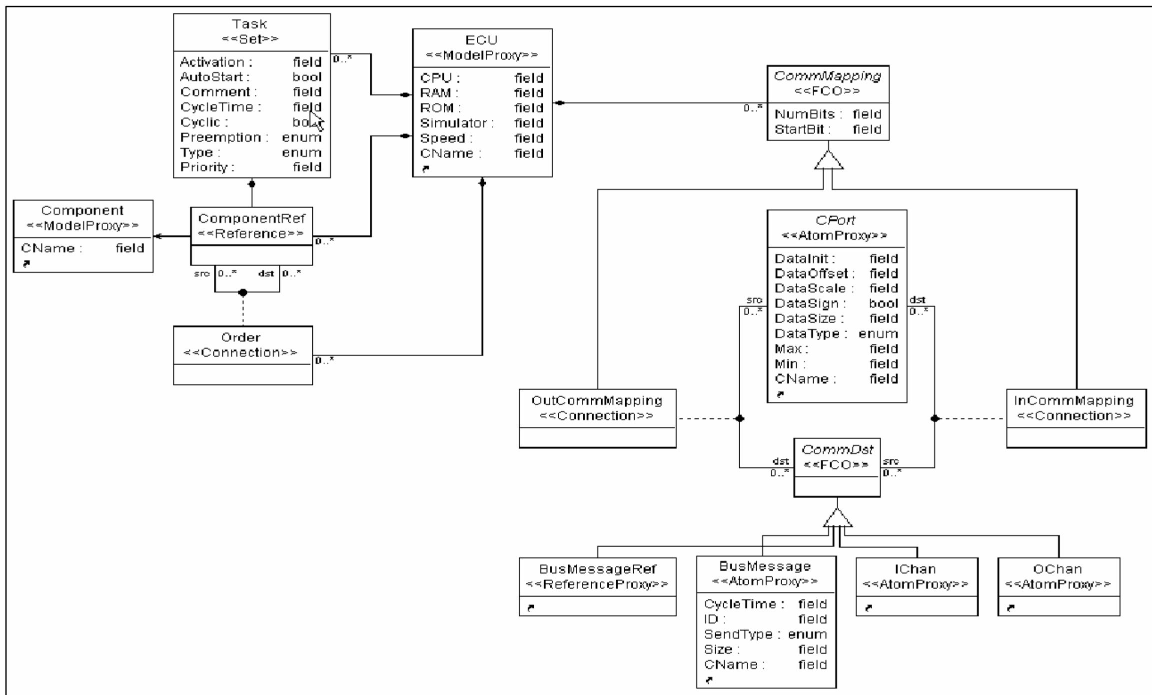

Figure 8: ESMoL Deployment Modeling

Figure 8 shows the Mapping (deployment) modeling portion of the ESMoL 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 participate 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 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 Buchan, 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.

5. Code Generation

5.1. 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 ESMoL. This stage deals with synthesizing implementation from Simulink and Stateflow sublanguage of ESMoL.

5.1.1. 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 ESMoL 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 ESMoL 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 ESMoL 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 ESMoL 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 ESMoL 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 ESMoL 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 ESMoL 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.

5.1.2.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.

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 9 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.

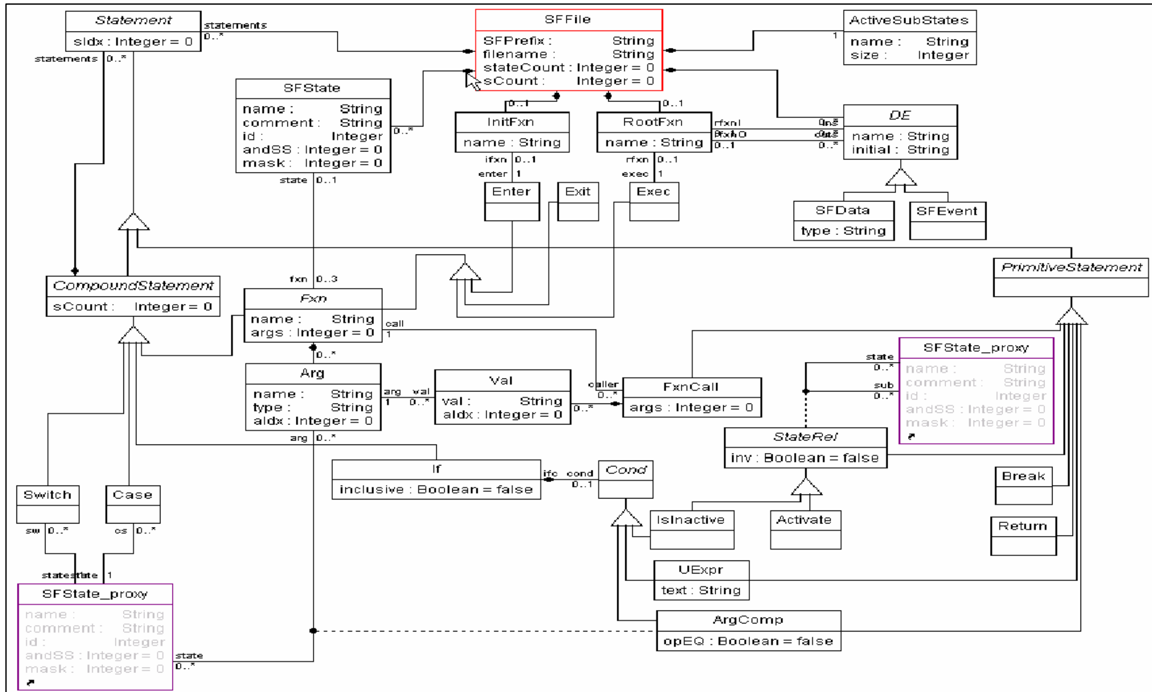

Figure 9: Meta-Model of State-Flow C (SFC)

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 [2].

