An Embedded Control Software Development Environment with Data Consistency Verification for Preemptive Multi-Task Systems

Taira Ito, Masayoshi Tamura, Myungryun Yoo and Takanori Yokoyama

Abstract—The paper presents an embedded control software development environment that provides a tool to verify the data consistency of embedded control software designed with Simulink models and UML models. A controller model is built with MATLAB/Simulink in the control logic design phase. Then a software model that correctly executes the control logic in the actual computing environment is built in the software design phase. It must be verified during software design that the data consistency of the software is preserved in the preemptive multi-task environment because the simulation of Simulink models is performed in an ideal environment in which "zero-time execution" is assumed. We present a method to verify the data consistency with SPIN model checker. We also present a tool that automatically generates Promela code for data consistency verification. We have applied the tool to a number of software models transformed from Simulink models and have confirmed its usefulness for embedded control software design.

Keywords—embedded control software, real-time systems, model-based design, verification, model checking

I. Introduction

Model-based design has become popular in embedded control software design, especially in the automotive control domain. A CAD/CAE tool such as MATLAB/Simulink[1] is used to design control logic. A controller model is designed with block diagrams and verified by simulation, and source code can be generated from the controller model by a code generator such as Embedded Coder[1]. However, such CAD/CAE tools are not sufficient for software design. Sangiovanni-Vincentelli and Di Natale pointed out the shortcomings of the tools: lack of separation between the functional and architecture model, lack of support for defining the task and resource model, lack of modeling for analysis and backannotation of scheduling-related delays and lack of sufficient semantics preservation[2]. The CAD/CAE tools should be used for just control logic design, not for software design.

Software modeling languages such as UML should be used for software design. UML provides a number of kinds of diagrams, which are useful for not only functional design but also nonfunctional design.

An embedded control system is usually designed as a preemptive multi-task system. Tasks are allocated to CPU cores when a multi-core processor is used. We have to design the software to meet timing constraints and to correctly execute control logic in the preemptive multi-task environment. We design the task structure, task priorities, inter-task communication, inter-task synchronization and mutual exclusion to preserve data consistency. So the efficient verification of data consistency is required.

Verification methods and tools utilizing model checkers such as SPIN[3] have been presented for UML models[4][5][6] and Simulink models[7][8], most of which are based on state transitions. Some researches deal with multi-task environments on real-time operating systems[9]. However, no tools are presented for the data consistency verification of the control software that executes the control logic in a preemptive multi-task environment.

The goal of the research is to present an embedded control software development environment that supports data consistency verification. We present a tool to generate Promela code from UML models that describe the structure and the behavior of the control software executed in a preemptive multi-task environment. We can verify the data consistency of the control software by running the Promela code through SPIN.

The rest of the paper is organized as follows. Section II describes the control software development process with Simulink models and UML models. Section III describes a data consistency verification method and a tool that generates Promela code for data consistency verification. Finally, Section IV concludes the paper.

II. Control Software Development Process

A. Software Development Flow

Fig. 1 shows the embedded control software development flow, which consists of the control logic design phase, the software design phase and the programming phase[10].

In the control logic design phase, we build a Simulink model that represents a control system. A Simulink model of a control system usually consists of a plant model and a controller model. The controller model represents control logic.
Fig. 2 shows an example Simulink model, which is a part called Throttle Controller of an automotive control system. The model consists of three input blocks for Engine Revolution, Engine Status and Accelerator Opening, two subsystem blocks for Torque Calculation and Throttle Opening Calculation, and an output block for Throttle Opening. The details of the calculation of Torque and Throttle Opening are described in the lower layer models of the subsystem blocks. The calculations are periodically executed in the control period.

In the software design phase, we build a software model in UML to implement the controller model. Software design consists of functional design and nonfunctional design. We transform a Simulink model into a functional model represented in UML in functional design. Then we build an implementation model taking account of nonfunctional properties in nonfunctional design.

Finally, C or C++ source programs are generated from the implementation model in the programming phase.

B. Functional Design

The transformation from a Simulink model into a functional model represented in UML is automatically performed by a model transformation tool[10], the transformation rules of which are based on the design method of the time-triggered object-oriented software[11][12]. The tool generates class diagrams, object diagrams and sequence diagrams. A control system consists of controller objects that represent subsystems of the control system and value objects that represent important data, which represent reasonable physical quantities such as input values, output values, observed values, estimated values and desired values.

