An Embedded Control Software Development Environment with Simulink Models and UML Models

Tatsuya Kamiyama, Masayoshi Tamura, Takahiro Soeda, Myungryun Yoo, and Takanori Yokoyama

***Abstract*—The paper presents an embedded control software development environment with Simulink models and UML models. The embedded control software development process consists of the control logic design phase, the software design phase, and the programming phase. MATLAB/Simulink is widely used to build a controller model in the control logic design phase. On the other hand, UML is widely used in the software design phase. To shift from the control logic design phase to the software design phase smoothly, we have developed a model transformation tool to transform a Simulink model to a UML model. The UML model generated by the transformation tool consists of classes that encapsulate data and calculation methods of the data. To improve the reusability of the classes, the Simulink model should be well-layered. We have also developed a layering support tool for efficient layering of the Simulink model. Code generation tools are used to generate source programs from Simulink models and UML models in the programming phase. We have developed a code composition tool to integrate the code generated from UML models and the code generated from Simulink models. We have applied those tools to a number of Simulink models and found it useful for embedded control software design.**

***Index Terms*—embedded software, model-based design, software tools, control systems, real-time systems.**

I. Introduction

THE embedded control software development process can be divided into the control logic design phase, the software design phase, and the programming phase. In the control logic design phase, control engineers design control logic, just considering functional properties. In the software design phase, software engineers design the software structure and behavior to implement the control logic, considering not only functional properties but also nonfunctional properties. Recently, code generation tools are used to generate source programs in the programming phase.

Model-based design has become popular in embedded control software design, especially in the automotive control domain. In model-based design, 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]. CAD/CAE tools such as MATLAB/Simulink should be used for just control logic design, not for software design. Software modeling languages such as UML should be used for software design. The reusability of the source code generated directly from a controller model is not good because we have to write glue code to integrate the code modules generated independently. We also have to add the code for synchronization, mutual exclusion, or inter-task communication to execute the generated code correctly in the preemptive multi-task environment. A method to generate more reusable code is required.

A control system consists of various software modules. Simulink models are suitable to represent control logics such as feedback control and feedforward control. On the other hand, UML is suitable for some software modules such as application modules with procedural algorithms, input and output modules, and network communication modules. To integrate those models, Simulink models should be transformed to UML models before the integration because UML is suitable for software design. UML provides a number of kinds of diagrams, which are useful for not only functional design but also nonfunctional design. The source code should be generated after the nonfunctional design.

Ramos-Hernandez et al. have presented a tool that transforms a Simulink model to a UML model[3][4]. The tool generates classes corresponding to each blocks of the Simulink model. A dependency is generated corresponding to a line that connects blocks. Müller-Glaser et al. have presented a method to transform a Simulink model to a UML model, in which each object of the generated UML model corresponds each element of the Simulink model[5][6]. Blocks, lines and junctions are represented as objects in the UML model. Sjöstedt et al. have presented a tool that transforms a Simulink model to a UML model[7]. The tool generates composite structure diagrams as structural models and activity diagrams as behavior models. However, classes of UML models generated by those tools may not be reusable because each class represents just an element of the original Simulink models. To improve the reusability of the software,

Manuscript received July 6, 2012. This work is supported in part by KAKENHI (20500037, 24500046).

T. Kamiyama was with Graduate School of Engineering, Tokyo City University. He is now with A&D Company, Limited, 1-243, Asahi, Kitamoto-shi, Saitama-ken 364-8585 Japan, 1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan.

M. Tamura is with Graduate School of Engineering, Tokyo City University, 1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan (e-mail: g1181524@tcu.ac.jp).

T. Soeda was with Graduate School of Engineering, Tokyo City University, 1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan.

M. Yoo and T. Yokoyama are with Department of Computer Science, Tokyo City University, 1-28-1, Tamazutsumi, Setagaya-ku, Tokyo 158-8557 Japan (e-mail: {yoo, yokoyama}@cs.tcu.ac.jp).

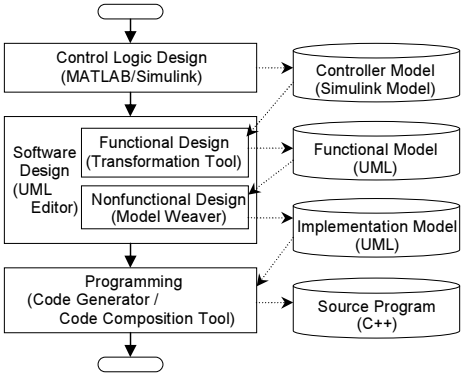

Fig. 1. Development Flow of Embedded Control Software

a UML model should be structured based on the object-oriented concept.

The goal of the research is to develop a software development environment, which transforms Simulink models to reusable UML models and generates reusable source code. To achieve the goal, we define rules to transform a Simulink model to a UML model based on the design method for the time-triggered object-oriented software[8][9][10]. A control systems designed by the design method consists of objects that represent reasonable physical quantities in the control logic, for example, input values, output values, observed values, estimated values, and desired values. We have developed a model transformation tool based on the defined rules[11]. The tool generates UML structural models and behavioral models: class diagrams, object diagrams and sequence diagrams. Each class of the generated UML model corresponds to a reasonable physical quantity in the Simulink model. The method of the class corresponds to the subsystem block in the layered Simulink model. So a subsystem block calculating a physical quantity can be reused as a class. We have also developed a code composition tool to integrate the code generated from UML models and the code generated from Simulink models.