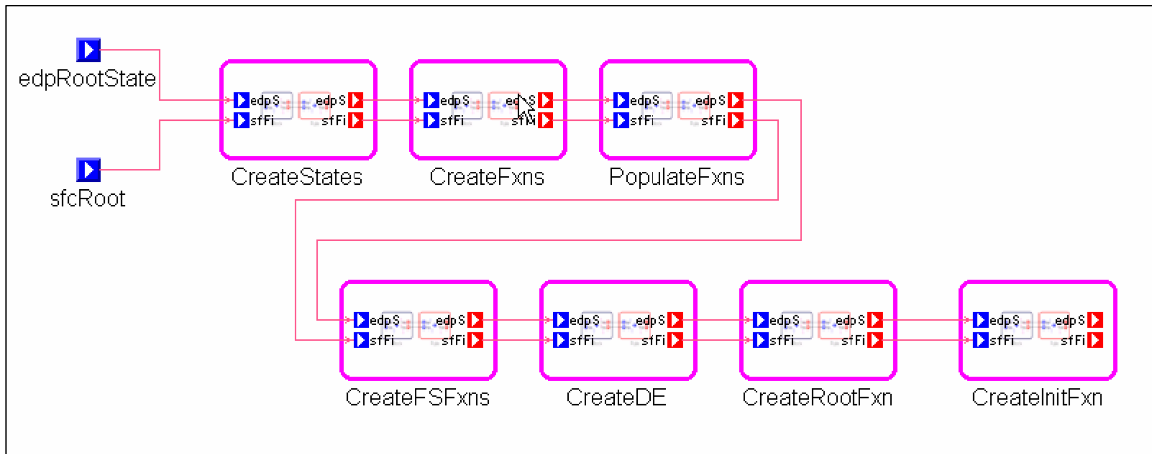

Figure 10: Top-Level ESMoL to SFC Transformation Rule

Figure 10 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 ESMoL 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.

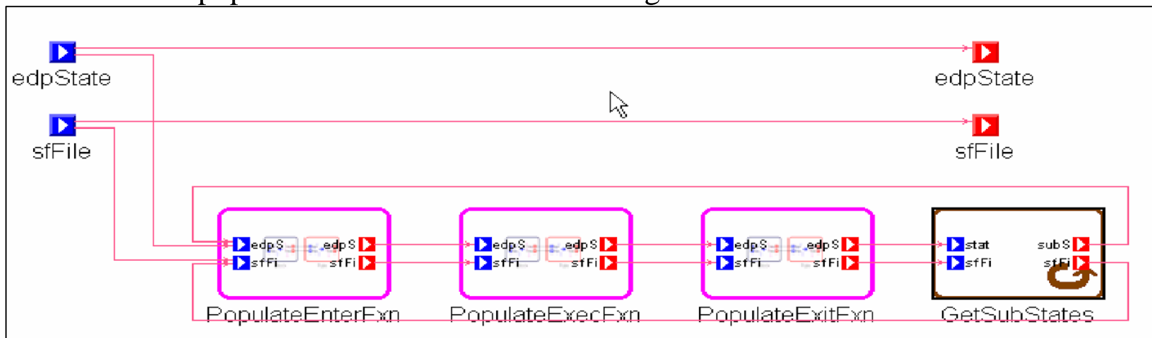

Figure 11: PopulateFxn Rule in the ESMoL to SFC Transformation

Figure 11 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 demonstrates 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.

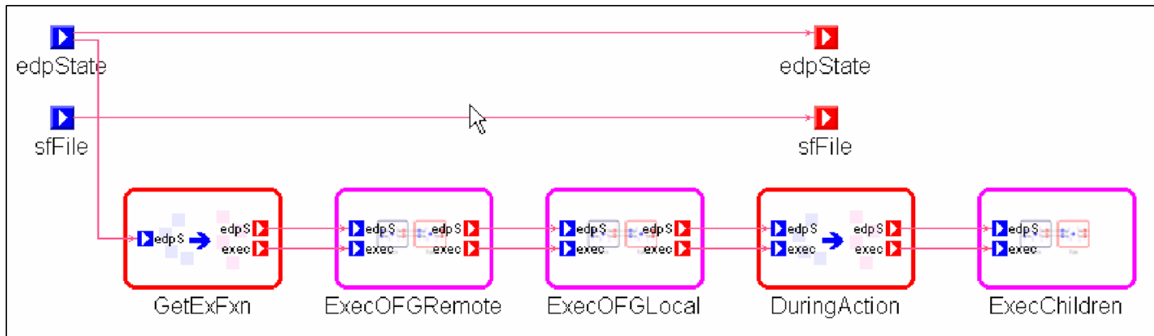

Figure 12: PopulateExecFxn Rule in the ESMoL to SFC Transformation

Figure 12 shows the PopulateExecFxn rule. This rule generates code for the Exec function, 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 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.

