

Verification of Multi-Task Software

Toshiaki Aoki

Research Center for Trustworthy e-Society
Japan Advanced Institute of Science and Technology

Introduction

- Many formal verification techniques have been studied for a long time.
 - Some of them are becoming matured so that it can be applied to practical software.
- We focus on the following formal verification techniques.
 - Theorem Proving
 - interactively proving facts using inference rules based on higher order logic.
 - Model Checking
 - automatically check behavior represented as finite states and their transitions.

Introduction

- We can not directly apply those techniques to practical software development.
 - There are gaps between documents/products made in the development and what they deal with.
- We are working on bridging them.
 - Formalizing the document and products.
 - Customizing the verification techniques.
- We are proposing verification methods in the following two fields.
 - UML design models.
 - **Multi-task software on RTOS.**

Multi-Task Software on RTOS

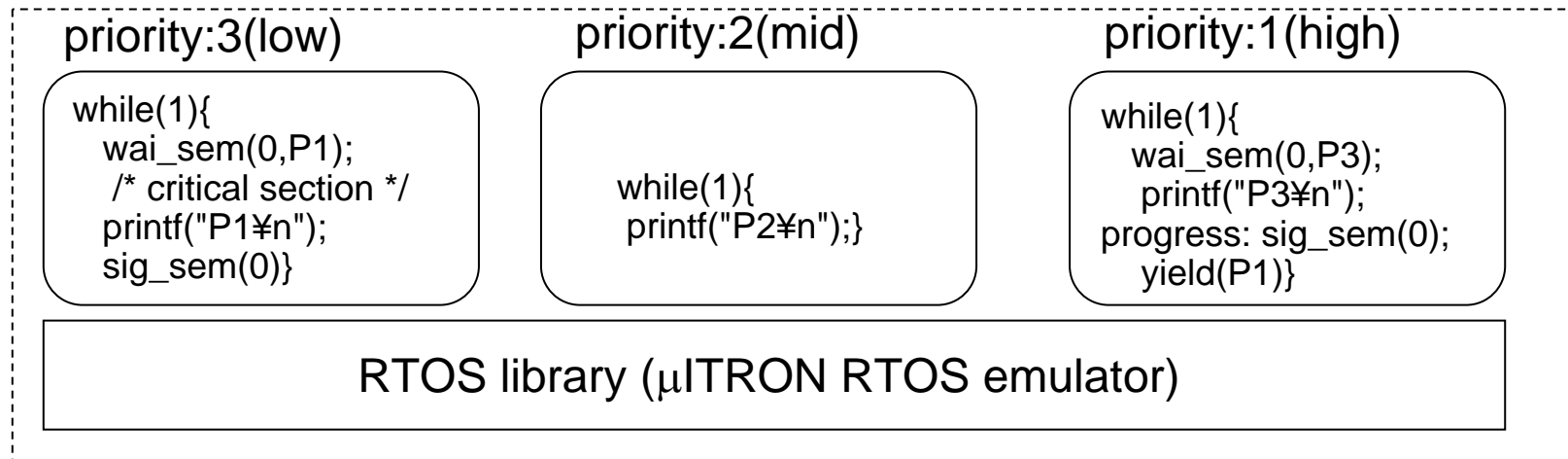
- RTOS is usually used for embedded software.
- Embedded software on RTOS.
 - multi-tasks
 - scheduling primitives
 - priorities, task communications, resource managements, interrupt handlers, etc.
 - periodic execution
- Analysis of such software behavior is very hard.
- Verifying behavior of software on RTOS by model checking.
 - We need deal with the scheduling primitives.
 - Precise analysis.
 - Many false negatives cause if we do not take them into account.
 - We need deal with periods to execute tasks.
 - real-time is complex.
 - We do not need 'real' time if we focus only periodic execution.

RTOS Library

- We have proposed a method to verify behavior of tasks executed on RTOS which conforms to μ ITRON specification.
 - We use Spin model checker.
 - concurrent processes
 - We have implemented a library for software on RTOS.
 - The library emulates behavior of RTOS based on μ ITRON.
 - Counter examples become long because they contain execution sequences of the library.
 - We have Implemented a filter for removing those sequences from the counter examples.
 - Using the libraries, we can describe and verify task behavior with scheduling primitives of μ ITRON.