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 the model transformation environment including the model transformation tool and shows model transformation examples. Section IV describes the code generation environment including the code composition tool and shows code generation examples. Section V describes the experiments of the software development environment. Finally, Section VI 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.

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 throttle control part of an automotive control system. The

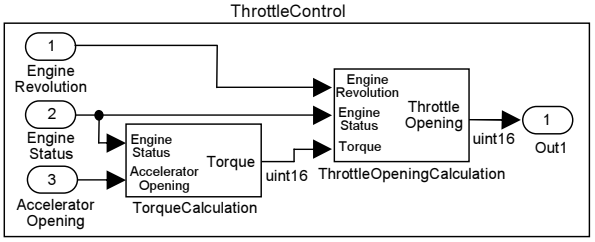

Fig. 2. Example Simulink Model

Simulink model inputs engine revolution, engine status, and accelerator opening, and outputs throttle opening. The model consists of three inport blocks for engine revolution, engine status and accelerator opening, two subsystem blocks to calculate torque and throttle opening, and an outport block for throttle opening. Fig. 2 shows the higher layer model of the layered Simulink model. 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 can be divided into functional design and nonfunctional design. In functional design, we transform a Simulink model into a UML model. We call the UML model the functional model because the model represents implementation-independent control functionalities. A functional model may be integrated with other models built in UML. In nonfunctional design, we build an implementation model taking account of nonfunctional properties.

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

Fig. 3 shows the toolchain for embedded control software development. The shadowed rectangles show the tools we have developed and the non-shadowed rectangles show existing commercial tools.

B. Functional Design

We transform a Simulink model into a functional model represented in UML with a model transformation environment. Our model transformation method is based on the design method of the time-triggered object-oriented software[8][9][10]. A control system designed by the design method consists of objects that correspond to data in the block diagram. The design method identifies objects referring to the data flow of the block diagram representing control logic. The important data representing reasonable physical quantities, such as input values, output values, observed values, estimated values, and desired values, are candidates for objects, because those values are rarely deleted or added even if the detailed control logic is modified[10].

The object representing data is called the value object. The value object encapsulates the data and the calculation method of the value of the data. The value object class has attribute *value*, method *update* that calculates value, and method *get* to read value. Two value objects can be linked with an association named *cons*, which means the relation between a producer object and a consumer object. Fig. 4 shows two value object classes linked with association *cons*. Method *update* of *Consumer* gets the value by calling method

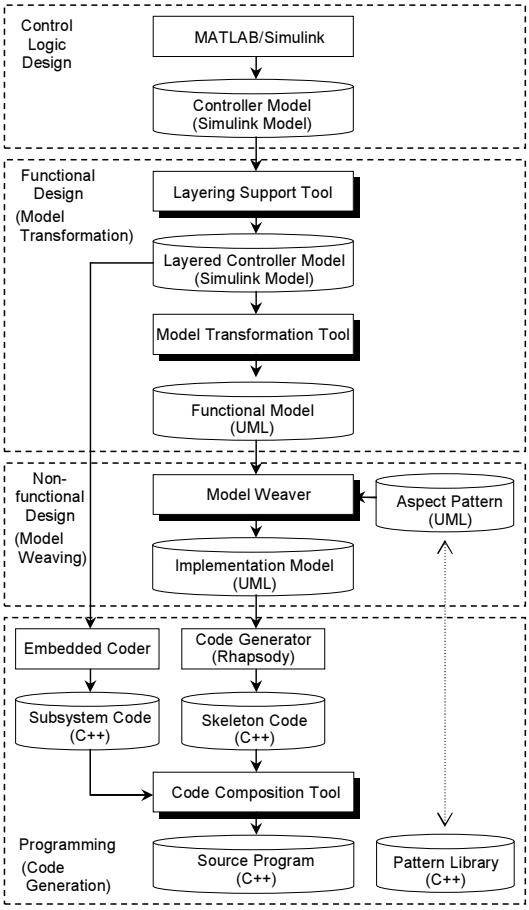

Fig. 3. Toolchain for Embedded Control Software Development

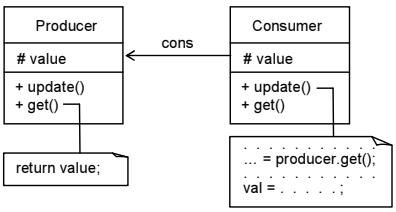

Fig. 4. Two value object classes linked with producer-consumer association

get of *Producer* and calculates its own value and stores the calculated value in attribute *value*.

Fig. 5 shows the class diagram of the functional model corresponding to the Simulink model shown by Fig. 2. The class diagram shown by Fig. 5 consists of six value object classes: *EngineRevolution*, *EngineStatus*, *AcceleratorOpening*, *Torque*, *ThrottleOpening*, and *ThrottleControl*. Related value object classes are linked with *cons* associations. For example, method *update* of class *Torque* gets the value of *EngineStatus* and the value of *AcceleratorOpening*, calculates its own value, and stores the calculated value in attribute *torque*. *ThrottleControl* is a whole object, which corresponds to the whole Simulink model shown by Fig. 2. Method *exec* of *ThrottleControl*, which is periodically executed in the control period, calls method *update* of class *Torque* and method *update* of class *ThrottleOpening*.