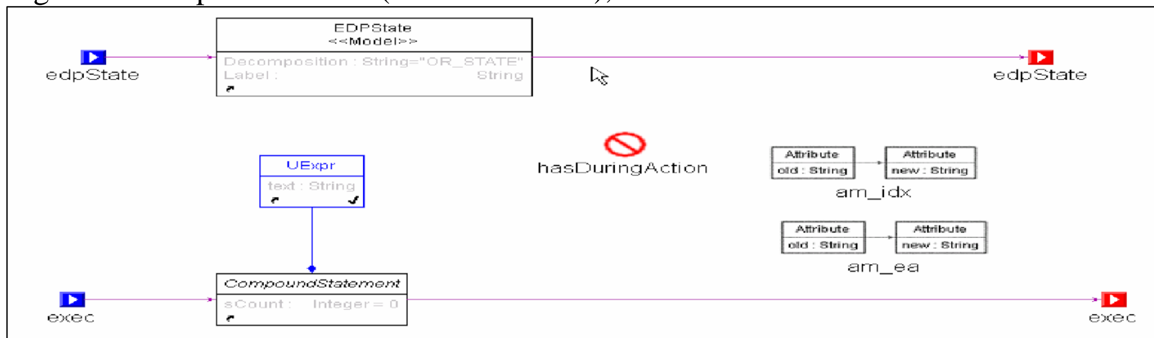

Figure 13: DuringAction Rule in the ESMoL to SFC Transformation

Figure 13 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 ESMoL 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 compliable code. The code is compiled and linked with the other code generation stages to build the complete code generator.

5.2. Platform code generation

The platform code generator component synthesizes code artifacts necessary for system implementation. These include:

- 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.

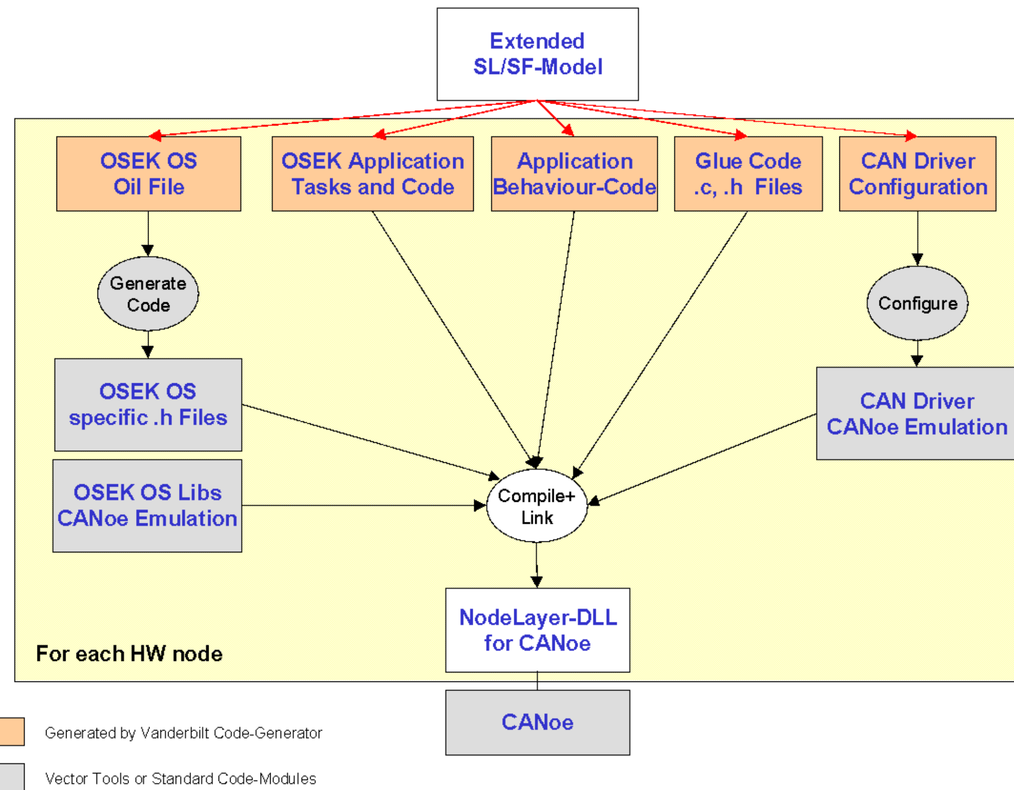

Figure 14: Platform Code Generation Artifacts

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 (see Figure 14). 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.

5.2.1. 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 ESMoL 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 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 ESMoL 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 IChat and Chan 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.

5.2.2. 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 ESMoL 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 ESMoL 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 is bound with Alarm-triggered Events that are associated with Network Management messages, CCL, Receive and Transmit.

5.2.3. 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 ESMoL 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 IChat, 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

5.2.4. 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

6. Scheduling

The Cyber Design toolchain relies on constraint-based schedule solver for computing a periodic time triggered schedule. This section describes the code generator that creates the input for the scheduler.

