Architecture Driven Generation of Distributed Embedded Software from Functional Models

Gopal Raghav,
Swaminathan Gopalswamy,
Karthikeyan Radhakrishnan
Emmeskay, Inc.
Plymouth, MI
{gopalanr, swami, rg-karthikeyan}@emmeskay.com

Julien Delange,
Jérôme Hugues
Telecom ParisTech
Paris, France

{delange, hugues}@enst.fr

ABSTRACT

Embedded systems are becoming increasingly complex and more distributed. Cost and quality requirements necessitate reuse of the functional software components for multiple deployment architectures. An important step is the allocation of software components to hardware. During this process the differences between the hardware and application software architectures must be reconciled. In this paper we discuss an architecture driven approach involving model-based techniques to resolve these differences and integrate hardware and software components. The system architecture serves as the underpinning based on which distributed real-time components can be generated. Generation of various embedded system architectures using the same functional architecture is discussed. The approach leverages the following technologies – IME (Integrated Modeling Environment), the SAE AADL (Architecture Analysis and Design Language), and Ocarina. The approach is illustrated using the electronic throttle control system as a case study.

1. INTRODUCTION

Embedded systems in ground vehicles are becoming increasingly complex in the functionality they support. Safety and security are very critical. Innovative approaches are needed to develop such systems efficiently without compromising on quality. A growing trend in development of complex embedded systems is the use of model-based development (MBD) techniques. Essentially MBD involves modeling the behavior of the embedded systems to enable simulation of the embedded system performance for various stimuli under various operating conditions. MBD supported by CAE tools facilitates the design of advanced control functionality by enabling early V&V before the mechanical and electronic hardware become available. The current state of MBD technologies is evolved enough to allow embedded software to be automatically generated from the functional models. Such tools and processes facilitate code generation for a single ECU. In practice however, as the number of processors and complexity of algorithms keep growing, there are two critical needs that emerge:

(i) The development framework needs to support modular development of embedded software promoting re-usability. Further, we need to support multiple variants in the implementation of re-usable components. This leads to the idea of an “architecture” becoming the underpinning description of a system, with variant management built around this architecture – Architecture Driven Development (ADD). These issues have already been addressed in previous work, e.g. [1]

(ii) The second need is related to the fact that the functional model of an application often has a very different architecture from the architecture of the application embedded software. The functional architecture of the system corresponds to the optimum architecture required for control system development, and is concerned with the functional performance of the physical system being controlled. On the other hand, the embedded systems architecture required for any application is concerned with the number of processors, the different tasks and threads and their scheduling, etc. We need an approach that explicitly attempts to resolve such differences.

The focus of this paper is to discuss the second need above. In particular, this paper discusses an approach to extending the current technologies to allow generation of distributed embedded software from functional models, seamlessly reconciling the differences between functional architectures and embedded systems architecture.

In addition to the above consideration, a single functional architecture could often support multiple embedded systems architectures. For example, there could be technological advancements in the hardware used in the system, requiring new hardware architectures even though the functional architecture does not change.

It is also common to reuse software components between different vehicles where the functional architecture might change but the hardware architecture remains the same.

The approach discussed in this paper also addresses such additional practical issues that are of relevance during the development of complex and large embedded systems.

Section 2 discusses the architecture driven development approach for generation of distributed software. Two major activities are described in this section – (a) development of functional models consistent with a system architecture and (b) generation of distributed embedded software. Workflows to perform these activities are discussed. Section 3 discusses enabling tools and technologies that are leveraged in this process. Three major technologies are leveraged in our work – the SAE AADL (Architecture Analysis and Design Language), IME (Integrated Modeling Environment) and Ocarina (AADL toolsuite with code generation facilities). Section 4 describes a case study using the electronic throttle control application. Finally in section 5 we summarize our findings.

2. ARCHITECTURE DRIVEN APPROACH

The system architecture can be used as the underpinning using which functional models as well as embedded system models and software can be generated. The approach primarily consists of two major activities. First the system architecture needs to be defined and functional models (Executable Specifications) be developed that are consistent with the system architecture. Second the functional models should be integrated with the hardware architecture and embedded software generated from the complete system model.

2.1. Generation of Functional Models

The Generation of Functional Models can be captured through four key steps in our proposed approach:

(i) Functional Architecture Definition: One of the first steps is to develop the system architecture. The architecture is the topology of the system and describes the structural hierarchy of the subsystems and their interfaces and connections. Several stakeholders are involved in this step – control engineers, software engineers and managers. Usually this step is performed by the OEM based on product goals and requirements. This architecture can then be used to communicate the requirements to the suppliers of the individual components in the system, who could be a division of the OEM or an outside supplier.