The transformation from a Simulink model to a UML model is performed with the model transformation environment shown in Fig. 3. The details of the environment are described in Section III

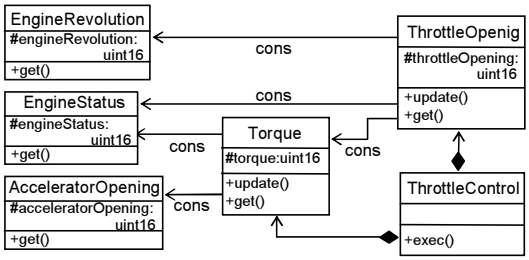

Fig. 5. Example Class Diagram of Functional Model

C. Nonfunctional Design

An embedded control system is a hard real-time system with timing constraints. We design the task structure, scheduling policy, task priorities to meet timing constraints in nonfunctional design. We may also add mechanisms such as synchronization, mutual exclusion, and inter-task communication to the model so that the software correctly executes in the preemptive multi-task environment. Aspect-oriented programming[12] has been applied to separate non-functional properties from functional properties. Model level aspects for non-functional requirements have also been presented[13][14].

Our nonfunctional design is based on the aspect-oriented design method we have already presented[15][16]. We have also presented aspect patterns for nonfunctional properties of embedded control software and developed a model weaver to weave the aspect patterns into the functional model. For example, mechanisms for triggering methods (time-triggered or event-triggered[17]), synchronization, and inter-task communications are defined as aspect patterns. As shown in Fig. 3, we select aspect patterns and weave them into the functional model with the model weaver to get the implementation model.

We consider the case in which the calculation of the values of *EngineRevolution*, *EngineStatus*, and *AcceleratorOpening* is executed by one periodic task and the calculation of the values of *TargetTorque* and *ThrottleOpening* is executed by another periodic task. If the priority of the former task is higher than the priority of the latter task, the latter task may be preempted by the former task. So a mechanism of mutual exclusion or inter-task communication is needed for data integrity. Here, we use buffering mechanism, which is one of wait-free inter-task communications.

Fig. 6 shows the class diagram of the structural aspect pattern of buffering, which connects a producer object and a consumer object. Class *ProducerBuffer* has attribute *buf* to store the value, method *update* to get the value from *Producer* and store the value in *buf*, and method *get* for *Consumer* to get the value stored in *buf*. The class diagram of the aspect pattern is enclosed by a package with stereotype $<<aspect>>$, which represents that the enclosed diagram is an aspect. The behavioral aspect pattern of buffering is also defined with a sequence diagram.

The binding expression is written under $<<aspect>>$. Crosscutting elements of the aspect pattern are represented as variables (variable elements), which are to be bound with the actual elements of the base model. In this case, *Producer*, *Consumer*, *Relation*, and *Controller* are variables. The binding expression means that variable *Producer* is to be

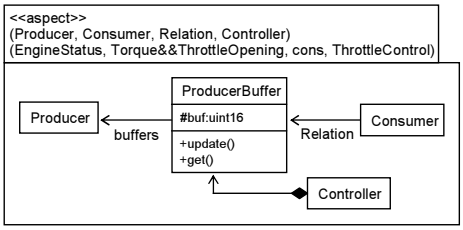

Fig. 6. Aspect Pattern of Buffering

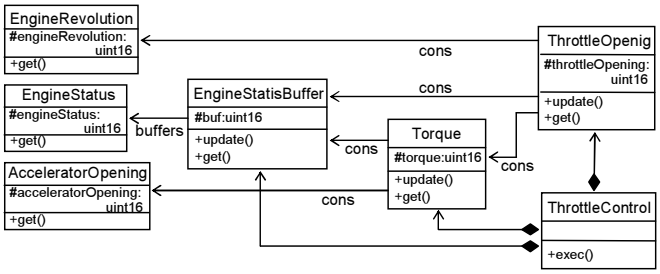

Fig. 7. Example Class Diagram of Implementation Model

bound with *EngineStatus*, variable *Consumer* is to be bound with *TargetTorque* and *ThrottleOpening*, variable *Relation* is to be bound with *cons*, and variable *Controller* is to be bound with *ThrottleControl*. We put the class diagram shown by Fig. 5 and the aspect shown by Fig. 6 into the input of the model weaver, and we get the woven class diagram shown by Fig. 7.

D. Programming

Source programs can be generated from an implementation model. Fig. 8 shows the generated source program of class *ThrottleOpening* shown in Fig. 7. The code of Fig. 8 is simplified for explanation purposes. File *ThrottleOpening.h* is a header file that defines member functions (methods) and data members (attributes) of class *ThrottleOpening*. The value of throttle opening is stored data member *throttleOpening*. The pointers to related objects, *EngineRevolution*, *EngineStatus* (*EngineStatusBuffer* in the implementation model), and *Torque* are stored in data members *itsEngineRevolution*, *itsEngineStatus*, and *itsTorque*.

The code of the member functions are defined in file *ThrottleOpening.cpp*. Method *update* gets the values of *EngineRevolution*, *EngineStatus* (*EngineStatusBuffer* in the implementation model), and *Torque* by calling their *get* functions, calculates its own value, and stores the calculated value in attribute *throttleOpening*.