6.1. TT Schedule Generation

The time-triggered schedule generator uses the scheduling information inside an ESMoL model to generate a time-triggered schedule. In the first step, the generator traverses the ESMoL model and obtains the relevant scheduling attributes from the software components. Then, the generator uses this information to formulate a constraint satisfaction problem. This constraint satisfaction problem is then given to a solver, and if a solution is found, the generator translates the solution into a valid time-triggered schedule for the components in the model.

Figure 15 below shows how scheduling information is assigned to components in an ESMoL model, and Figure 16 shows the scheduling attributes of the TTExecInfo1 object.

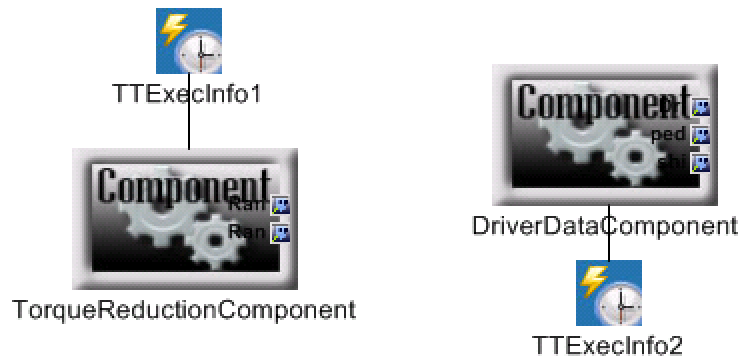

Figure 15: Assigning Scheduling Information to Components

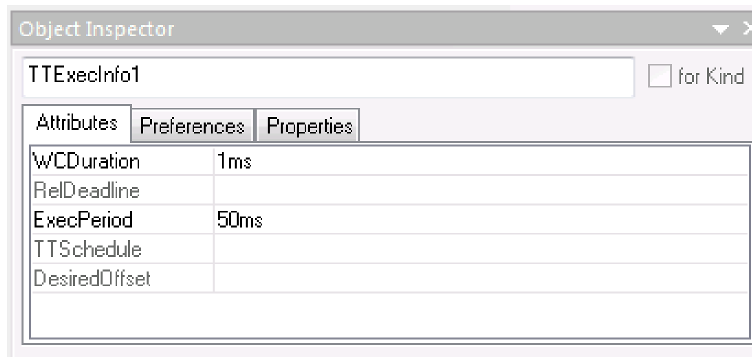

Figure 16: The Time-Triggered Scheduling Attributes of the TTExecInfo1 Object in Figure 15

7. Interface to Physical Dynamics

The interface to physical dynamics requires integrating the Cyber generated code with the physical dynamics specified in Modelica models. The Dynamics Test Bench and tools generate the Modelica models for the entire system as well as for the Test Bench. In the generated Modelica models, the Cyber components manifest as a Modelica model with an appropriate causal interface i.e. input and output signals. This Modelica model is essentially a wrapper that invokes the generated Cyber code based on a defined sampling period. Modelica language allows invocation of triggered function using a *when* construct. The example below illustrates the generated Modelica wrapper and invocation of a cyber-generated Shift controller code (see Figure 17).

```
model ShiftController_type

ShiftController_wrapper tcontext =
ShiftController_wrapper(sample_period, min_shift_timer_std,
min_shift_timer_extd, low_gear, top_gear);

public
      C2M2L_Ext.Interfaces.Context_Interfaces.Driver.Driver_Bus
Driver_Bus;
      C2M2L_Ext.C2M2L_Component_Building_Blocks.Drive_Line.Torque_Conver
ters.Common_Controls.Torque_Converter_Control_Bus
Torque_Converter_Control_Bus;
      C2M2L_Ext.C2M2L_Component_Building_Blocks.Drive_Line.Range_Packs.C
ommon_Controls.Range_Pack_Control_Bus Range_Pack_Control_Bus;

      parameter Real sample_period;
      parameter Real min_shift_timer_std;
      parameter Real min_shift_timer_extd;
      parameter Real low_gear;
      parameter Real top_gear;

      output Boolean sampleTrigger;

equation
      sampleTrigger = sample( 0, sample_period);
      when sampleTrigger then
            Range_Pack_Control_Bus.gear_selected =
                  ShiftController_wrapper_main(
            tcontext,Driver_Bus.gear,Torque_Converter_Control_Bus.input
            _speed_torque_converter,Torque_Converter_Control_Bus.output
            _speed_torque_converter,Range_Pack_Control_Bus.shift_reques
            ted,Driver_Bus.gear);
      end when;

end ShiftController_type;
```

Figure 17: Modelica Wrapper Code generated for Cyber Controller

The wrapper object is instantiated with a call to Shift_Controller_wrapper class constructor. This initializes the data structures of the cyber generated code, and stores the data structures in the tcontext variable in Modelica model. The Boolean variable sampleTrigger is set by using the Modelica built-in sample function, which returns a true when the simulation time matches with the sample period. This Boolean variable is used as a trigger to invoke the main function of the ShiftController wrapper. The main function is the top level step function of the behavioral code which executes one step of the controller behavior and computes the value of the output signal given the current state of the controller behavior (as stored in tcontext), and the value of the input signals.