(ii) Organize and Mine Component Functional Models Repository: A step that happens in tandem with the architecture is the development of component models. To enable maximum reuse, modular component models are developed over time by the organization and collected in a repository that is accessible to all authorized developers. The larger the repository of such models the quicker the development of functional applications. However, it is important to have the repository well organized and searchable. In particular, being able to search the repository based on architectural metadata of the component models (such as their interfaces, hierarchy, etc) will dramatically improve the efficiency of application development.

(iii) Associate Component Models to Architecture: Since we want the application models to be consistent with the system architecture, we need to identify component models that fit into the corporate hierarchy. As part of such an association, the ability to search the model repository for architectural metadata should be leveraged to verify architectural consistency. Both structural and interface consistency must be performed. Additionally constraints specified in the architecture can be combined with metadata inserted into the component models to enable guided searches of the model repository.

(iv) Generate Simulatable Application Models: The final task is the generation of simulatable application models that can be used for confirming the functionality of the control system. As an example, from the system architecture we can compose models in a simulation domain such as Simulink [2].

Some of the steps in the generation of functional models are shown in Figure 1.

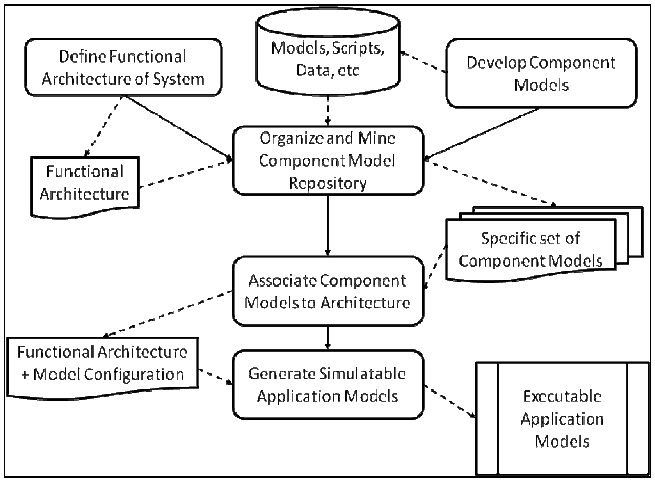

Define Functional Architecture of System
Models, Scripts, Data, etc
Develop Component Models
Organize and Mine Component Model Repository
Functional Architecture
Specific set of Component Models
Associate Component Models to Architecture
Functional Architecture + Model Configuration
Generate Simulatable Application Models
Executable Application Models

Figure 1: Process Steps to Generating (executable) Application Models from Functional Architecture

2.2. Generation of Distributed Embedded Software

The generation of Distributed Embedded Software can be captured in these next steps:

(i) Embedded Systems Architecture Definition: The embedded system architecture is concerned with the processors in the system, the information communicated between them, the processes and threads within each of the processor, and the scheduling of those processes and threads. AADL is a powerful mechanism for describing and communicating such an architecture [3]. Thus the embedded systems architecture is developed by the stake holders consisting of the program managers, and the embedded software and hardware engineering team.

(ii) Reconciliation of Functional Architecture with Embedded Systems Architecture: It is important to recognize that the architecture used to generate functional models is often quite different from the embedded system (hardware) architecture. What is common between the functional architecture and the embedded systems architecture is the set of component functional models associated with either architecture. Thus, in order to reconcile the functional architecture with the embedded systems architecture, a key step is to define the bindings between the component functional models in the functional architecture and the nodes of the embedded systems architecture.

When such a binding is done, it is important to ensure that the functional connectivity between the software components as specified in the functional architecture is not broken, while simultaneously the communication requirements of the embedded systems architecture is satisfied.

(iii) Generation of Distributed Embedded Software: The final step in this process is to generate the distributed embedded software based on the embedded systems architecture and the associated component models.