Fig. 9 shows the generated source program of class *ThrottleControl*. Member function *exec* calls *update* functions of class *EngineStatusBuffer*, class *Torque*, and class *ThrottleOpening*.

The generated source programs can be used to build a embedded control system without glue code. The source file of each class can be also reused for another system. The source programs are generated by the code generation environment shown in Fig. 3. The details of the environment are described in Section IV.

ThrottleOpening.h

```
class ThrottleOpening {
  public:
    ThrottleOpening( . . . );
    void update();
    short get();
  protected:
    short throttleOpening;
    EngineRevolution* itsEngineRevolution;
    EngineStatus* itsEngineStatus;
    Torque* itsTorque;
};
```

ThrottleOpening.cpp

```
#include "ThrottleOpening.h"

void ThrottleOpening::update() {
  throttleOpening =
    . . . . . .
    . . (*itsEngineRevolution).get() . .
    . . (*itsEngineStatus).get() . .
    . . (*itsTorque).get() . .
    . . . . . .
}

short ThrottleOpening::get() {
  return throttleOpening;
}
```

Fig. 8. Example Source Program of Value Object Class

ThrottleControl.h

```
class ThrottleControl {
  public:
    ThrottleControl( . . . );
    void exec();
  protected:
    EngineStatusBuffer* itsEngineStatusBuffer;
    Torque* itsTorque;
    ThrottleOpening* itsThrottleOpening;
};
```

ThrottleControl.cpp

```
#include "ThrottleControl.h"

void ThrottleOpening::exec() {
  (*itsEngineStatus).update();
  (*itsTorque).update();
  (*itsThrottleOpening).update();
}
```

Fig. 9. Example Source Program of Whole Object Class

III. Model Transformation Environment

A. Layering Support Tool

The target of the transformation to a UML model is the higher layer model of the layered Simulink model, which consists of subsystem blocks, inport blocks, and outport blocks. As described in Section II-B, a value object corresponds to a data in the higher layer model and method *update* of the value object corresponds to the subsystem block that calculates the value of the data. To make a class reusable, the Simulink model should be well-layered before transformation so that subsystem blocks calculating the important data such as reasonable physical quantities are presented at the higher layer.

We have developed a layering support tool to select important data in a Simulink model and layer the Simulink model[11]. Fig. 10 illustrates the layering work with the tool. The layering support tool analyzes an input Simulink model, and shows all data of the Simulink model on the window of the tool. Each data of the Simulink model is shown by a row of the table on the window. Column *System* means the subsystem or the whole model in which the data is presented, column *Src Block* means the source block of the data, column

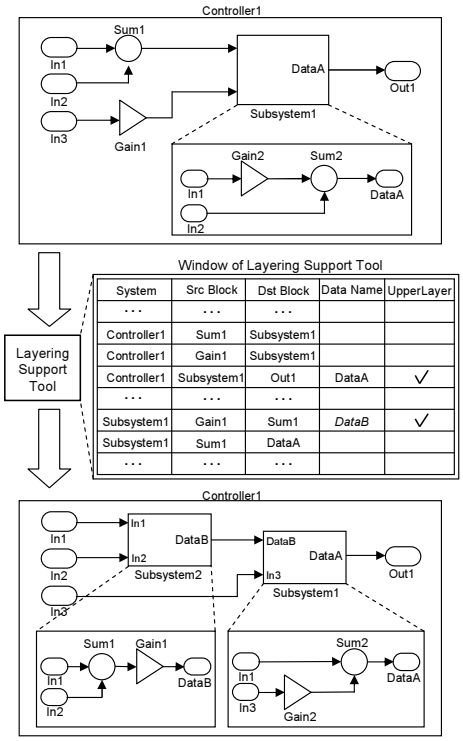

Fig. 10. Layering of Simulink Model

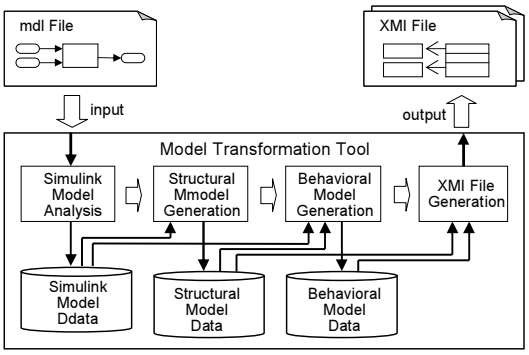

Fig. 11. Model Transformation Tool

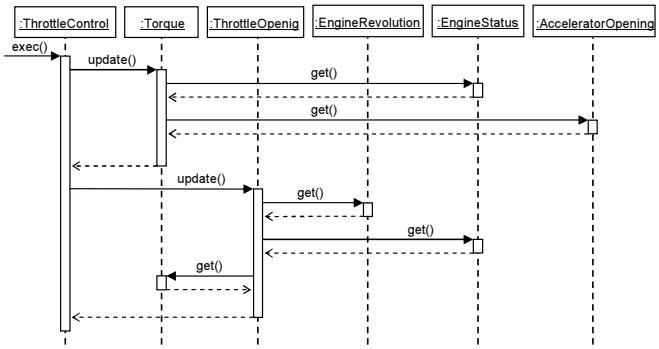

Fig. 12. Example Sequence Diagram of Functional Model

Dst Block means the destination block of the data, and *Data Name* means the name of the data if the data has a name. For example, the third row of the table in Fig. 10 shows the data from *Sum1* to *Subsystem1* with no name in *Controller1* (the whole model).

