

SAFE AUTOMOBILE CONTROLS

Subaru uses SCADE software to develop safe and reliable electronically controlled circuits and systems for hybrid-electric vehicles.

By Masaru Kurihara, Deputy General Manager, Electronics Engineering Department, Fuji Heavy Industries Ltd., Tokyo, Japan



Market pressures to increase fuel economy, maintain safety and provide entertainment are forcing the auto industry to develop automobiles that are increasingly complex and adaptable. Vehicles are now largely computerized, and the electronic control unit (ECU) that manages the systems in each model is governed by complex software. New hybrid-engine vehicle (HEV) technologies rely on extensive circuitry and software. In an HEV, a central computer manages both a traditional combustion engine system and an electric motor via ECU.

Because of the need to continually juggle costs and design requirements, the automotive industry employs AUTOSAR (a development partnership of electronics, semiconductor and software organizations that provides standards to manage growing electronics complexity in this industry) standards and a methodology called model-based development or design (MBD). MBD requires design engineers to use a common design environment that supports model integration and virtual real-time testing of the entire system.

Subaru®, the automotive brand of Fuji Heavy Industries (FHI) Ltd. — a comprehensive, multifaceted transport equipment manufacturer — recently started its own HEV and electric

Vehicles are now increasingly complex, adaptable and largely computerized.

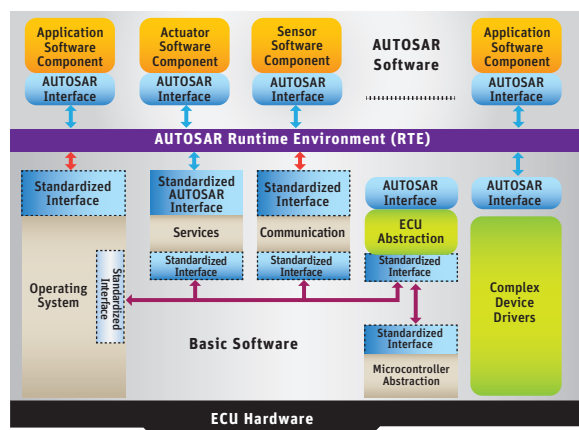
vehicle (EV) programs and adopted the MBD approach for all upcoming development projects, like the Subaru XV. In the company's search for a development environment tool that meets MBD requirements, Subaru engineers evaluated the SCADE software modeling tool from Esterel Technologies, a wholly owned ANSYS company. SCADE provides an intuitive graphical interface so system and software engineers can easily integrate and verify their models. C source code is generated automatically from the models produced in SCADE. This minimizes the chance of programmer error and automatically incorporates SCADE's strict standards and safety requirements.

To continually juggle costs and design requirements, the automotive industry employs AUTOSAR standards and model-based design. ▶

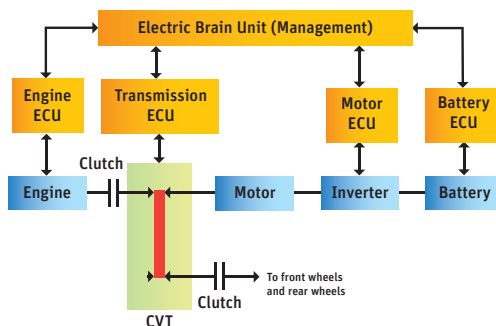
REORGANIZATION OF SOFTWARE ARCHITECTURE

Since HEV development was quite new to Subaru, engineers had the opportunity to create the software architecture from the ground up. Applying AUTOSAR standards to the new ECU reduced basic software development cost and time. However, an even more challenging and important goal was to ensure that the software components could be reused for future projects. All requirements for the ECU are marked as either application software or basic software to configure three layers of software, as per AUTOSAR architecture. Services, drivers and the operating system (OS) are components of basic software; they are created by hand-coding or are available as commercial off-the-shelf (COTS) products. Subaru focuses on pure application design according to the requirements, and this facilitates the deployment of SCADE since the software's models are independent from the OS and hardware-dependent implementations.

When SCADE is deployed, interface nodes for the middleware layer are created automatically by Subaru's own Java® utilities based on Eclipse APIs that analyze the definition files written for basic software. Data types and data structures are extracted from the definition files and defined as SCADE types in the SCADE project. Bridging the gap between application and services with a middleware model layer minimizes human error. When the interface between the middleware layer and the basic software layer is changed, the definition of SCADE types is automatically updated in a consistent way. SCADE's semantic checker continually verifies the SCADE type definition with the models to eradicate modeling error.