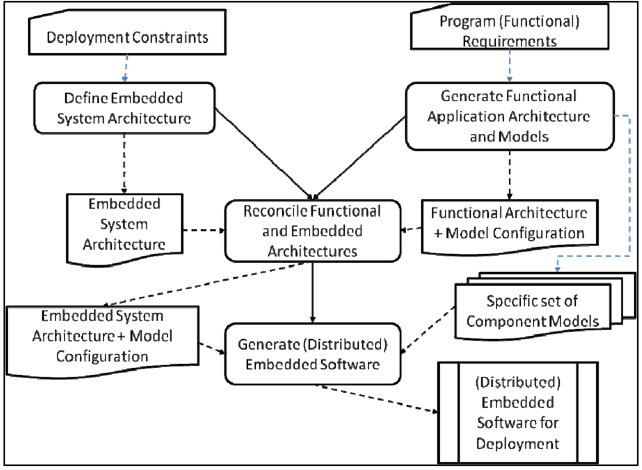

Deployment Constraints
Program (Functional) Requirements
Define Embedded System Architecture
Generate Functional Application Architecture and Models
Embedded System Architecture
Reconcile Functional and Embedded Architectures
Functional Architecture + Model Configuration
Specific set of Component Models
Embedded System Architecture + Model Configuration
Generate (Distributed) Embedded Software
(Distributed) Embedded Software for Deployment

Figure 2: Process Steps in generating (Distributed) Embedded Software for Deployment

This workflow is shown in Figure 2 above. If an automated process exists to generate the distributed embedded software from the architecture, then this provides a powerful methodology to generate different deployment variants of the distributed embedded system, all starting from the same functional architecture. The different deployments could pose different constraints, and these constraints could be accommodated easily by the appropriate binding between the embedded architecture and functional architecture. (see Figure 3 below)

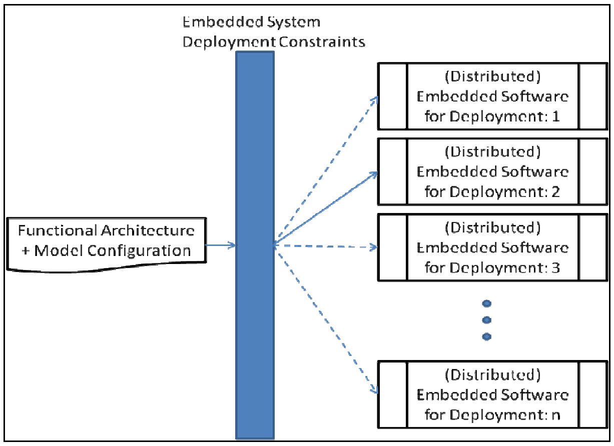

Embedded System Deployment Constraints
Functional Architecture + Model Configuration
(Distributed) Embedded Software for Deployment: 1
(Distributed) Embedded Software for Deployment: 2
(Distributed) Embedded Software for Deployment: 3
(Distributed) Embedded Software for Deployment: n

Figure 3: A single Functional Architecture can support Multiple Deployment Architectures

An example of multiple deployments for a given functional architecture could be as below: (i) A low-cost solution dictates that the entire application is deployed within a single processor, accepting the associated schedulability issues, and consequent performance pull-back. (ii) A medium-cost solution that allows for smart actuators and/or smart sensors to run some of the functionality in local processors, enabling faster internal feedback, and consequent improved system performance; the bulk of the functionality still runs in a central processor. (iii) A high-cost system that calls for separate processors to provide monitoring, safety and redundancy functionalities. By using the advocated approach, each of the deployment could be initiated conveniently from a common functional architecture.

3. ENABLING TECHNOLOGIES

In the previous section we discussed how the system architecture drives the various activities involved in the embedded software generation process. Several enabling technologies are needed to efficiently realize the processes:

Architecture Description Methodology: In order to perform reconciliation between the functional and hardware architectures a first requirement is that we need to be able to have standardized descriptions of both the architectures. Such an Architecture Description Language (ADL) needs to allow specification of several different components (functional, software, electronics, sensors and actuators) as well as the communication between them. Along with the architectural view that lists the topology of the system, non-functional properties of relevance (priority, time slot for CAN buses, memory capacity, etc.) need to be specified at the different nodes of the architecture. There is also a need for a well defined mechanism to bind the functional components to the embedded system components while still retaining both their properties.

We chose the SAE AADL as the backbone modeling notation because it has sufficient richness in definition to be a good language for architecture driven development of embedded systems. Further the AADL is a tool neutral language and so facilitates exchange of architecture descriptions between different tools.

Architecture Driven Modeling: We need an environment where (i) Architectures are comprehended and (ii) Architecture transformations (from functional architecture to embedded system architecture) can be achieved. Such an environment should allow for linking and management of the relationships between architectures and functional models that are developed in state of the art modeling tools such as Simulink. Such an environment should also provide support for migration of the functional models to embedded system models.