We can select the data to be presented in the higher layer by checking column *Upper Layer*. In this case, the data named *Data1* from *Subsystem1* to *Out1* in *Controller1* and the data from *Gain1* to *Sum1* in *Subsystem1* are checked. If the checked data has no name, we have to attach the name to the data. In this example, name *DataB* is attached to the data from *Gain1* to *Sum1* in *Subsystem1*. Then, the tool generates a layered Simulink model in which the just the checked data are presented in the higher layer. In this example, there are two subsystem blocks in the higher layer of the generated Simulink model. One subsystem block outputs *DataB* and another subsystem block outputs *DataA*.

B. Model Transformation Tool

We have developed a model transformation tool to transform a layered Simulink model to a UML model. We developed the first version of the tool to generate class diagrams and object diagrams as the UML structural model[18]. A class diagram is generally used to represent the structure of object-oriented software. An object diagram is also useful for the embedded control system, in which most objects are statically created at the initialization process, not dynamically created. Then we have extended the model transformation tool to generate sequence diagrams as the UML behavioral model[11]. A Sequence diagram is used to represent interactions between objects in time sequence.

Fig. 11 shows the internal processing flow of the model transformation tool. The tool inputs an mdl file, which is a file to store the information on a Simulink model. Then the tool analyzes the mdl file and extracts Simulink model data needed for transformation. The tool generates structural model data referring to the Simulink model data. The tool also generates behavioral model data referring to the Simulink model data and the structural model data. Finally, the tool translates the structural model data and the behavioral model data into XMI files. XMI is a standard file format of UML[19].

Fig. 5 shows the generated class diagram from the Simulink model shown by Fig. 2. Fig. 12 shows the generated sequence diagram from the Simulink model shown by Fig. 2. The sequence diagram shows that method *exec* of object *ThrottleControl* calls method *update* of object *Torque* and method *update* of object *ThrottleOpening* sequentially. Method *update* of object *Torque* calls methods *get* of *EngineStatus* and *AcceleratorOpening* to get those values. Method *exec* of *ThrottleControl* is executed periodically in the control period.

IV. Code Generation Environment

A. Code Generation from models

We use Rational Rhapsody Developer[20] to generate C++ skeleton code from a UML model. Another code generator can be used if it generates skeleton code from a UML model. The skeleton code generated from a UML model does not contain the code for calculation. So we extract the code for calculation from the control logic code generated by Embedded Coder and embed the extracted code in the skeleton code of the value object.

Fig. 13 shows the skeleton code of class *ThrottleOpening*. The code of Fig. 13 is simplified for explanation purposes. File *ThrottleOpening.h* is a header file that defines member functions and data members of class *ThrottleOpening*. The

ThrttleOpening.h

```
class ThrottleOpening {
  public:
    ThrottleOpening( . . . );
    void update();
    short get();
  protected:
    short throttleOpening;
    EngineRevolution* itsEngineRevolution;
    EngineStatus* itsEngineStatus;
    Torque* itsTorque;
};
```

ThrttleOpening.cpp

```
#include "ThrottleOpening.h"

void ThrottleOpening::update() {

}

short ThrottleOpening::get() {

}
```

Fig. 13. Example Code Generated from UML Model

ThrttleOpeningCalculation.h

```
. . . . . .
typedef struct {
    . . . . . .
} ExternalInputs_ThrottleOpeningCalculation;

typedef struct {
    . . . . . .
} ExternalOutputs_ThrottleOpeningCalculation;

struct Parameters_ThrottleOpeningCalculation {
    . . . . . .
}
. . . . . .
```

ThrttleOpeningCalculation_data.cpp

```
. . . . . .
Parameters_ThrottleOpeningCalculation ThrottleOpeningCalculation_P = {
    . . . . . .
    . . . . . .
}
. . . . . .
```

ThrttleOpeningCalculation.cpp

```
. . . . . .
ExternalInputs_ThrottleOpeningCalculation ThrottleOpeningCalculation_U;
ExternalOutputs_ThrottleOpeningCalculation ThrottleOpeningCalculation_Y;

Static void ThrottleOpeningCalculation_step()
{
  ThrottleOpeningCalculation_Y.ThrottleOpening =
    . . . . . .
    . . ThrottleOpeningCalculation_P...  . . .
    . . ThrottleOpeningCaclculation_U.EngineRevolution  . . .
    . . ThrottleOpeningCaclculation_U.EngineStatus  . . .
    . . ThrottleOpeningCaclculation_U.Torque  . . .
    . . . . . .
}
. . . . . .
```

Fig. 14. Example Code Generated from Simulink Model

skeletons of function *update* and function *get* are defined in file *ThrottleOpening.cpp*.

Fig. 14 shows the generated subsystem code from the lower layer Simulink model of subsystem block *ThrottleOpeningCalculation* in Fig. 2. The code of Fig. 14 is simplified for explanation purposes. The data types used in the generated code are defined in file *ThrottleOpeningCalculation.h*. The values of the parameters are defined in file *ThrottleOpeningCalculation_data.cpp*. File *ThrottleOpeningCalculation.cpp* contains function *ThrottleOpeningCalculation_step* that calculates the output value of the subsystem block.