A simulation of this integrated Modelica model produces a hybrid cosimulation of the physical model as well as the Cyber controller. It should be noted that this cosimulation assumes an idealized (or synchronous) implementation of the controller, i.e., the step function takes zero time to execute. A non-ideal implementation would take a finite non-zero time to execute, and in case of a real-time constraint violation (i.e., when execution of the current step does not finish within the sample period) causing instability in the controller behavior. The simulation of the performance of the controller in presence of timing constraints is referred to as True Time analysis, and is facilitated by a different simulation in the Cyber toolchain.

8. Examples

This section will provide a small case-study of using the Cyber Design Evaluation capability of the OpenMETA tools. The example illustrated is the Drive Line model that is analogous to the FANG vehicle architecture. In this driveline mode the shift controller that is responsible for changing the gear of the vehicle automatically based on the speed of the vehicle and torque demand is implemented as a Cyber controller using a Stateflow model. The description below presents the Cyber component model, shows its integration in the CyPhy System model, shows the Cyber platform model on which the controller will be deployed, and then illustrates generated artifacts as well as results from a Hybrid dynamics cosimulation, and finally illustrates artifacts generated for the platform software implementation.

8.1. Cyber Component Model

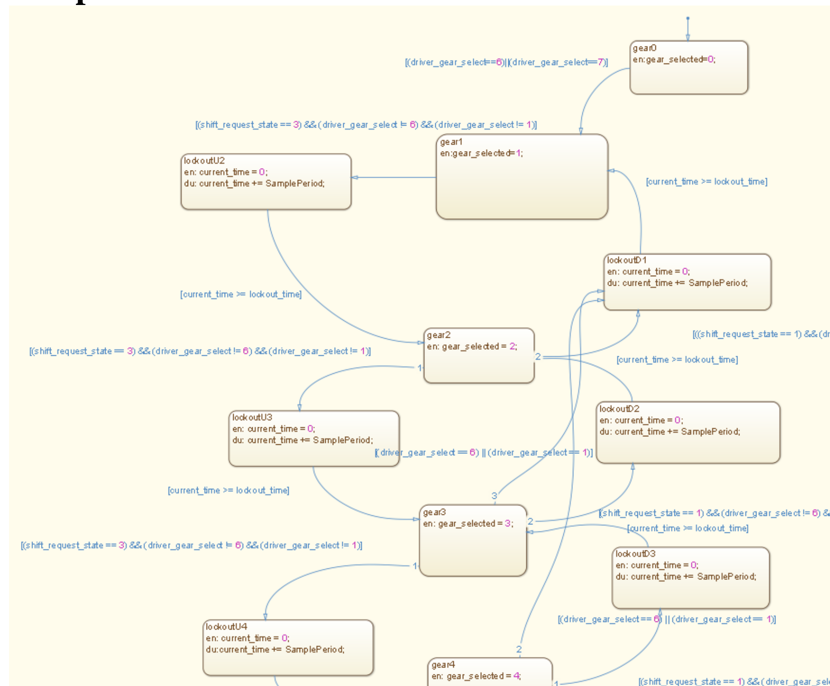

Figure 18 : Stateflow Model of Shift Controller

Figure 18 shows a Stateflow model of the shift controller. The controller has multiple states corresponding to the current gear of the vehicle, as well as the intermediate torque converter lock out states when the controller initiates a gear shift. The inputs to the controller are shift requests, and output from the controller is the gear selected. We refrain from discussing the details of the control algorithm as it is outside the scope of this report.

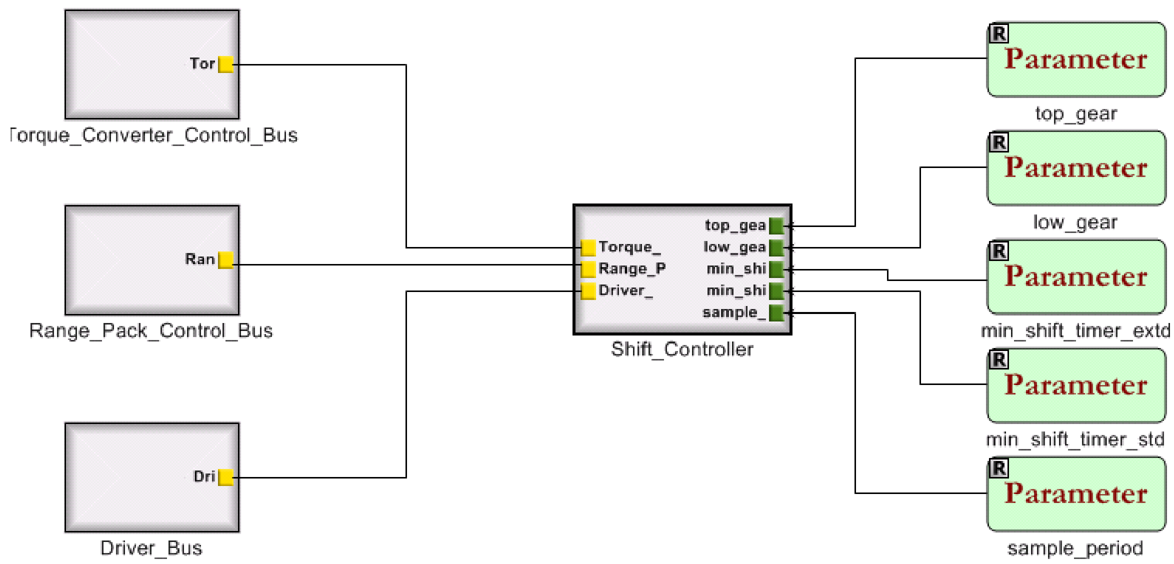

Figure 19: CyPhy Model of the Shift Controller Component

Figure 19 shows the Stateflow controller imported in CyPhy as an AVM component. The controller model is shown as an embedded Domain model in a CyPhy component wrapper, with the appropriate signal and parameter interface mapped into the Domain model. After this wrapping this Cyber component can be used as any other component in a CyPhy system design model.