In this paper we use the commercial tool IME (Integrated Modeling Environment) to evaluate and demonstrate the approach.

Architecture Driven Embedded Software Creation: We need technologies that will synthesize the embedded software corresponding to the functional models, and the distributed real-time software components.

For generation of the embedded software corresponding to the functional models, many of the native modeling tools themselves provide this capability. In this paper we use Simulink as the modeling tool.

For generating the real time executive for distributed processor system, in this paper, we use Ocarina to evaluate and demonstrate the approach. Ocarina toolsuite provides support for code generation from AADL models for the real time executables, with appropriate integration of the software auto-generated from functional models.

These technologies – AADL, IME and Ocarina facilitate the designer to go from high-level modeling down to code seamlessly, and are described in detail in the following sections.

3.1. AADL

The Architecture Analysis and Design Language (AADL) was adopted as an SAE (Society of Automotive Engineers) standard in the year 2004. The AADL is a tool neutral language that can be used to describe the run-time architecture of the embedded system. Model-based analysis for schedulability, safety, security, etc. can be performed using these descriptions [4]. It naturally provides the ability to describe the architecture of both hardware and software components and data along with their variations with regards to implementations. AADL components are described as component types and component implementations. Component type defines the interface of the component. Component implementation inherits the properties of the component type and describes the sub-components and connections. There can be different component implementations for a single component type definition leading to variant implementations for the same component. In our work the following subset of AADL components are used.

System – These components can be used to describe the system architecture. They can be used to represent any software or hardware component or a combination of both.

Process - Process components are an abstraction of software responsible for scheduling and for executing threads. They execute in their own memory in a processor.

Thread – Thread components are an abstraction of software responsible for scheduling and executing sub-programs. Thread execution periods and execution times can be set as properties.

Sub-programs – Subprograms represent elementary pieces of code that process inputs to produce outputs. In the AADL only the interfaces are described. The implementations must be provided by the host language. In our work the functional models developed in Simulink provide the implementation.

Processors – These components are abstractions of hardware and software that schedule and execute processes. Each processor will have its own clock which is the base time for all the components running on the processor.

Buses – Bus components are used to exchange data between hardware components.

A complete description of the AADL is provided here [5].

The functional architecture can be initially described using the AADL System components. Ports and connections define the component interfaces and communication. Once functional models are associated with the System components the architecture can be migrated to the behavioral modeling domain. The System components can later be translated to embedded components consisting of Processor, Process, Thread and Subprogram components.

3.2. Integrated Modeling Environment (IME)

IME is a model management and architecture creation and analysis tool [6]. As described in Section 2 the system architecture drives the creation of functional models and generating distributed software. A visualization environment must be provided for the system architecture. Such an environment should be able to exchange architecture descriptions with other tools. Component behavior models developed over time are archived in a model repository for reuse. In order to find the consistent models their architectural information must be extracted and stored. Intelligent queries to search for certified models coupled with consistency checks facilitate selection of models. Once the selections are complete the architecture needs to be migrated to the behavioral modeling domain. As discussed in section 2.2 the functional models must be integrated seamlessly with the hardware architecture descriptions. During this step the environment must facilitate engineers to define the bindings between functional and embedded components. The bindings define the allocation of functional components to the AADL Processes which must be bound to AADL Processors. The AADL Processor components represent the different ECUs. Once the bindings are defined the embedded system architecture must be generated. From this architecture AADL descriptions need to be exported. IME is a tool that can support all the activities described above. The resulting AADL models can then be used in code generation.

3.3. Ocarina

Ocarina [7] is a toolsuite developed by the AADL group at Telecom ParisTech. It aims at providing AADL model manipulation, syntactic/semantic analysis, model analysis capabilities (using external tools like Cheddar [8]) or embedded (e.g. generation of Petri Net models, analysis of models using the REAL constraint language). Besides, Ocarina proposes code generation from AADL models to either C or Ada using the PolyORB-HI family of AADL runtimes. Code for the runtime, and the code generated follow carefully restrictions for the High-Integrity domains as mandated by the space, avionics or automotive domains. Targeted RTOS range from bare boards Ada runtime, real-time executive like RTEMS or RT-Linux, domain-specific OS like POK for avionics systems [9] or native platforms for rapid prototyping. Ocarina supports both AADLv1.0 and AADLv2.0. Contrary to many MDE tools, Ocarina relies on its own internal meta-model engine, closer to a compiler AST. This allows for a wide range of internal optimizations, allowing to process large models quickly.