RTOS Library

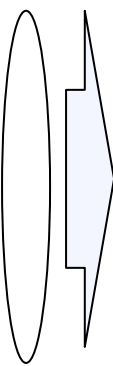
Model Checking target



Model Checker(Spin)

Priority Inversion Problem
is detected.

Counter Example



```
4:  high(3):[wai_sem(0 ,2);now.turn=top();]
P3  high(3):[printf('P3¥n')]
28  high(3):[sig_sem(0);now.turn=top();]
30  high(3):[yield(0)]
32  low(2):[wai_sem(0,0);now.turn=top();]
34  high(3):[wai_sem(0,2);now.turn=top();]
<<<<<START OF CYCLE>>>>>
P2
36:  mid(4):[printf('P2¥n')]
```

Counter Example Filter

RTOS Library

- We do not have the overhead of the state space inserted by the library.
 - The calculation for scheduling tasks is done atomically.
 - The state space to check tasks depends on that of the tasks themselves.
- We experimented our approach by typical examples such as producer-consumer and priority inversion problem.
 - We are applying our approach to middle-scale embedded software.
- We are applying it to a car audio system.
- Prizes.
 - 優秀論文賞, Embedded System Symposium 2005.
 - 山下記念研究賞, 2006.

Periodic Execution

- Tasks on RTOS are often executed periodically.
 - managing devices.
 - guaranteeing their deadlines.
 - To guarantee the deadlines, many approaches such as scheduling theories are proposed.
 - Resource managements are also important.
 - We focus on state (in)consistency among modules.

State Inconsistency

- We constructed a design model of a CD/DVD player in a joint research project with IPA/SEC.
 - It is a typical design model constructed by an engineer who has been developing CD/DVD players.
- Inconsistency problems among states of modules.
 - It is important that the application and driver correctly grasp the state of the drive.
 - The states are inconsistent if they are not equal to each other in some senses.



State Inconsistency

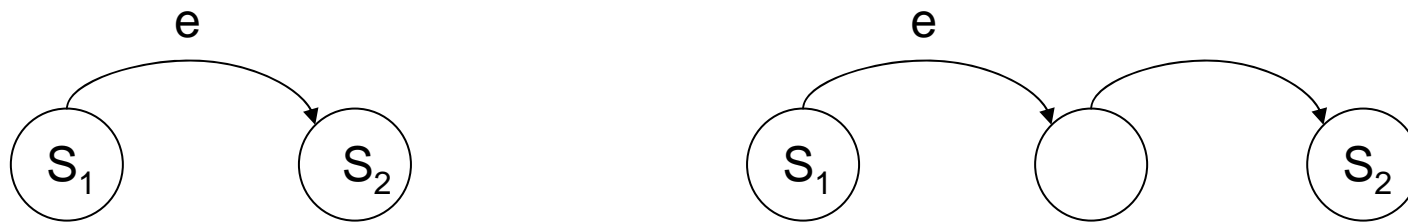
- Many definitions of inconsistencies among states can be considered based on equalities of those states.
 - Skip of a state: The application and driver can not refer to a particular state of the drive.
 - Continuity of difference of states: The state of the application and driver are always different from that of the drive from some point.
 - ...We need to identify state inconsistencies.
- Reasons for state inconsistencies:
 - Mismatch of periods in which the application and driver observe the state of the drive.
 - The timing to update the states of the driver and application is wrong.

