Formal Model-Based Test for AUTOSAR multicore RTOS

Ling Fang

2012.04.19

Outline

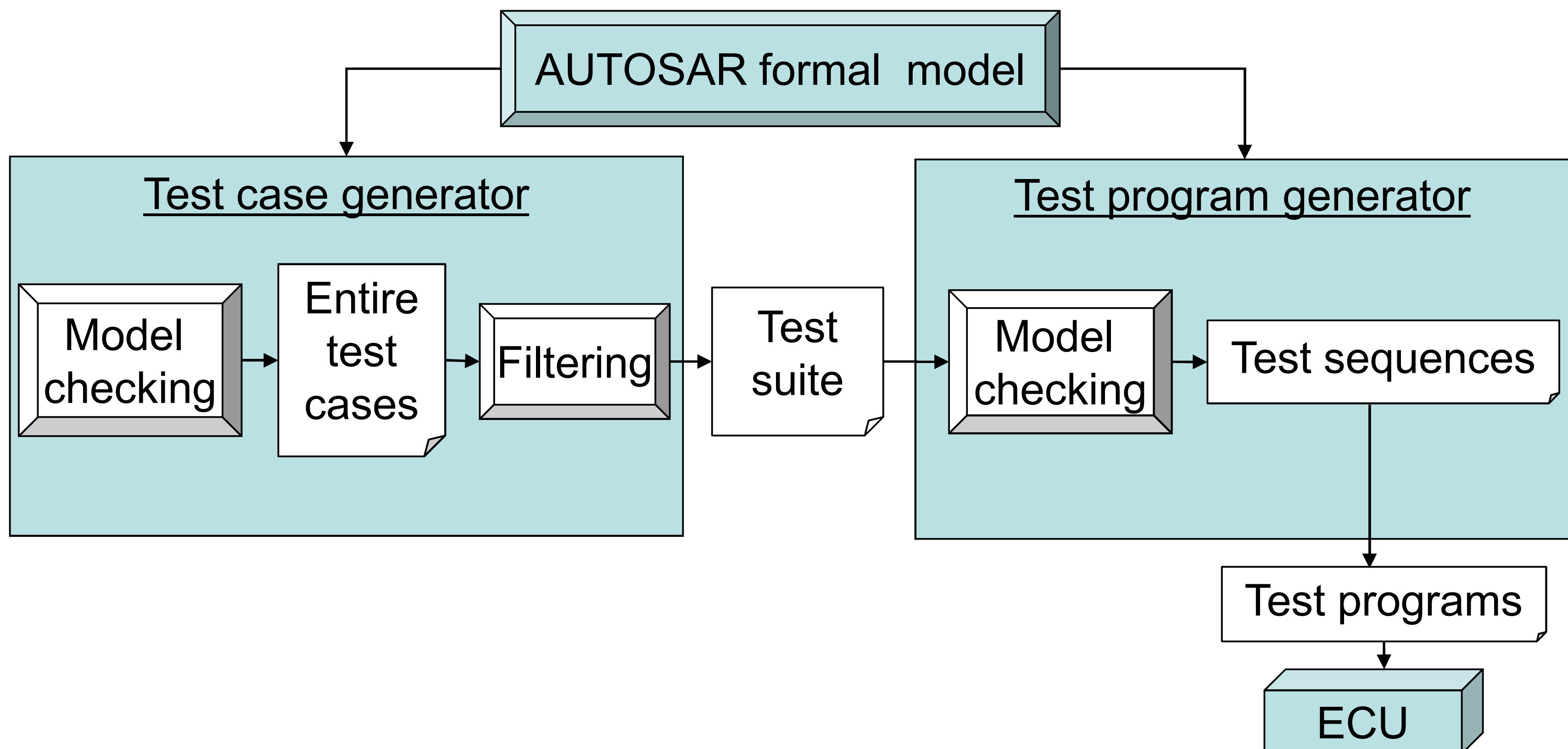
1. Abstract
   - Motivation
   - System outline
   - Related work
2. Preliminaries
   - LTL and SPIN
   - AUTOSAR RTOS
3. Formal model of AUTOSAR RTOS
   - Construct model
     * Abstract model
     * Concrete model
   - Verify the model
4. Use the formal model for test
   - Test suite generation
     * Extract a test case set
     * Test selection strategies
   - Test program generation
   - Bug analysis
5. Experiments, conclusion and future work
   - Experiment results
   - Conclusion and future work

Motivation

- AUTOSAR multicore RTOS
  - Safety-critical and high quality is required
  - Correctness is confused by concurrency
    - Multi-core and multi-task
  - Conventional test is the main method for detecting the bugs: low coverage and high cost
    - Difficult to ensure a perfect coverage
    - The manual design is an inherently time-cost process
    - Must be carried out for every system with a different configuration
    - Bug analysis usually also consumes much time
- Our proposal
  - A formal model of AUTOSAR RTOS
  - Test automation
    - A test suite generator
    - A test program generator
  - Advantage
    - It improves the coverage
    - It is completely automatic, saves cost and development time
    - Bug analysis also becomes easy

System outline

- A formal model-based test method
  - A complete test suite generator
  - Automation of the test