Fig. 3 shows the class diagram of the software corresponding to the Simulink model shown by Fig. 2. The class Controller is a class for controller objects and the class ValueObject is a class for value objects. The class ValueObject has the method update that calculates value and the method get to read its value. The method update of ValueObject gets the values of other ValueObjects by calling their methods get and calculates its own value and stores the calculated value in its attribute. An object of Controller consists of a number of objects of ValueObjects and its method exec calls their methods update. The method exec is periodically executed by a task in the control period.

In Fig. 3, there is a subclass of Controller called ThrottleController, which corresponds to the Simulink model shown by Fig. 2. There are also subclasses of ValueObject called EngineRevolution, EngineStatus, AcceleratorOpening, Torque and ThrottleOpening, which correspond to the subsystem blocks of the Simulink model.

We also represent tasks and CPU as objects. In Fig. 3, the class Task represents tasks and the class Processor represents CPU that executes tasks. The class Task has an attribute that represents the priority of the task.

Fig. 4 shows the object diagram of the function model corresponding to the Simulink model shown by Fig. 2. The
C. Nonfunctional Design

We design the task structure, task allocation to CPUs and task priorities to meet timing constraints in nonfunctional design. We also verify that the data consistency is preserved in the preemptive multi-task environment because the functional model is transformed from the Simulink model, the simulation of which is performed in an ideal environment in which "zero-time execution" is assumed.

The data consistency depends on the task structure, task allocation and task priorities. For example, when the methods update of all value objects of Fig. 4 are executed by one task, the data consistency is preserved. However, if we implement the functional model with multiple tasks, data consistency may be violated.

Fig. 6 (a) shows an example of preemptive execution of the tasks on a single processor. Here, we assume update of Torque and update of ThrottleOpening are executed by TaskA, update of EngineStatus is executed by TaskC, and the priority of TaskC is higher than the priority of TaskA. TaskA(n) (the nth job of TaskA) is preempted by TaskC(m+1) (the (m+1)th job of TaskC). TaskA(n) executes update of Torque before the preemption and executes update of ThrottleOpening after the preemption. The calculation of Torque uses the value of EngineStatus calculated by TaskC(m), but the calculation of ThrottleOpening uses the value of EngineStatus calculated by TaskC(m+1). So the data consistency is violated in this case.

Fig. 6 (b) shows an example of parallel execution of the tasks on a multicore processor. TaskA and TaskC are executed in parallel. TaskA(n) executes update of Torque before the update of EngineStatus executed by TaskC(m+1) and executes update of ThrottleOpening after that. So the data consistency is violated as similar to the case (a).

Fig. 4 shows an example object diagram of an implementation model. ThrottleController is executed by TaskA with priority 10, which is allocated to CPU1. The methods update of EngineRevolution, EngineStatus and AcceleratorOpening are called by EngineRevolutionController executed by TaskB with priority 10, EngineStatusController executed by TaskC with priority 20 and AcceleratorOpeningController executed by TaskD with priority 30 each. TaskB, TaskC and TaskD are allocated to CPU2. Fig. 5 and Fig. 7 show the sequence diagrams of the implementation model.

After the task design, we verify the data consistency of the implementation model using the verification method described in Section III. If the data consistency of the verified model is violated, we modify the model to preserve the data consistency manually or using a model weaver[13].

Fig. 8 shows the object diagram of a modified implementation model with a buffering mechanism, which is one of wait-free inter-task communication mechanisms. Fig. 9 shows the sequence diagram of Task2 of the modified model. The sequence diagrams of another tasks are the same as Fig. 7. The data consistency is preserved in the modified model.

### Data Consistency Verification

#### A. Promela Code for Verification

Data consistency verification is performed with the model checker SPIN. Promela is a verification modeling language used in SPIN. To verify the data consistency of value objects, we check the number of updating of the value used to calculate the values of related value objects. For example, the number of updating of EngineStatus used to calculate Torque is m and the number of updating of EngineStatus used to calculate ThrottleOpening is m+1 in Fig. 6. The data consistency is violated in this case because the former number of updating is different from the latter number of updating.