Period/Timing Design Model

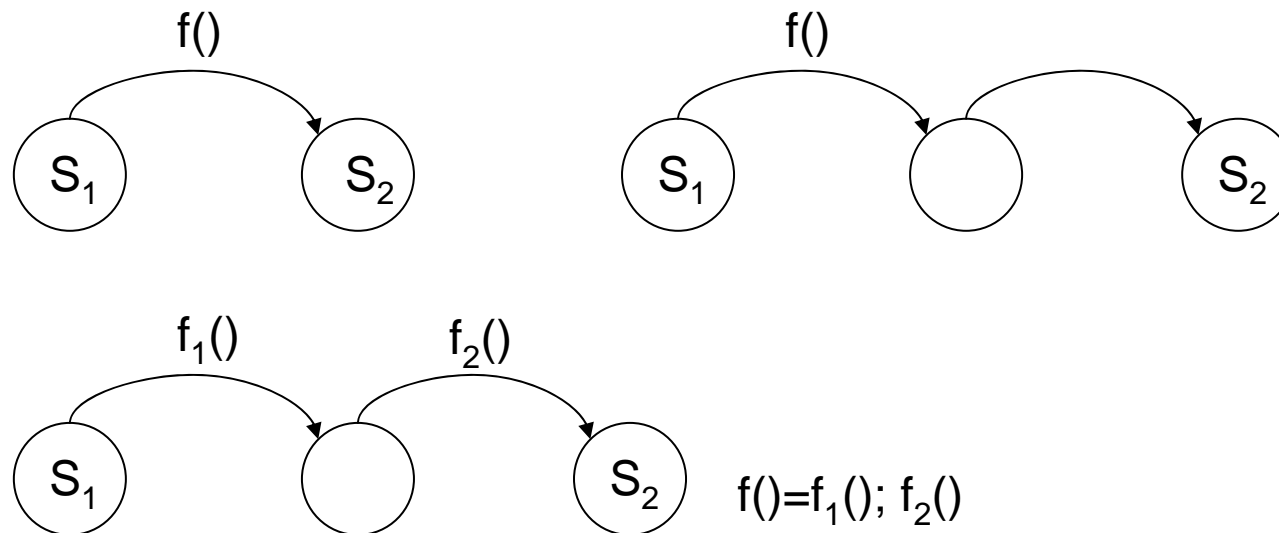
- Rigorously modeling the timing and periods is needed to detect state inconsistencies.
- Design model for timing and periods.
 - We model timing and periods of tasks with state transition diagram which has the following semantics.
 - transitions: impossible to be interleaved with the other transitions.
 - states: possible to be interleaved with transitions.
 - Inter-task communication: reference to shared variables, synchronous function calls and asynchronous message passing.
- Objective: Modeling tasks so that state inconsistencies can not be happened.

