State Oriented Programming

Hideki Nomoto
Massachusetts Institute of Technology
Aeronautical and Astronautical Engineering
77 Massachusetts Avenue, Cambridge, MA
nomo@mit.edu

Abstract

This paper describes a methodology called "State Oriented Programming" to generate code from state-machine based formal design specification. The methodology is focused on the safety critical systems. In order to achieve highly reliable system development, the following several key techniques are presented: 1) "State-of-interest" based design technique which seamlessly incorporates system/hazard analysis into the design and executable code 2) Automated "transparent" code generation technique that mitigates unexpected behaviors of the code 3) Fully deterministic "backward" execution mechanism of the generated code

1. Introduction

Safety critical system designing is a parallel efforts of achieving functional efficiency within a limited resource and mitigation of possible hazards. The two processes shall be performed concurrently in order to obtain the best trade-off balance between these often conflicting objectives. State Oriented Design can offer solution for this goal.

2. State Oriented Design

As shown in the Figure 1, system model derived from the root node "Attitude" will be divided into two trunks: nominal tree and off-nominal tree. Any targets of control in a system (such as "Attitude" or "Trajectory") have pre-conditions for nominal state and off-nominal state. Logically complete state space is composed of at least two of these sets of conditions. Functional efficiency needs to be achieved within the envelope of the conditions defined in the nominal tree, while safety features shall offer controls for each defined state in the off-nominal tree. Concurrent engineering between two design groups can be harmonized by this binary tree architecture.

For the actual design process, we used SpecTRM-RL modeling language shown in Figure 2. The dual trunk design can be fully verified for its completeness and consistency using its model-checking algorithms[1]. On top of the verified design, in the next section, we developed the automated code generation subsystem.

3. Code Generation

Core philosophy of our code generation subsystem is to offer "transparent" relationship between the design and the code. Avoiding complicated or hidden execution mechanism in the code can help assured safety of the design being applicable to the executable code. Addition to that, since the code is "transparent", it can be automatically generated from the model. Basic generation is conducted by one-by-one mapping of the code from the model as shown in Figure 2. Reflecting the design architecture, generated code will be executed using only structured procedure calls in its pragmatics. Difference between traditional procedure decomposition techniques and State Oriented Programming is in its semantics. The next section describes it.
4. Backward Execution

The hardest problem for real time system design often comes from timing issues. Programming technique itself does not guarantee elimination of this kind of problem. Only proper design can mitigate it. The timing issue can be raised by an indeterminate execution branch depending on the timing deviation of inputs. If one of prerequisite conditions changes when it is not expected, resulting output could be affected severely. State Oriented Programming methodology provides "Backward Execution Mechanism" which permits obtaining input data only when it is expected.

The program starts from the main loop routine which is the top node of the model. In the example of Figure 1, it is "Attitude" of the spacecraft. The generated code is executed "backward" from the root node down to the leaves (input values) of the inverted tree. After completion of this searching of the tree, required procedures are determined at the root node. If the result of the searching meets the preconditions defined for the nominal attitude state, the triggered procedure will generate a guidance command. If it does not meet, the triggered procedure will generate a fault protection sequence.

The problems of this architecture are 1) unnecessary inquiry for child nodes can happen when input frequency is less than inquiry frequency 2) if the same input value is referred from multiple state definitions, change of the input value in the middle of the searching can lead to incoherence of the final determination of the current state of the top node. In the next section, solutions for these problems using "backward reachability algorithm" (BRA) will be discussed.

4.1. Code Optimization

BRA[2] generates tree from the pre-defined state definition. From each state node, the search is performed and relevant input variables are identified. Then the flags for input-update status are included in the code at the time of generation. Each node checks the flags prior to making call for its child node. This mechanism eliminates the unnecessary call for the child node which has not been updated.

The second problem is resolved using another function of the BRA which identifies single failure points in the fault tree. It searches the space for a node which is referred by multiple ancestor nodes. If such a node is found, instead of inquiring the latest value for this single failure point variable, a copy of the value stored at the root node is used in the child nodes.

5. Conclusion

Code generation technique based on the State Oriented Design was described. Auto code generation method was designed targeting safety critical systems. Further code optimization and reaction sequence generation subsystem will be discussed in future works.

References