8.2. System Model

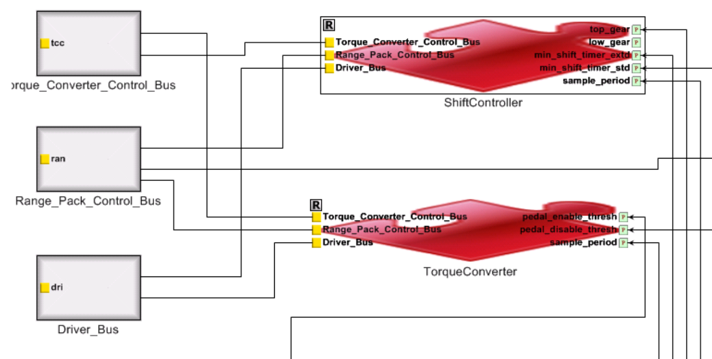

Figure 20: Torque Control Unit Subsystem in Drivetrain Model

In the System model the Shift Controller component is included as part of a Torque Control Unit (TCU) subsystem. Figure 20 depicts a view of the TCU assembly that shows the ShiftController as well as a TorqueConverter component.

Figure 21: Subset of the Drivetrain Assembly Model

The TCU assembly (labeled as TCU_Cyber in Figure 21) is wired into the rest of the driveline model. The details of the driveline model are outside the scope of this chapter.

8.3. Cyber Platform Model

The Cyber platform model consists of a mapping of the Shift controller component to a computing platform.

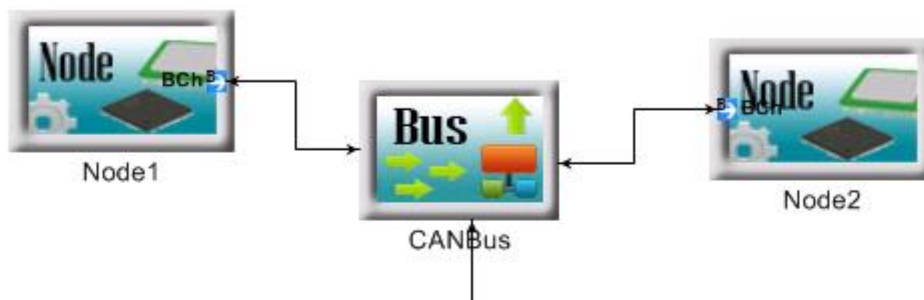

Figure 22: Platform Model for Cyber controllers of the Drivetrain System

Figure 22 shows a simple platform mode, with two processor nodes connected by CAN Bus. The Nodes and the CAN Bus is characterized with respect to its performance and capacity.

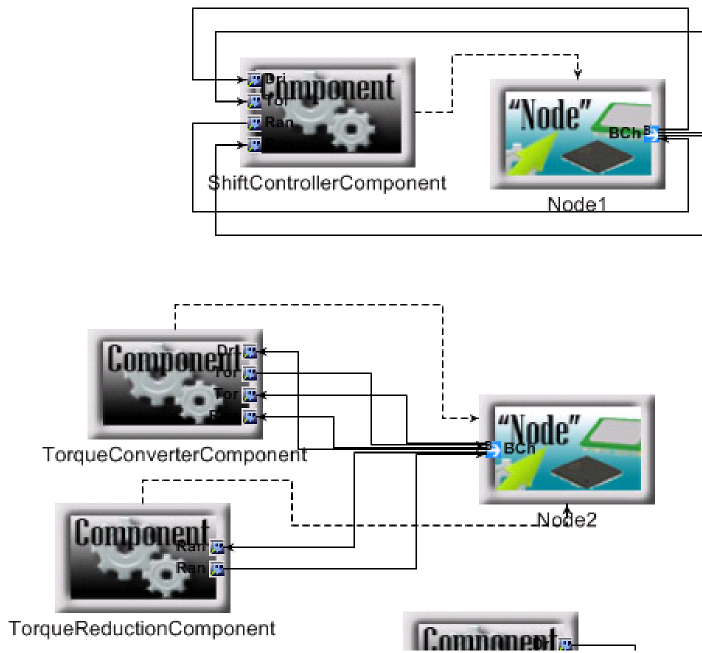

Figure 23: Mapping and Deployment Model of the Shift Controller Drivetrain System Controller Platform

Figure 23 shows mapping of the ShiftController component on processing Node1 (with a dashed arrow), while the solid arrows indicate the mapping of component interface (data objects) onto bus messages (represented with BChan port).