Period/Timing Design Model

synchronous and asynchronous transitions triggered by an event

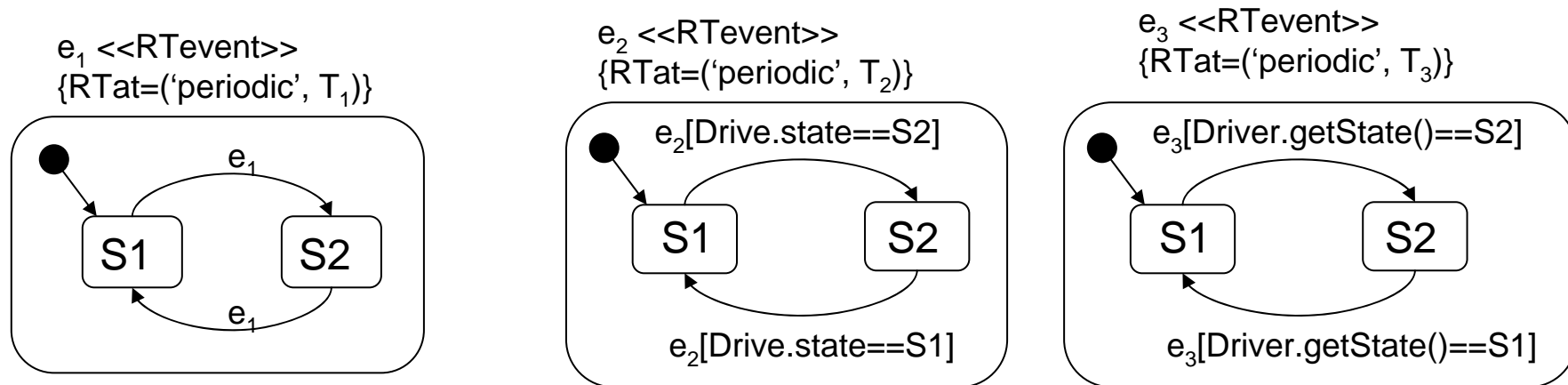


synchronous and asynchronous function calls

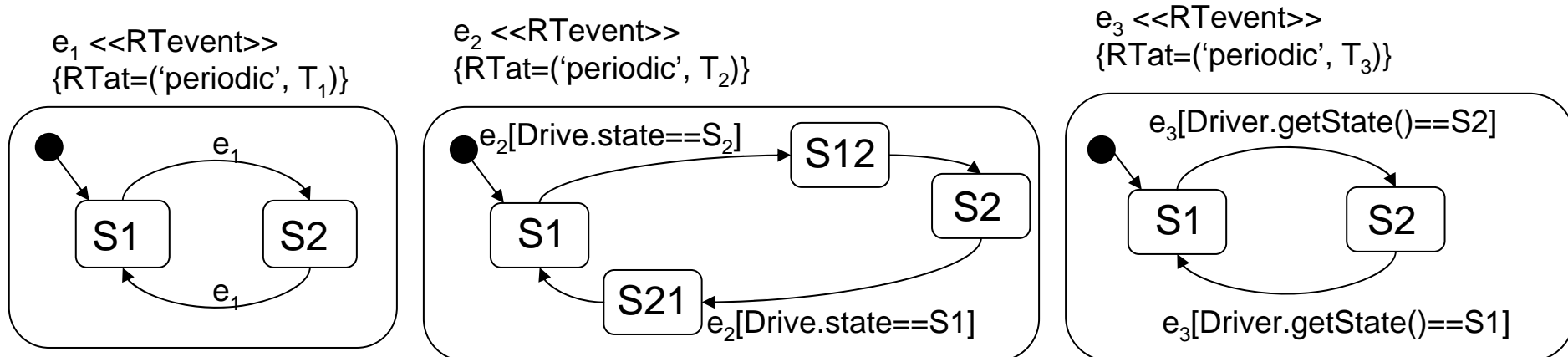


Period/Timing Design Model

Periodic reference to the state of the driver and its synchronous update



Periodic reference to the state of the driver and its asynchronous update



Verification of Design Model

- We detect state inconsistencies by a model checking tool Spin.
- Periodic events.
 - Periodic events are characterized by event sequences.
 - EX) $e_1:2, e_2:3, (e_1e_2e_1(e_1||e_2))^+$
 - $(e_1||e_2) \equiv (e_1e_2) \mid (e_2e_1)$
 - Event sequences which characterize periodic events e_1 and e_2 whose periods are T_1 and T_2 such that $T_1 \leq T_2$ respectively are defined as follows.
 - $(e_1^{k_1}e_2e_1^{k_2}e_2 \dots e_1^{k_n}(e_1||e_2))^+$
 - $k_i = \text{floor}(((T_2 * (i-1) \bmod T_1) + T_2) / T_1)$
 - $(k_1 + \dots + k_n + 1)T_1 = nT_2$