We use two kinds of Promela code: one for random simulation, the other for correctness verification with LTL (Linear Temporal Logic) formula.
Fig. 10 shows the Promela code for random simulation of the implementation model represented by Fig. 3, Fig. 4, Fig. 5 and Fig. 7. Some parts of the code is omitted for simplicity.

A task is represented by a process with an infinite loop. TaskA and TaskC is represented as processes (TaskB and TaskD are omitted, but similar to TaskC). Priority-based scheduling is simulated with provided clauses. A method of an object is represented as an inline function. For example, update of Torque is represented as update__Torque() and exec of ThrottleController is represented as ThrottleController().

The number of updating of an object is stored in a variable. For example, the number of updating of Torque is stored in _Torque and the number of updating of EngineStatus is stored in _EngineStatus. If the value of an object is read by another object, the number of updating of the former object read by the latter object is also stored in a variable. For example, the number of updating of EngineStatus read by Torque is stored in EngineStatus__Torque and the number of updating of EngineStatus read by ThrottleOpening is stored in EngineStatus__ThrottleOpening.

If a value is read by two or more objects belonging to the same controller, the number of updating of the value read by each object is checked for verification by an assertion. In this example, the number of updating of EngineStatus read by Torque must be equal to the number of updating of EngineStatus read by ThrottleOpening. So the assertion to check that EngineStatus__Torque is equal to EngineStatus__ThrottleOpening is located in ThrottleController(). We can detect the violation during random simulation with the Promela code.

The Promela code shown above is useful to find when and where data consistency violations occur. However, the Promela code that contains infinite loops cannot be used to verify that no violation occurs, i.e. the correctness of the model. For example, the Promela code for random simulation cannot verify the correctness of the modified implementation model represented by Fig. 3, Fig. 7, Fig. 8 and Fig. 9. So we use another Promela code with LTL (Linear Temporal Logic) formula for correctness verification.

Fig. 11 shows Provera code for correctness verification of the implementation model represented by Fig. 3, Fig. 4, Fig. 5 and Fig. 7. A task is represented by a process with no infinite loop. Some parts of the code is omitted for simplicity.

The number of updating of the value of an object read by another object is stored in two variables. For example, the number of updating of EngineStatus read by Torque is stored
in \texttt{EngineStatus\_Torque} and \texttt{used\_EngineStatus\_Torque}. The number of updating of \texttt{EngineStatus} read by \texttt{ThrottleOpening} is stored in \texttt{EngineStatus\_ThrottleOpening} and \texttt{used\_EngineStatus\_ThrottleOpening}. The storing in \texttt{used\_EngineStatus\_Torque} and the storing in \texttt{used\_EngineStatus\_ThrottleOpening} are atomically executed in \texttt{ThrottleController()}. 

The LTL formula used for correctness verification is $[p]$, which means that the proposition $p$ is always true. The never claim for $![p]$ is located at the end of the code. The proposition $p$ is a conjunction of conditions. The proposition $p$ of this example is that \texttt{used\_EngineStatus\_Torque} is equal to \texttt{used\_EngineStatus\_ThrottleOpening}.

We can generate a model checker from the Promela code with the never claim by SPIN and perform correctness verification. For example, we get one error result by the model checker for the implementation model represented by Fig. 3, Fig. 4, Fig. 5 and Fig. 7 and no error result by the model checker for the modified implementation model represented by Fig. 3, Fig. 7, Fig. 8 and Fig. 9.

\section*{B. Promela Code Generation Tool}

We have developed a Promela code generation tool, which inputs an implementation model and generates two kinds of Promela code described in the previous section: one for random simulation, the other for correctness verification. We have applied the Promela code generation tool to a number of software models transformed from Simulink models: a fuel injections system, a hybrid electric vehicle system, a stepping motor control system and an engine speed control system, which are provided by the MathWorks, Inc.[1]. When data consistency violations were detected, we modified the models and verified the data consistency of the modified models. Through the experiments, we have confirmed that the tool is useful for the data consistency verification of embedded control software.

\section*{iv. Conclusion}

We have presented a method to verify the data consistency of embedded control software with SPIN model checker. We have also presented a tool that automatically generates Promela code for data consistency verification. We have applied the tool to a number of software models transformed from Simulink models and have confirmed its usefulness for embedded control software design.

\section*{References}