Ocarina is available under the GPL license and runs on most operating systems (Windows, Mac OS X, Linux). Releases, examples of application-level models, case studies as well as documentation are available on our AADL portal [10]. Ocarina has been successfully tested in the European project IST-ASSERT [11], led by the European Space Agency, and the French R&D project Flex-eWare [12], led by Thales.

One notable feature of Ocarina is the ability to include seamlessly any functional notations as implementation of blocks such as subprograms or threads. Instead of writing C code, the designer may insert a SCADE or Simulink functional blocks. Then, the Ocarina code generator will generate all the required glue code to integrate C code generated from Simulink Real-Time workshop or SCADE's kcg C code into AADL runtime code. This suppresses a tedious and error prone integration work, and allows the user to focus only on the behavior of its system instead of bothering with low-level implementation details. This feature allows for a natural bridge between a high level architecture based tool like IME and simulation on real hardware, in a distributed setup, using Ocarina generated code. We illustrate this in the next section.

4. CASE STUDY

In this section we describe the application of the architecture driven approach to the Electronic Throttle Control (ETC) system. ETC replaces a mechanical system consisting of a linkage between the accelerator pedal to the throttle plate. In the mechanical system the vehicle operator directly regulates the engine airflow by adjusting the position of the throttle plate via the accelerator pedal. At idle speed conditions, the airflow bypasses the throttle plate and is regulated with an Idle Air Control (IAC) valve. In the ETC system the throttle plate is actuated electronically. The desired throttle plate position (setpoint) is determined based on the pedal position as well as other inputs and operating conditions. A primary benefit of ETC is it enables system designers to incorporate throttle control into other vehicle functions, such as cruise control and vehicle stability control. ETC is considered a safety critical system. As a result a considerable portion of ETC functionality is in place for redundancy and safety monitoring. Figure 4 shows the functional architecture of one ETC system as visualized within IME. The main components in the architecture are the core controller, actuator, sensors and the plant. The core controller consists of three important functions – safety monitor, manager and servo control.

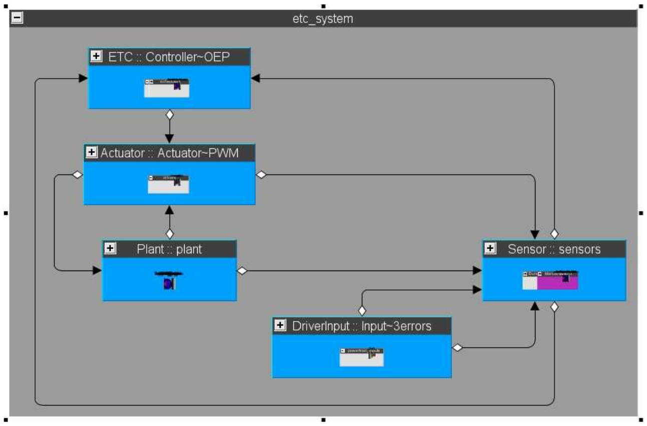


Figure 4: ETC Closed Loop System

In this case study we will consider three deployment scenarios for the embedded software.

Scenario1 – Single ECU: We consider the case in which a supplier is assigned the responsibility of developing the actuator system. The supplier develops a first version of the actuator based on the specifications from the OEM. The product development cycle requires virtual integration of the entire system much in advance of the availability of actual physical hardware. Therefore the supplier develops the actuator driver models and the plant models and delivers the closed loop actuator system model to the OEM. The OEM integrates the actuator models into the bigger ETC system models. The next step for the OEM is to deploy all the controller components on a single target ECU in order to eventually generate the control software.

Scenario2 – Single ECU + Smart Actuator: As an alternative to the scenario 1 above, the supplier offers a technologically advanced actuator in which the driver software is tightly integrated with the physical actuator. In turn, the OEM can offer superior functionality on some of its product lines, without disturbing the basic hardware architecture. However, in this case, the actuator driver software executes on a dedicated ECU. When the supplier delivers the actuator driver and plant models to the OEM for integration into the ETC system model, the OEM should ensure that code generation does not include the drivers with the main ECU.

Scenario3 – Two ECUs + Smart Actuator: The OEM wants to reuse the same physical system and controls architecture for an advanced defense application, where safety criticality and redundancy are highly prioritized. In order to accommodate this modification the OEM wants to deploy the safety monitor component on a separate target ECU; the core functionality runs on the main ECU; the drivers run on the processor with the actuator.

In all the above scenarios the functional architecture remains the same but the deployment architecture is different. We now discuss the workflow and tool chain to support the deployment in the above scenarios.