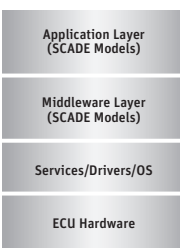
▲ Software architecture in EBU control software is layered to comply with the AUTOSAR standard.



▲ SCADE is deployed for the hybrid vehicle management system named electric brain unit (EBU) for a production vehicle released to the market in 2013. In this schematic, the electric brain unit manages several ECUs.

SOFTWARE DESIGN PROCESS WITH SCADE

Once the SCADE types are defined, Subaru can execute a set of small cycles iteratively until a detailed design and verification process is completed. The software development process with SCADE at Subaru starts with development of test scenarios based on software requirements and conversion of many pieces of functional models (Simulink®) designed by system engineers into SCADE models via a Simulink gateway. Once the models are converted successfully, they are integrated into a safe SCADE architecture model. To maintain consistency of the conversion from Simulink into SCADE, engineers feed the same test scenarios developed for functional model design simulation into SCADE for unit testing. When the detailed design is completed with unit tests, C code is generated from the integrated model to pass to the EBU test.



SAFE ARCHITECTURE DESIGN WITH SCADE

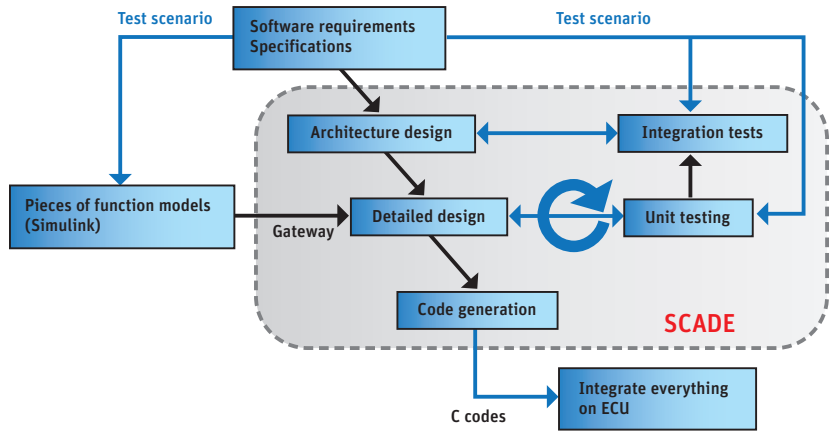
Due to the modularity of each SCADE node, functional part models are verified by unit testing. The most critical issue for safe software architecture is to ensure that all data are safe when the application operates under multi-task execution. Introducing a safe partitioning architecture from a COTS OS into the automotive software is not easy because of the trade-off between cost and functionality;

multi-task execution is mandatory for optimized execution. To satisfy the trade-off issue, a safe SCADE architecture model is designed.

A root node is commonly deployed to a safe architecture independent from projects or type of applications. It consists of a safe state machine containing a decision tree. The state machine has only two states: init and run. The init state is active only at initialization, while the run state is active for the rest of the cycle. The run state contains a decision tree with three selections that can be executed, depending on the value of a variable thread coming from the basic software. The difference in the values comes from the timing of execution configured in OS. For example, each subnode can be executed each 0.1 ms, 1 ms or 10 ms, as shown in the figure. However, subnodes communicate with each other via an interface and, therefore, need protection from interruption. A faster task could be interrupted during execution of a slower task. What if the slower task uses the data intended for a faster one? During the execution of the slower task, the output data from the faster one is updated by the interruption, and this may overwrite the values being used for the current execution. These types of data errors can produce random results in the model. For slower tasks to execute safely, the inputs should be stored explicitly before being used in the calculation.

COMBINING APPLICATION MODELS WITH A SAFE SCADE ARCHITECTURE MODEL

Once the root node describes concrete architecture, each subnode can be designed using a modular approach. Functional models for EBU applications are designed by a system engineering team with Simulink. These Simulink models are also imported into SCADE using the Simulink gateway. Before being imported, they are verified in the Simulink environment based on the software requirements. The test scenarios are described in Excel® sheets and converted into *.in format for the SCADE simulator. When both simulation results are identical, it means that the simulation is correct. Once all pieces of the functional Simulink models are converted into SCADE correctly, they are integrated into the safe SCADE architecture node.