We have developed a code composition tool, which integrates the skeleton code and the control logic code and generates complete source programs of value object classes.

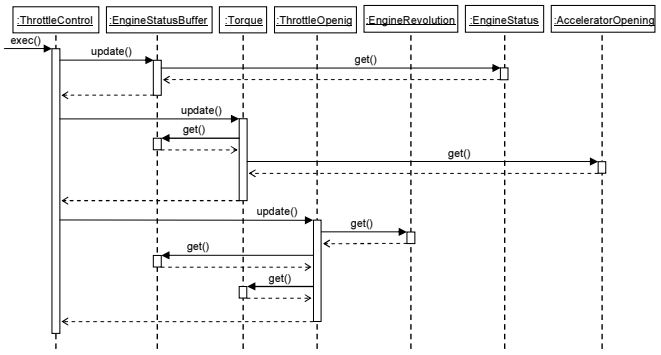

Fig. 15. Example Sequence Diagram of Implementation Model

Buffer.h

```
template<class P, typename V>
class Buffer {
  V buf;
  P* itsProducer;
public:
  Buffer(P* p) : itsProducer(p) {}
  void update();
  V get();
};
```

Buffer.cpp

```
#include "Buffer.h"

template<class T, typename V>
void Buffer<T, V>::update() {
  buf = (*itsProducer).get();
}

template<class T, typename V>
V Buffer<T, V>::get() {
  return buf;
}
```

EngineStatusBuffer.h

```
#include "Buffer.h"
#include "EngineStatus.h"

template class SimpleBuffer<EngineStatus, short>;

typedef SimpleBuffer<EngineStatus, short> EngineStatusBuffer;
```

Fig. 16. Example Code of Pattern Library

The code for calculation is embedded in member function *update* by the code composition tool. The details of the code composition tool are described in Section IV-B.

The source program of a whole object class can be generated just by a UML tool with code generation functions such as Rational Rhapsody Developer because the code of method *exec* can be generated from a sequence diagram executing the method. The model transformation tool generates the sequence diagram of the whole object. The sequence diagram may be modified by the model weaver in nonfunctional design. For example, we get the sequence diagram of the implementation model shown by Fig. 15 by weaving the behavioral aspect pattern of buffering into the sequence diagram of the functional model shown by Fig. 12[16]. The source program of class *ThrottleControl* shown by Fig. 9 can be generated from the class diagram shown by Fig. 7 and the sequence diagram shown by Fig. 15.

The source programs of classes corresponding to aspect patterns are provided by the pattern library. Fig. 16 shows the source program for the buffering pattern. A class for buffering is defined as class template *Buffer*. A concrete class for buffering the value of *EngineStatus* is defined as template class *EngineStatusBuffer*, which is generated from class template *Buffer*.

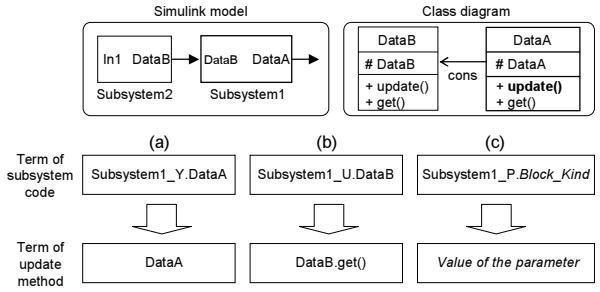

Fig. 17. Rewriting Rules for Code Composition

B. Code Composition Tool

We have developed a code composition tool to generate complete C++ source files of a value object class. The code composition tool inputs the C++ files of the skeleton code generated by Rational Rhapsody Developer and the C++ files of the subsystem code generated by Embedded Coder. Then the code composition tool extracts the code for calculation of the subsystem, rewrites the code to fit the class structure, and embeds the rewritten code in the skeleton of function *update*. Finally, the code composition tool outputs complete C++ files of the class.

Fig. 17 shows rules to rewrite the code. Column (a) shows the rule to replace the outport block name of a subsystem block (*Subsystem1_Y.DataA* in this case) with the attribute name of the corresponding class (*DataA* in this case). Column (b) shows the rule to replace the inport block name of a subsystem block (*Subsystem1_U.DataB* in this case) with the corresponding *get* method call (*DataB.get()* in this case). Column (c) shows the rule to replace a parameter name with its value.

For example, the code composition tool inputs the skeleton code shown by Fig. 13 and the subsystem code shown by Fig. 14 and outputs the source program shown by Fig. 8. The code in function *ThrottleOpeningCalculation_step* of Fig. 14 is rewritten and embedded in member function *update* of class *ThrottleOpening* of Fig. 8.

V. Experiments

We have applied the model transformation environment to a number of Simulink models: a fuel injections system, a hybrid electric vehicle system, a stepping motor control system, and a engine speed control system, which are provided by the MathWorks, Inc.[1].

At first, we made the original Simulink models layered with the layering support tool. Table I shows the number of blocks of the layered Simulink models used in the experiments. Column *Subsystem* shows the number of subsystem blocks, the column *Inport* shows the number of inport blocks, and the column *Outport* shows the number of outport blocks. Then we transformed the layered Simulink models to class diagrams, object diagrams, and sequence diagrams using the model transformation tool.