Figure 5 depicts the steps involved in the generation of functional models from the system architecture. The system architecture can be developed using AADL authoring tools such as OSATE (Open Source AADL Tool Environment) [5] and imported into IME. The model repository can be mined and consistent functional models can be selected and associated with the system architecture. The functional Simulink models of the system can then be generated for further analysis.

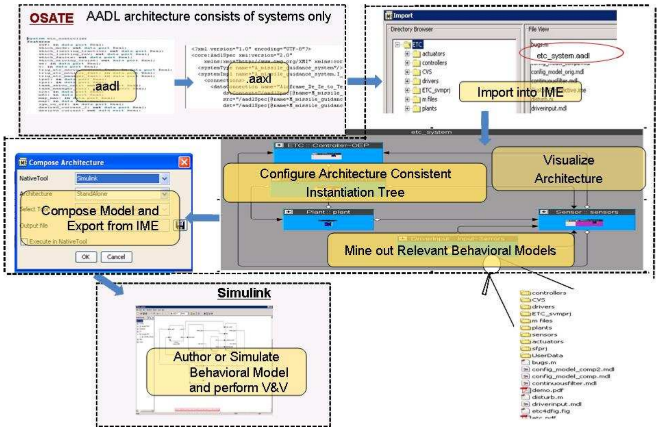
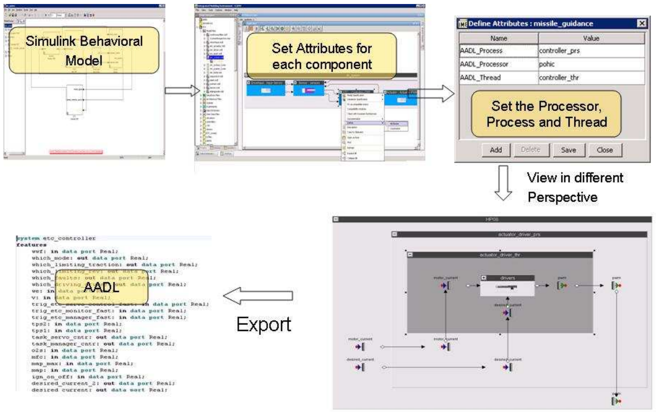


Figure 5: Generation of functional models consistent with system architecture.

The next step is to deploy the functional models on the target. Since the functional architecture is different from the target architecture and also the target architecture is different in the three scenarios considerable re-architecting of the functional architecture must be done. The workflow for allocating the functional software components to the target is shown in figure 6.


Simulink Behavioral Model
Set Attributes for each component
Define Attributes : missile_guidance
Set the Processor, Process and Thread
Add
Delete
Save
Close
View in different Perspective
AADL
Export
HPOS
actuator_driver_prs
actuator_driver_thr
drivers
motor_current
pwm
desired_current

Name	Value
AADL_Process	controller_prs
AADL_Processor	pohic
AADL_Thread	controller_thr

```
system etc_controller
features
    wwf: in data port Real;
    which_mode: out data port Real;
    which_limiting_traction: out data port Real;
    which_limiting_rev: out data port Real;
    which_faults: out data port Real;
    which_driving_[illegible]: out data port Real;
    we: in data port [illegible]
    v: in data port Real;
    trig_etc_servo_control_fast: in data port Real;
    trig_etc_monitor_fast: in data port Real;
    trig_etc_manager_fast: in data port Real;
    tps2: in data port Real;
    tps1: in data port Real;
    task_servo_cntr: out data port Real;
    task_manager_cntr: out data port Real;
    o2s: in data port Real;
    mfc: in data port Real;
    map_max: in data port Real;
    map: in data port Real;
    ign_on_off: in data port Real;
    desired_current_2: out data port Real;
    desired current: out data sort Real;
```

Figure 6: Generation of embedded system architectures from functional architectures

The functional system architecture can be annotated with the binding information. The binding information is different in the three deployment scenarios. The binding information must include the processor, process and thread in which each component must execute. Then the embedded architectures are generated from the functional architecture. Figures 7, 8 and 9 illustrate the embedded architectures for scenarios 1, 2 and 3 respectively. Figure 9 shows the detailed hierarchy of the embedded system. The embedded system consists of two ECUs – ECU1 and ECU2. A single process executes in ECU1 which manages a single thread. This thread executes the monitor component. The connections to the plant are not shown in the figure. In all the three target architectures the communication between the software components is maintained as specified in the functional architecture.