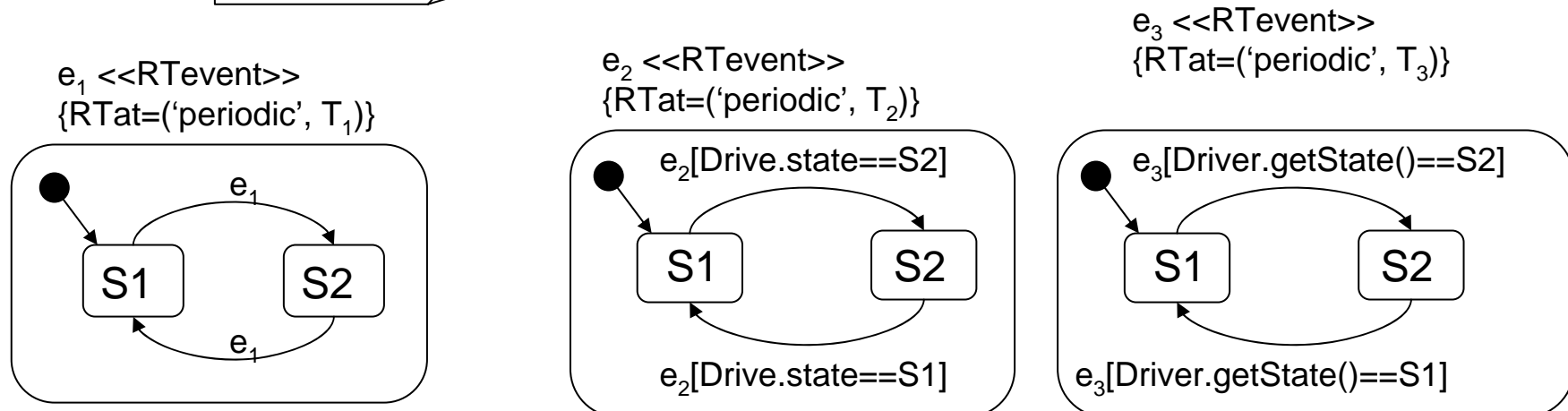
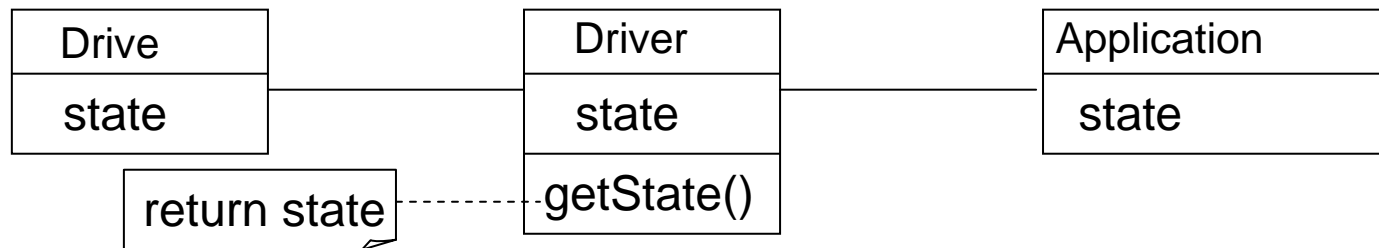
Verification of Design Model

- Verifying the design model by Spin.
 - Each state transition model \rightarrow a process of Spin.
 - Event sequences \rightarrow a process which sends events to the processes of the state transition models based on the event sequences.
 - The events are like clocks.
 - The whole system behave based on the clocks.

Verification of Design Model

Periodic reference and synchronous update

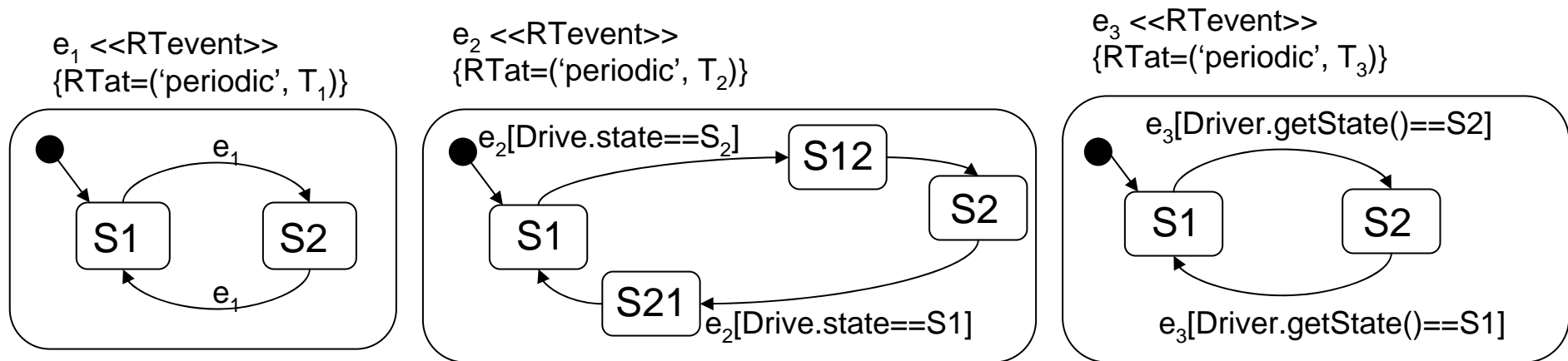
- $T_1=2T_2=4T_3: (e_3(e_2||e_3)e_3(e_1||e_2||e_3))^+ \rightarrow OK$
- $2T_1=3T_2=4T_3: (e_3e_2(e_1||e_3)e_2e_3(e_1||e_2||e_3))^+ \rightarrow OK$
- $T_1=2T_2=2T_3: ((e_2||e_3)(e_1||e_2||e_3))^+ \rightarrow NG(e_1e_3e_2e_1 \text{だと読み飛ばされる})$
- $T_1=3T_2=3T_3: ((e_2||e_3)(e_2||e_3)(e_1||e_2||e_3))^+ \rightarrow OK$