▲ HEV software development process with SCADE



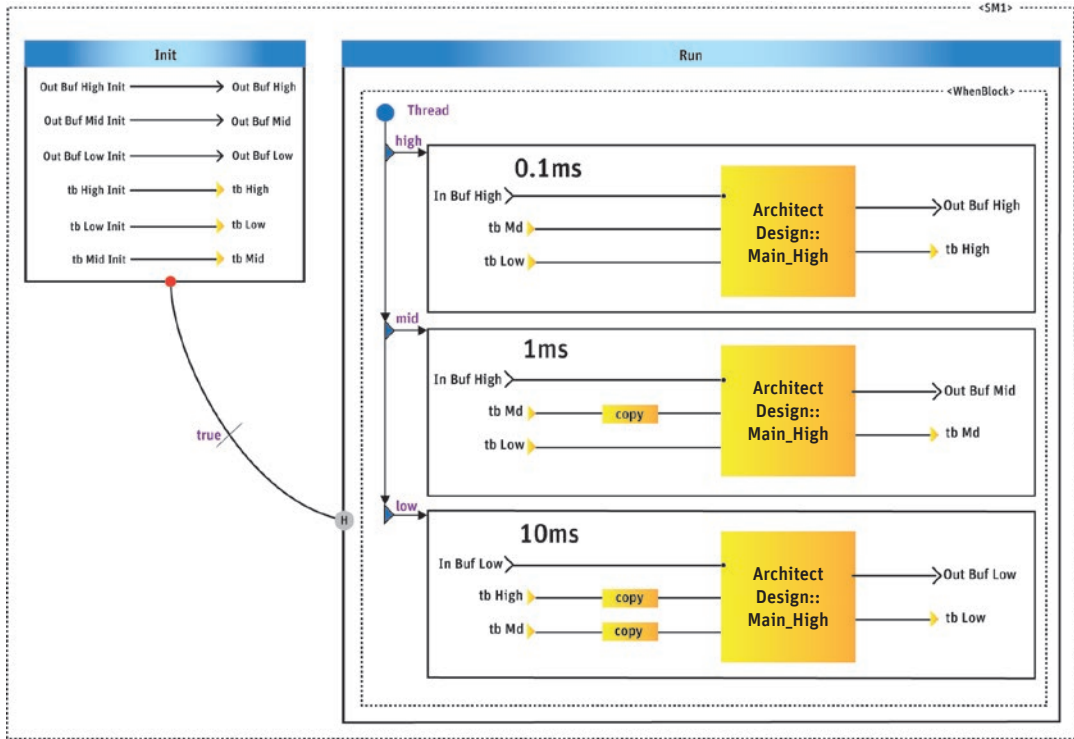
COURTESY SUBARU.

REFINEMENT OF SCADE MODELS

To reduce verification time, the SCADE Suite Design Verifier is used to check if the SCADE model prior to refinement is identical to the one after refinement. In general, formal verification techniques are used to test properties like safety, but verification algorithms may face numerical difficulties if the nodes contain more arithmetic calculations than decision diagrams and state machines. Subaru engineers refine the SCADE models daily, and Design Verifier helps to validate the models.

SAFE AND RELIABLE AUTOMATIC CODE GENERATION FOR INTEGRATION

The final stage of the SCADE process is to generate C source codes from the verified SCADE models using the IEC 61508 certified SCADE Suite KCG code generator. The generated code usually meets safety objectives. Because the KCG tool has been qualified and certified for this purpose, Subaru verifies that the safe properties obtained from the safe SCADE architecture model are retained in the generated codes. Two SCADE Suite KCG features are essential to ensure that safe properties



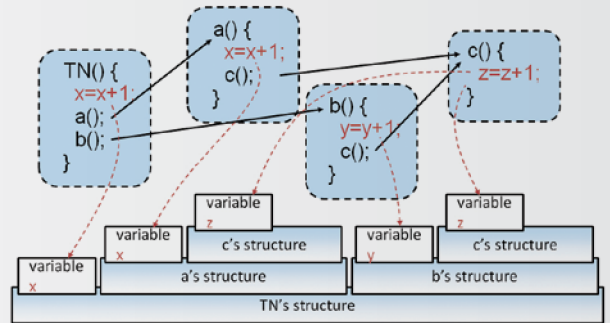
▲ A combination of safe state machine and decision diagram is the basis of the safe SCADE architecture model.

Comparing the Architecture Model and Generated Codes

SCADE assures data access with a unique path when a node contains variables that memorize the value. This is not applied to a “function” in SCADE because the function does not contain any variables that memorize the values. If node TN() calls node a() and node b(), and a() and b() commonly call node c(), and all of them contain variables that memorize values, the data are structured by the node in the generated codes. Although node c() is commonly called from two nodes a() and b(), its corresponding data are structured separately. Because KCG guarantees it, another data structure is created safely when the root node TN() is called by interruption while it is being executed.