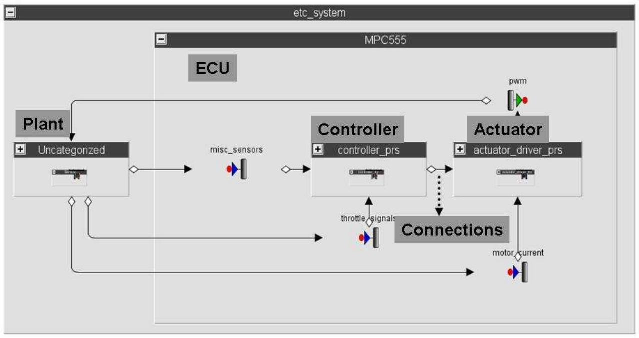


Figure 7: Illustrates scenario 1 - Controller and actuator drivers are executed in a single ECU

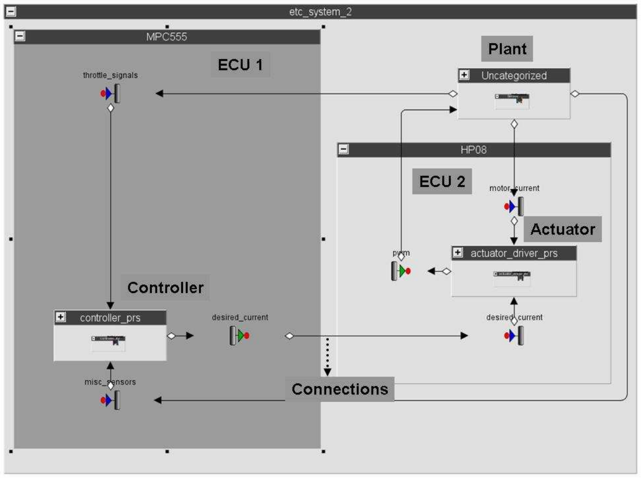


Figure 8: Illustrates scenario 2 – Actuator driver executes on a dedicated ECU

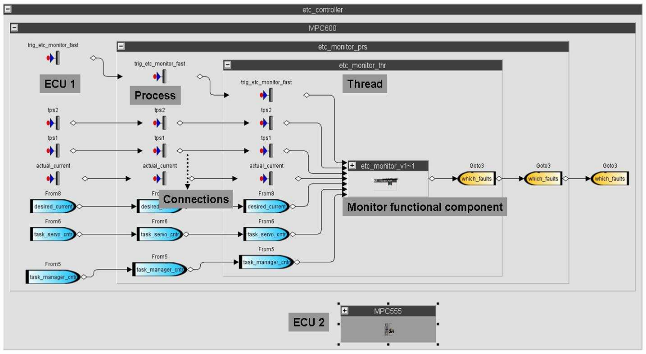


Figure 9: Illustrates scenario 3 – Monitor component executes in a dedicated ECU. Other controller components execute in a second ECU.

The next step is the generation of embedded software. In scenarios 1 and 2 the controller components were deployed on a single ECU. So the re-architected embedded architectures can be migrated back to the Simulink domain which supports code generation for a single ECU. In scenario 3 the controller components are distributed among two ECUs. In order to generate distributed software using a single model the architecture is exported as an AADL model. The Ocarina toolsuite can parse the AADL model and extract the execution characteristics of the embedded system. Executables can be generated for each process. Within each process the required threads are created which in turn call the subprograms that represent the interfaces of the functional components. Simultaneously code can be generated from the functional models for each component. The functional components are called from within the subprograms. The execution architecture and the communication among the embedded and functional components are shown in figure 10. Figure 11 illustrates the structure of the monitor subprogram and the call to the monitor functional component. The integrated software can be executed on a real-time system.