Verification of Design Model

Periodic reference and asynchronous update

- $T_1=2T_2=4T_3$: $(e_3(e_2||e_3)e_3(e_1||e_2||e_3))^+ \rightarrow NG$
- $T_1=3T_2=3T_3$: $((e_2||e_3)(e_2||e_3)(e_1||e_2||e_3))^+ \rightarrow NG$
- $T_1=4T_2=2T_3$: $(e_2(e_2||e_3)e_2(e_1||e_2||e_3))^+ \rightarrow NG$
- $T_1=6T_2=2T_3$: $(e_2(e_2||e_3)e_2(e_2||e_3)e_2(e_1||e_2||e_3))^+ \rightarrow OK$



Detailed Analysis of Design Model

- Periodic reference and synchronous updater
 - The sequence $e_2^*e_3$ should exist between e_1 and e_1 to prevent that the application skips to refer to the state of the drive.
- Periodic reference and asynchronous update
 - The sequence $e_2^*e_2^*e_3$ should exist between e_1 and e_1 to prevent that the application skips to refer to the state of the drive.
 - The first e_2 detects the state change of the driver, then it is reflected to the state of the deriver by the second e_2 .

Detailed Analysis of Design Model

- The general form of the event sequences allows us to obtain event periods from event sequences.
 - EX) $(e_1 e_2 e_1^2 e_2 e_1 (e_1 || e_2))^+$
 - $T_1 < T_2, 5T_1 = 3T_2$
- For more than three events, we can obtain event periods by making and solving equations representing relations between any pair of them.
 - $(e_3 e_2 (e_1 || e_3) e_2 e_3 (e_1 || e_2 || e_3))^+$
 - $(e_3 e_2 e_3 e_2 e_3 (e_2 || e_3))^+ \rightarrow 4T_3 = 3T_2$
 - $(e_2 e_1 e_2 (e_1 || e_2))^+ \rightarrow 3T_2 = 2T_1$
 - $(e_3 (e_1 || e_3) e_3 (e_1 || e_3))^+ = (e_3 (e_1 || e_3))^+ \rightarrow T_1 = 2T_3$
 - Hence, $2T_1 = 3T_2 = 4T_3$

General Form

$$(e_1^{k_1} e_2 e_1^{k_2} e_2 \dots e_1^{k_n} (e_1 || e_2))^+$$

where $k_i = \text{floor}(((T_2 * (i-1) \bmod T_1) + T_2) / T_1)$

$$(k_1 + \dots + k_n + 1)T_1 = nT_2$$

Detailed Analysis of Design Model

Periodic reference and synchronous update

- The sequence $e_2^*e_3$ should exist between e_1 and e_1 to prevent that the application skips to refer to the state of the drive.
- Event sequence 1: $(e_3e_2e_3(e_1||e_2||e_3))^+$ $\rightarrow T_1=2T_2=3T_3$
- Event sequence 2: $(e_2e_3e_2(e_1||e_2||e_3))^+$ $\rightarrow T_1=3T_2=2T_3$