We show the case of a hybrid electric vehicle system. The example hybrid electric vehicle is a series-parallel hybrid electric vehicle that consists of a gasoline engine and an electric motor. Fig. 18 shows the higher layer of the layered Simulink model of the hybrid electric vehicle system. Fig. 19 shows the generated class diagram and the object diagram.

TABLE I
Models Used in Experiments

Target System	Number of Blocks		
	Subsystem	Inport	Outport
Fuel Injections	15	4	1
Hybrid Electric Vehicle	30	6	5
Stepping Motor Control	8	1	4
Engine Speed Control	5	2	1

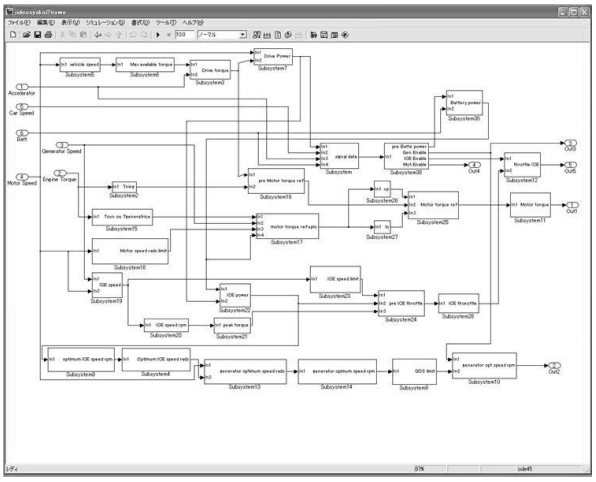

Fig. 18. Simulink Model of Hybrid Electric Vehicle System

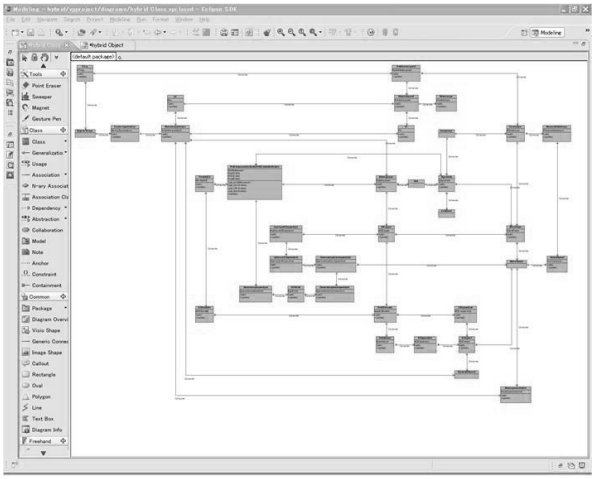

Fig. 19. Class Diagram of Hybrid Electric Vehicle System

The class diagram consists of thirty-seven classes. Fig. 20 shows the generated sequence diagram.

The original Simulink models used in the experiments represent just control logics. They are built by control engineers without considering implementation. After layering the original models, the transformation tool successfully transforms the layered Simulink models to class diagrams, object diagrams, and sequence diagrams. So we think the transformation tool can be applied to embedded control software design.

We have also applied the code generation environment to the Simulink models and UML models shown above and successfully get the C++ source programs of the classes. Fig. 21 shows example C++ source programs of value object classes, in which the code generated by Embedded Coder is embedded in *update* methods of the classes.

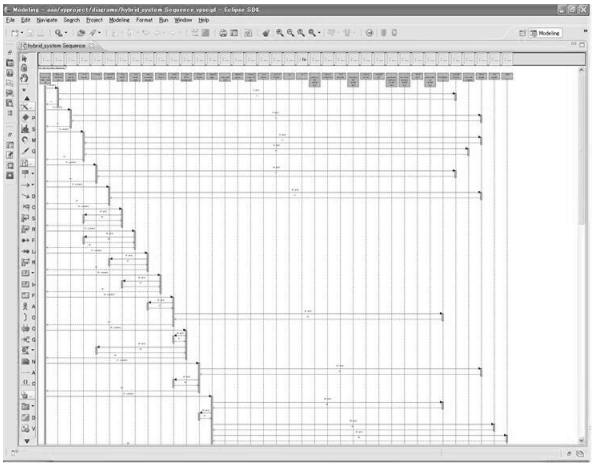

Fig. 20. Sequence Diagram of Hybrid Electric Vehicle System

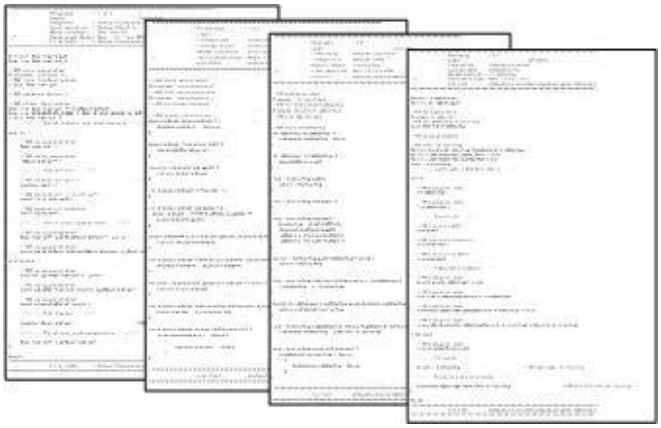

Fig. 21. C++ Source Program of Value Object Classes

VI. Conclusion

We have developed an embedded control software development environment with a Simulink model layering support tool, a Simulink to UML model transformation tool, and a code composition tool. The model transformation tool generates class diagrams, object diagrams and sequence diagram. Each class of a generated UML model corresponds to a data in the Simulink model and the method of the class corresponds to the subsystem block that calculates the value of the data. The code composition tool integrates the code generated from UML models and the code generated from Simulink models, and generates the complete source programs of the classes. We have applied the environment to a number of Simulink models and found it useful for embedded control software development.