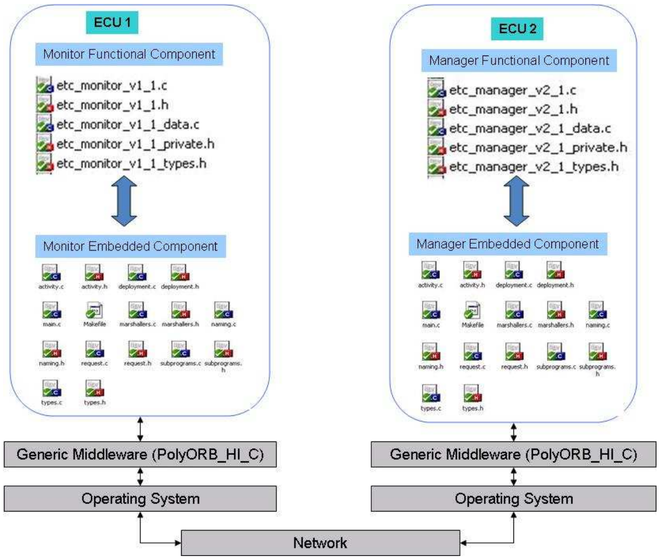
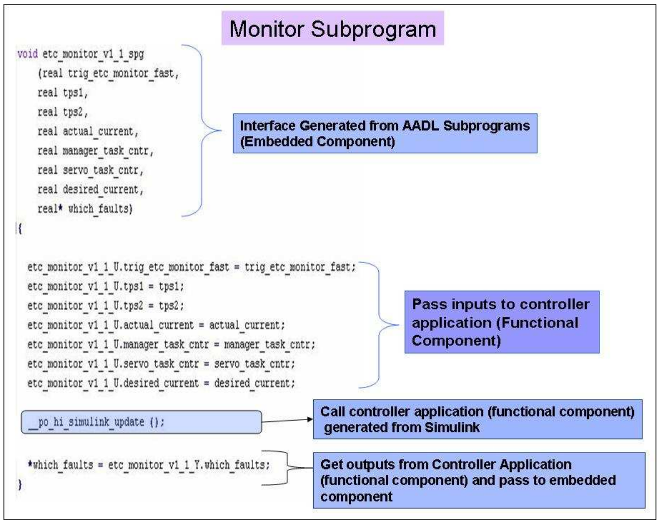


Figure 10: Communication between the monitor and manager components executing in separate ECUs.


Monitor Subprogram
Interface Generated from AADL Subprograms (Embedded Component)
Pass inputs to controller application (Functional Component)
Call controller application (functional component) generated from Simulink
Get outputs from Controller Application (functional component) and pass to embedded component

```
void etc_monitor_v1_1_spg
    (real trig_etc_monitor_fast,
    real tps1,
    real tps2,
    real actual_current,
    real manager_task_cntr,
    real servo_task_cntr,
    real desired_current,
    real* which_faults)
{

  etc_monitor_v1_1_U.trig_etc_monitor_fast = trig_etc_monitor_fast;
  etc_monitor_v1_1_U.tps1 = tps1;
  etc_monitor_v1_1_U.tps2 = tps2;
  etc_monitor_v1_1_U.actual_current = actual_current;
  etc_monitor_v1_1_U.manager_task_cntr = manager_task_cntr;
  etc_monitor_v1_1_U.servo_task_cntr = servo_task_cntr;
  etc_monitor_v1_1_U.desired_current = desired_current;

  __po_hi_simulink_update ();

  *which_faults = etc_monitor_v1_1_Y.which_faults;
}
```

Figure 11: Structure of a subprogram

5. CONCLUSIONS

In this work we discussed the approaches and specific technologies that enable the generation of distributed embedded software from functional models. An architecture driven approach facilitates the reconciliation of the functional and embedded architectures and their integration. The specific tools and technologies also support modular development and reuse of software components. These improve the efficiency of the engineering activities. Further studies needs to be performed with regards to deploying the embedded software on the target hardware and real-time testing of the system.

REFERENCES

[1] S. Gopalswamy, et.al., “Practical Considerations for the Implementation of Model Based Control System Development Processes”, Proceedings of the Conference on Control Applications, 2004.

[2] Simulink is a registered trademark of The Mathworks, www.mathworks.com.

[3] SAE. Architecture Analysis & Design Language v2.0 (AS5506), September 2008.

[4] B. Lewis, P. Feiler, Multi-Dimensional Model Based Engineering for Performance Critical Computer Systems using the AADL, *ERTS 2006*, Toulouse, France.

[5] The SAE AADL, www.aadl.info.

[6] IME, http://www.emmeskay.com/tools/ime.

[7] "OCARINA: An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications" Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Reliable Software Technologies'09 - Ada Europe. Brest, France. June 2009 pp. 237-250

[8] Cheddar, http://beru.univ-brest.fr/~singhoff/cheddar

[9] J. Delange, L. Pautet and F. Kordon. Code Generation Strategies for Partitioned Systems. In 29th IEEE Real-Time Systems Symposium (RTSS'08) Work In Progress, IEEE Computer Society, December 2008.

[10] http://aadl.telecom-paristech.fr .

[11] http://www.assert-project.net

[12] http://www.flex-eware.org