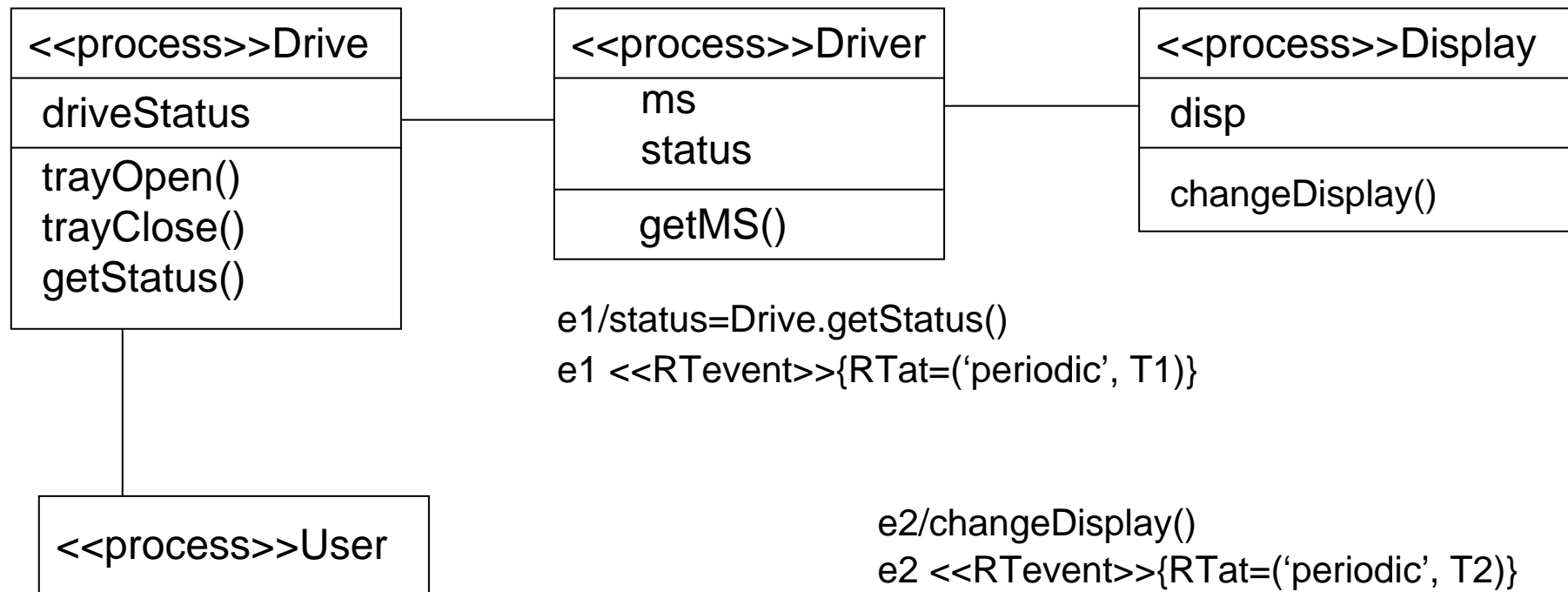
Periodic reference and asynchronous update