We are going to extend the model transformation tool to generate a state machine diagram to make software design more efficient and to deal with Simulink models with Stateflow charts.

References

[1] The MathWorks Inc., http://www.mathworks.com/.

[2] Sangiovanni-Vincentelli, A. and Di Natale, M., Embedded System Design for Automotive Applications, *IEEE Computer*, Vol.40, No.10, 2007, pp.42–51.

[3] Ramos-Hernandez, D. N., Fleming, P. J., Bennett, S., Hope, S., Bass, J. M. and Baxter, M.J., Process Control Systems Integration Using Object Oriented Technology, *Proceeding of Technology of Object-Oriented Languages and Systems TOOLS 38*, 2001, pp.148–158.

[4] Ramos-Hernandez, D. N., Fleming, P. J. and Bass, J. M., A Novel Object-Oriented Environment for Distributed Process Control Systems, *Control Engineering Practice*, vol.13, Issue 2, 2005, pp.213–230.

[5] Kühl, M., Spitzer, B. and Müller-Glaser, K. D., Universal Object-Oriented Modeling for Rapid Prototyping of Embedded Electronic Systems, *Proceedings of the 12th IEEE International Workshop on Rapid System Prototyping*, 2001, pp.149–154.

[6] Müller-Glaser, K. D., Frick, G., Sax E. and Kühl, M., Multiparadigm Modeling in Embedded Systems Design, *IEEE Transactions on Control Systems Technology*, Vol.12, No.2, 2004, pp.279–292.

[7] Sjöstedt, C.-J., Shi, J., Törngren, M., Servat, D., Chen, D., Ahlsten, V. and Lönn, H., Mapping Simulink to UML in the design of embedded systems: Investigating scenarios and structural and behavioral mapping, *OMER 4 Post Workshop Proceedings*, 2008.

[8] Yokoyama, T., Naya, H., Narisawa, F., Kuragaki, S., Nagaura, W., Imai, T. and Suzuki, S., A Development Method of Time-Triggered Object-Oriented Software for Embedded Control Systems, *Systems and Computers in Japan*, Vol.34, No.2, 2003, pp.338–349.

[9] Yokoyama, T., An Aspect-Oriented Development Method for Embedded Control Systems with Time-Triggered and Event-Triggered Processing, *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Application Symposium*, 2005, pp.302–311.

[10] Yoshimura, K., Miyazaki, T., Yokoyama, T., Irie, T. and Fujimoto, S., A Development Method for Object-Oriented Automotive Control Software Embedded with Automatically Generated Program from Controller Models, *2004 SAE World Congress*, 2004-01-0709, 2004.

[11] Tamura, M., Kamiyama, T., Soeda, T., Yoo M. and Yokoyama, T., A Model Transformation Environment for Embedded Control Software Design with Simulink Models and UML Models, *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2012*, IMECS 2012, 14-16 March, 2012, Hong Kong, pp.795–800.

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. Loingtier, J. M. and Irwin, J., Aspect-Oriented Programming, *Proceedings of 11th European Conference on Object-Oriented Programming*, 1997, pp.220–242.

[13] Wehrmeister, M. A., Freitas, E., Pereira, C. E. and Wagner, F. R., An Aspect-Oriented Approach for Dealing with Non-Functional Requirements in a Model-Driven Development of Distributed Embedded Real-Time Systems, *Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2007, pp.428–432.

[14] Driver, C., Reilly, S., Linehan, E., Cahill, V. and Clarke, S., Managing Embedded Systems with Aspect-Oriented Model-Driven Engineering, *ACM Transactions on Embedded Computing Systems*, Vol.10, No.2, 2010, pp.21:1–26.

[15] Soeda, T., Yanagidate, Y. and Yokoyama T., Embedded Control Software Design with Aspect Patterns, *Proceedings of International Conference on Advanced Software Engineering and Its Applications 2009*, 2009, pp.34–41.

[16] Soeda, T., Yanagidate, Y. and Yokoyama T., Embedded Control Software Design with Aspect Patterns, *Journal of the Chinese Institute of Engineers*, vol.34, Issue 2, 2011, pp.213–225.

[17] Kopetz, H., Should Responsive Systems be Event-Triggered or Time-Triggered?, *IEICE Transaction on Information & Systems*, Vol.E76-D, No.11, 1993, pp.1325–1332.

[18] Kamiyama, T., Soeda, T., Yoo, M. and Yokoyama, T., A Simulink to UML Transformation Tool for Embedded Control Software Design, *Proceedings of 2010 International Conference on Computer and Software Modeling*, 2010, pp.93–97.

[19] Object Management Group, *XML Metadata Interchange Specification*, Version 2.0.1, 2005.

[20] International Business Machines Corp., http://www.ibm.com/us/en/.