The generated code is also correct thanks to SCADE Suite’s KCG IEC 61508 certified code generator. C code is a part of the generated codes from a safe SCADE Suite architecture model using KCG. To compare the safe SCADE architecture model with the codes, all copies of inputs are coded by the `kcg_copy` macro function using the default `memcpy` function. KCG makes it possible to replace the macro with a user-defined macro after


code generation without any impact on the other codes. Subaru replaced the `kcg_copy` macro with the one inhibiting interruption during the copy.



▲ Safe access to variables that memorize the values exists as a unique path.

are met in the codes: The variables are protected from overwriting when interruption occurs, and input data is appropriately copied before being used as described in the safe SCADE architecture model.

DEVELOPING CONTROL SOFTWARE FOR HEVS USING SCADE

Using SCADE software, Subaru was able to describe consistent readable models ranging from safe architecture design to detailed designs. Thanks to SCADE Suite's KCG IEC 61508 certified code generator, the verification time at code level was significantly reduced, as most of the verification was completed upfront at the SCADE model level. A small group of Subaru engineers completed a large and very complex application while significantly reducing software development and testing time. Subaru engineers continue to use SCADE Suite as an important part of their HEV development process. 

Subaru engineers completed a large and very complex application while significantly reducing software development and testing time.



COURTESY SUBARU.

```

if (Ctxt_App_Main.init) {
    Ctxt_App_Main.init = kcg_false;
    SM1_state_act = SSM_st_Init_SM1;
}
else {
    SM1_state_act = Ctxt_App_Main.SM1_state_nxt;
}
switch (SM1_state_act) {
    case SSM_st_Run_SM1 :
        Ctxt_App_Main.SM1_state_nxt = SSM_st_Run_SM1;
        switch (thread) {
            case low :
                copy_tbMidType(&Ctxt_App_Main.tbMid, &_L8_SM1_Run_WhenBlock_low);
                copy_tbHighType(&Ctxt_App_Main.tbHigh, &_L7_SM1_Run_WhenBlock_low);
                Main_Low(&InBufLow, &_L7_SM1_Run_WhenBlock_low, &_L8_SM1_Run_WhenBlock_low, &Ctxt_App_Main._2_Context_2);
                kcg_copy_tbLowType(&Ctxt_App_Main.tbLow, &Ctxt_App_Main._2_Context_2.tbLow);
                kcg_copy_
                OutBufLowType(&OutBufLow, &Ctxt_App_Main._2_Context_2.OutBufLow);
                break;
            case mid :
                copy_tbHighType(&Ctxt_App_Main.tbHigh, &_L8_SM1_Run_WhenBlock_mid);
                Main_Mid(&InBufMid, &_L8_SM1_Run_WhenBlock_mid, &Ctxt_App_Main.tbLow, &Ctxt_App_Main._1_Context_2);
                kcg_copy_tbMidType(&Ctxt_App_Main.tbMid, &Ctxt_App_Main._1_Context_2.tbMid);
                kcg_copy_
                OutBufMidType(&OutBufMid, &Ctxt_App_Main._1_Context_2.OutBufMid);
                break;
            case high :
                Main_High(&InBufHigh, &Ctxt_App_Main.tbMid, &Ctxt_App_Main.tbLow, &Ctxt_App_Main.Context_2);
                kcg_copy_tbHighType(&Ctxt_App_Main.tbHigh, &Ctxt_App_Main.Context_2.tbHigh);
                kcg_copy_OutBufHighType(&OutBufHigh, &Ctxt_App_Main.Context_2.OutBufHigh);
                break;
        }
    case SSM_st_Init_SM1 :
        Ctxt_App_Main.SM1_state_nxt = SSM_st_Run_SM1;
        kcg_copy_tbLowType(&Ctxt_App_Main.tbLow, (tbLowType *) &tbLowInit);
        kcg_copy_tbMidType(&Ctxt_App_Main.tbMid, (tbMidType *) &tbMidInit);
        kcg_copy_tbHighType(&Ctxt_App_Main.tbHigh, (tbHighType *) &tbHighInit);
        kcg_copy_OutBufLowType(&OutBufLow, (OutBufLowType *) &OutBufLowInit);
        kcg_copy_OutBufMidType(&OutBufMid, (OutBufMidType *) &OutBufMidInit);
        kcg_copy_OutBufHighType(&OutBufHigh, (OutBufHighType *) &OutBufHighInit);
        break;
}

```