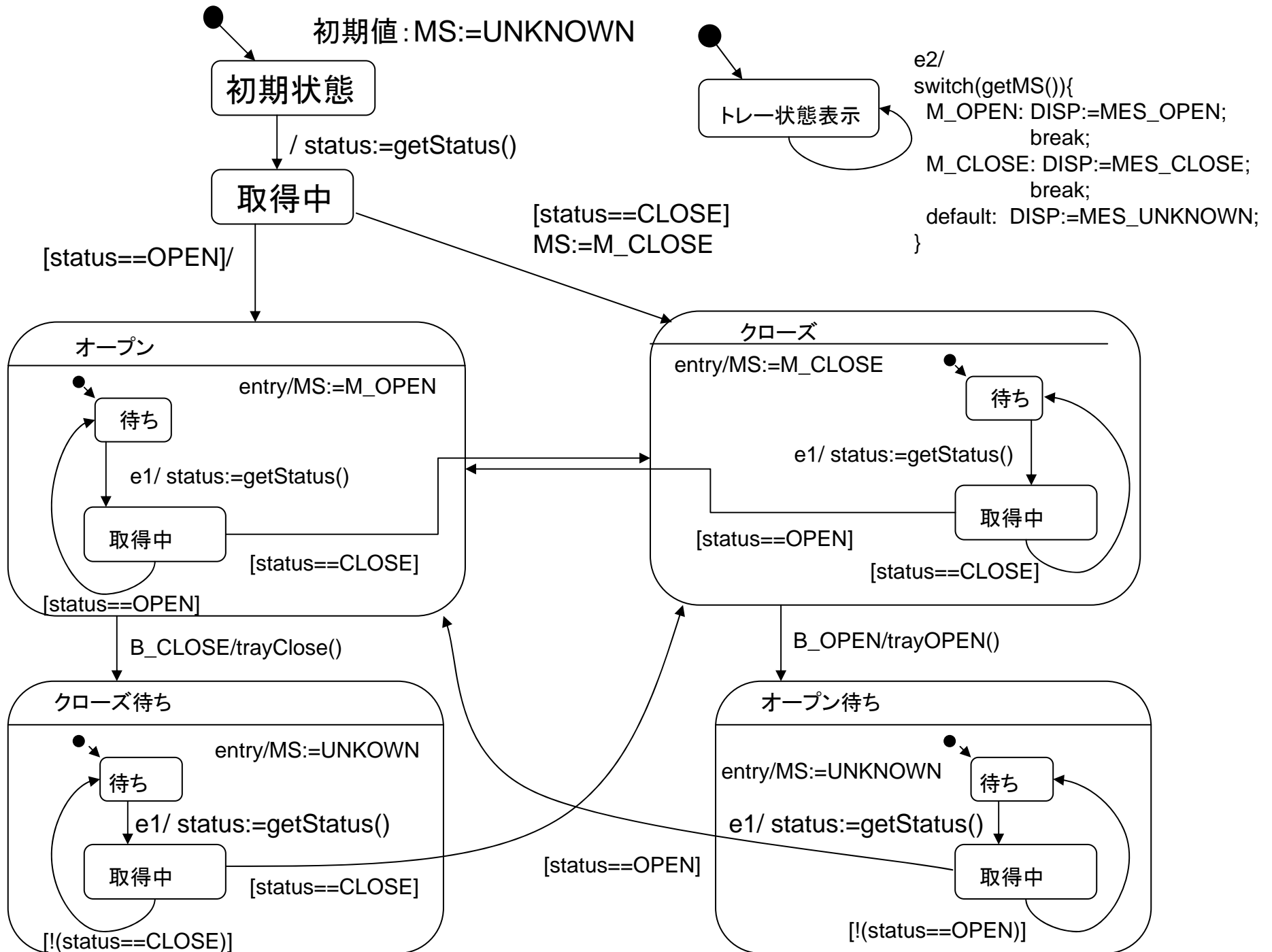
- The sequence $e_2^*e_2^*e_3$ should exist between e_1 and e_1 to prevent that the application skips to refer to the state of the drive.
- Event sequence: $(e_2^2e_3e_2^2(e_1||e_2||e_3))^+$ $\rightarrow T_1=5T_2=2T_3$

Verification of CD Player

- We construct of a design model which represents timing and periods of the CD Player.
 - Focusing on a mechanism to grasp the state of the drive.
- Verification results.
 - We detected state inconsistencies.
 - The application may skip to refer to the state of the drive.
 - The state of the application may always be different from the state of the drive from some point.

Verification of CD Player





Analysis of Periodic Execution

- The proposed approach allows us to verify timing properties based on periodic execution of tasks.
- We have applied it to the practical design model of CD player.
 - We succeeded in finding inconsistencies among the modules.
- We have to provide formal semantics with the proposed design model.
 - What do events and function calls mean in terms of time?
- We will extend the analysis method so that we can use the RTOS library.

Conclusion

- We are studying how we verify multi-task software.
 - We have proposed two methods so far.
 - Timing problems of multi-tasks executed on RTOS.
 - State inconsistency of multi-tasks based on their periodic execution.
 - There is no single solution, that is, 'silver bullet' for solving problems of multi-task software.
 - We need identify typical problems, and propose solutions for them.
- Future Works:
 - Proposing a set of verification methods for embedded software.
 - Proposing a computer environment in which the proposed methods are integrated.
 - The proposed method can be flexibly plug-ined to the computer environment.