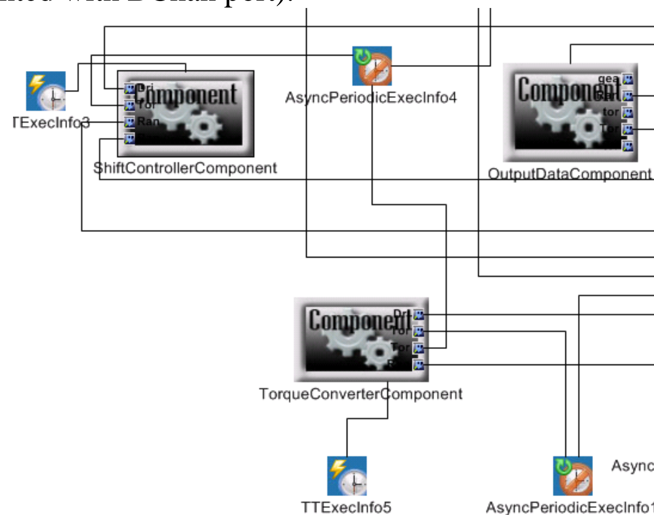

Figure 24: Execution view of the Shift Controller with Task and Timing Association

Figure 24 shows an execution view of the platform model that associates components with schedule objects. These schedule objects are populated after an execution of the scheduling tool.

8.4. Dynamics Evaluation

The dynamics evaluation is specified with a Dynamics Test Bench; a full speed on flat terrain Test Bench is illustrated in Figure 25.

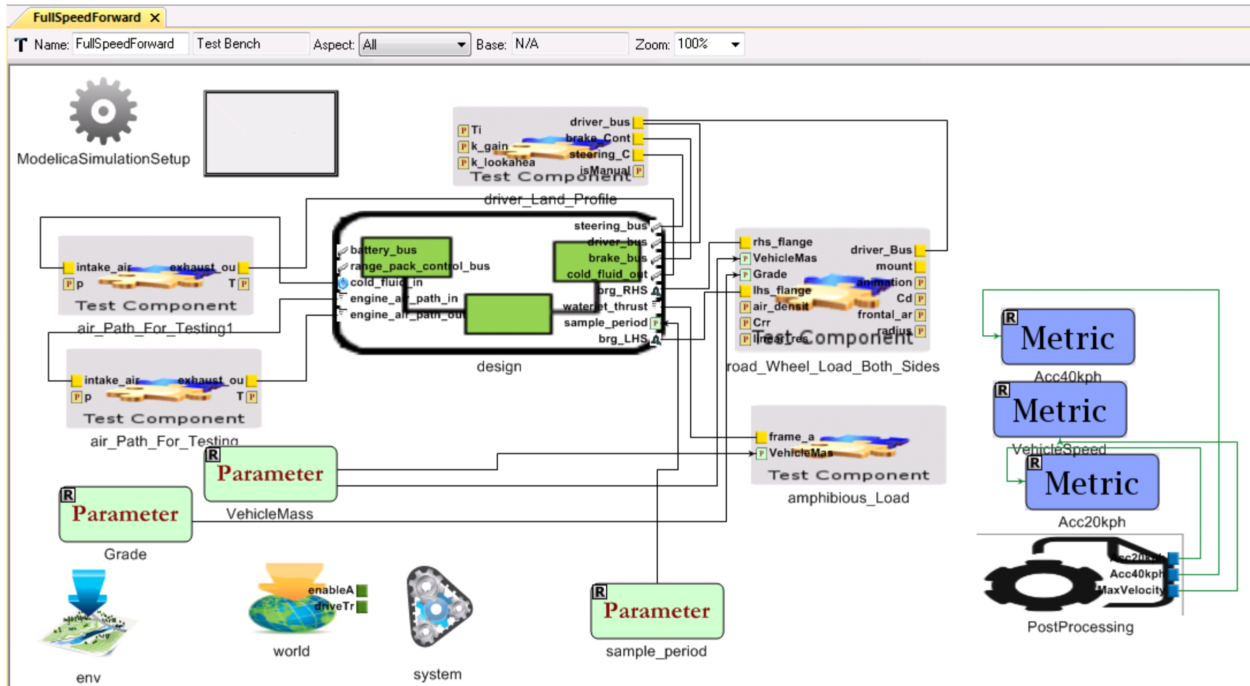

Figure 25: Dynamics Test Bench for Full Speed Test

The Modelica composer generates the Modelica model as shown in Figure 26:

Figure 26: Modelica Model generated from Dynamics Test Bench for Full Speed Test

The generated Modelica wrapper model was illustrated in **Error! Reference source not found.**

Figure 27: Generated Cyber Behavioral and Wrapper code

Figure 27 shows the generated Cyber code that shows the core Stateflow behavioral code, Simulink behavioral code, and wrapper code. Compiler make and project files are also generated that renders it easy for the generated code to be compiled and linked into Modelica models.

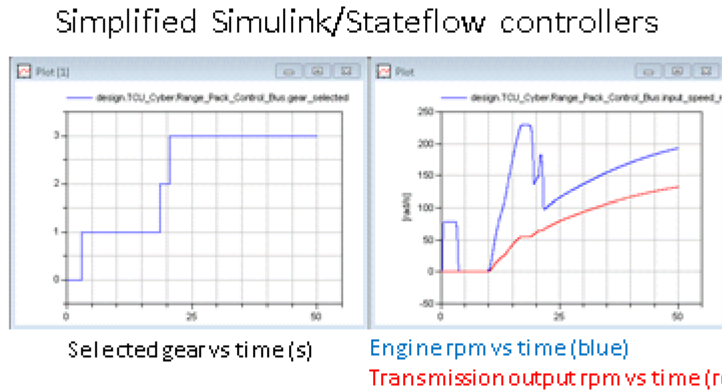

Figure 28: Simulation Results for Full Speed Test showing gear selected, engine RPM and transmission RPM

The results of the dynamics evaluation are shown in Figure 28, which shows the selected gear over time, as well as the engine RPM and transmission output RPM over time.

8.5. Platform Software Synthesis

The platform software includes the behavior code files depicted above as well as component and platform code files.

Node1.oil

```
247     };
248   };
249   TASK ShiftControllerComponent {
250     TYPE = AUTO;
251     SCHEDULE = ;
252     PRIORITY = 0;
253     ACTIVATION = 1;
254     AUTOSTART = TRUE
255     {
256       APPMODE = OSDEFAULTAPPMODE;
257     };
258     EVENT =
        ShiftControllerComponent_event;
259     Procedure = EventProcedure
260     {
261       EVENT =
          ShiftControllerComponent_event;
262       Name =
          "ShiftControllerComponent_proc";
263     };
264   };
265   TASK TorqueReductionComponent {
266     TYPE = AUTO;
267     SCHEDULE = ;
268     PRIORITY = 0;
269     ACTIVATION = 1;
270     AUTOSTART = TRUE
271     {
272       APPMODE = OSDEFAULTAPPMODE;
273     };
274     EVENT =
        TorqueReductionComponent_event;
275     Procedure = EventProcedure
276     {
277       EVENT =
          TorqueReductionComponent_event;
278       Name =
          "TorqueReductionComponent_proc";
```

Node2.oil

```
1   OIL_VERSION = "2.3";
2
3   #include <VecCanol.i23>
4   CPU Node2 {
5     #include <VecCanol.s23>
6     OS StdOS {
7       CC = AUTO;
8       STATUS = EXTENDED;
9       SCHEDULE = AUTO;
10      STARTUPHOOK = FALSE;
11      ERRORHOOK = TRUE;
12      SHUTDOWNHOOK = FALSE;
13      PRETASKHOOK = FALSE;
14      POSTTASKHOOK = FALSE;
15      USEGETSERVICEID = FALSE;
16      USEPARAMETERACCESS = FALSE;
17      TickTime = 0;
18    }: " no comments ";
19
20    COM StdCOM {
21      USEMESSAGERESOURCE = TRUE;
22      USEMESSAGESTATUS = TRUE;
23    }: " no comments ";
24
25    EVENT OutputDataComponent_event {
26      MASK = AUTO;
27    };
28    EVENT DriverDataComponent_event {
29      MASK = AUTO;
30    };
31    EVENT TorqueConverterComponent_event {
32      MASK = AUTO;
33    };
34    EVENT TorqueDataComponent_event {
35      MASK = AUTO;
36    };
37    EVENT RangePackDataComponent_event {
38      MASK = AUTO;
```

Figure 29: Generated Platform Code – OIL Files

Figure 29 depicts the OIL files for the two processor nodes. The OIL file shows the configuration of the ShiftController component task, its properties including scheduling priority, triggering event, and main procedure. The OSEK compiler instantiates the task based on this specification.

9. Future Work

The Cyber tools are comprehensive in terms of functionality: they can generate platform code, schedule analysis, behavioral code, and target hybrid systems verification. However, some of the limitations relate to the workflow as it spans multiple tools with their own methodology.

A controller is typically designed once a plant model is available (or created in Simulink), however, in OpenMETA we rely on concurrent development of controller model while we design, explore and evolve the plant model. Facilitating this co-evolution of plant with controller will require an iterative flow between Simulink and OpenMETA. Such a flow can be enabled by use of FMU (or Simulink S-function) that can capture the plant behavior and make it available in Simulink to facilitate refinement of controller design. We already support import of Simulink models in OpenMETA.

The platform modeling capability of the Cyber toolchain is somewhat standalone and loosely integrated with OpenMETA requiring manual efforts to map the Cyber models into a standalone platform model. Ideally, OpenMETA should capture platform models as they are physical

components and constitute electrical and thermal loads that must be analyzed together with the rest of the CPS. In the future we plan to streamline this integration and facilitate a direct mapping from OpenMETA into the ESMoL platform modeling tools

The Cyber platform component models are currently designed towards the OSEK component model. There are newer component models that can support a more dynamic deployment as well as resilience. In the future we plan to augment the Cyber platform modeling capability with additional component models.

Bibliography

1. Porter J., Karsai G., Sztipanovits J.: Towards a time-triggered schedule calculation tool to support model-based embedded software design. In: EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded Software, New York, NT, USA, 2009, pp 167-176.
2. Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: Proc. Int'l Conf. Engineering of Computer-Based Systems, IEEE Computer Society (2003), pp 159–168.