Related work

- Test case generation for RTOS
  - With model checking to find test cases
    - G. Hamon: iteratively extended already discovered tests and thus extended additional goals
    - Hong: enumerates the state wherein the user must provide a CTL formula for every state to be covered.
  - Use formal specification language to describe test scenario
    - Toppers project: used a formal specification for a test scenario, which needed analysis manually on about 59 thousand lines.
  - Classification Tree Model
    - Krupp A: Generate test cases with constrain conditions
- Formal RTOS development
  - FreeRTOS: developed with the B method
  - OpenComRTOS: concurrently with a formal model to certify that implementation meets its specification

LTL and SPIN

- Linear Temporal logic (LTL)
  - Kripke Structure: 4-tuple M = 〈S_0, S, R, L〉
    - S : is a set of states, S0 : initial state, R : a transition between states, L: a mapping from the states to atomic propositions, L(s): a set of atomic propositions that holds for a state s, Path: π〈s_0,s_1,...〉($\forall i \geq 0$:(s_i,s_{i+1} ∈ R)
  - LTL Syntax:

    $\pi,\psi ::= true \mid false \mid P \mid \varphi \vee \psi \mid \pi \wedge \psi \mid \neg \varphi \mid \circ \varphi \mid \Diamond \varphi \mid \Box \varphi$

    P: atomic proposition, ○ : next, ◇ : future, □ : global.
  - LTL Semantics:

    $\pi \models P$ iff $P \in L(\pi_0)$

    $\pi \models \neg \varphi$ iff not $\pi \models \varphi$

    $\pi \models \varphi \wedge \psi$ iff $\pi \models \varphi$ and $\pi \models \psi$

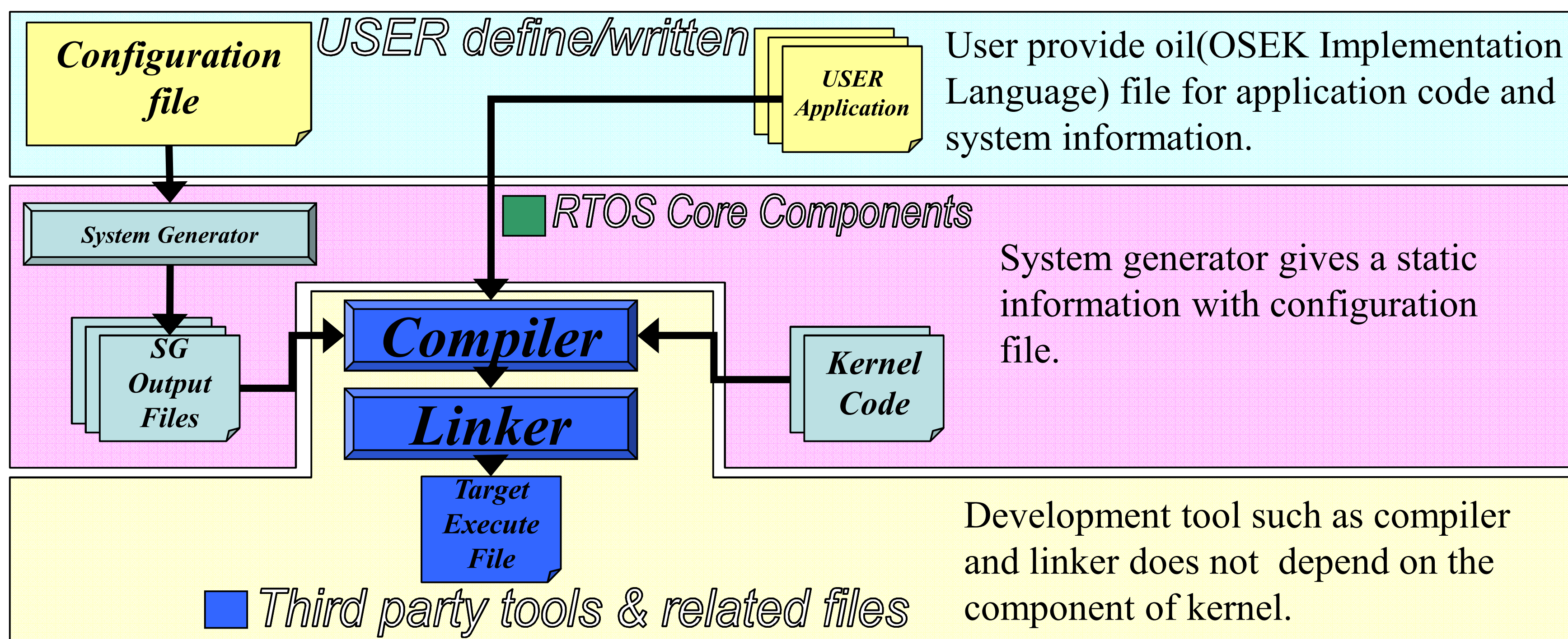
    $\pi \models \varphi \vee \psi$ iff $\pi \models \varphi$or $\pi \models \psi$

    $\pi \models \circ \varphi$ iff $\pi_1 \models \varphi$

    $\pi \models \Diamond \varphi$ iff $\pi_i \models \varphi$ for some $i \geq 0$

    $\pi \models \Box \varphi$iff $\pi_i \models \varphi$ for any $i \geq 0$
- SPIN model checker
  - SPIN is the most mature tool for LTL

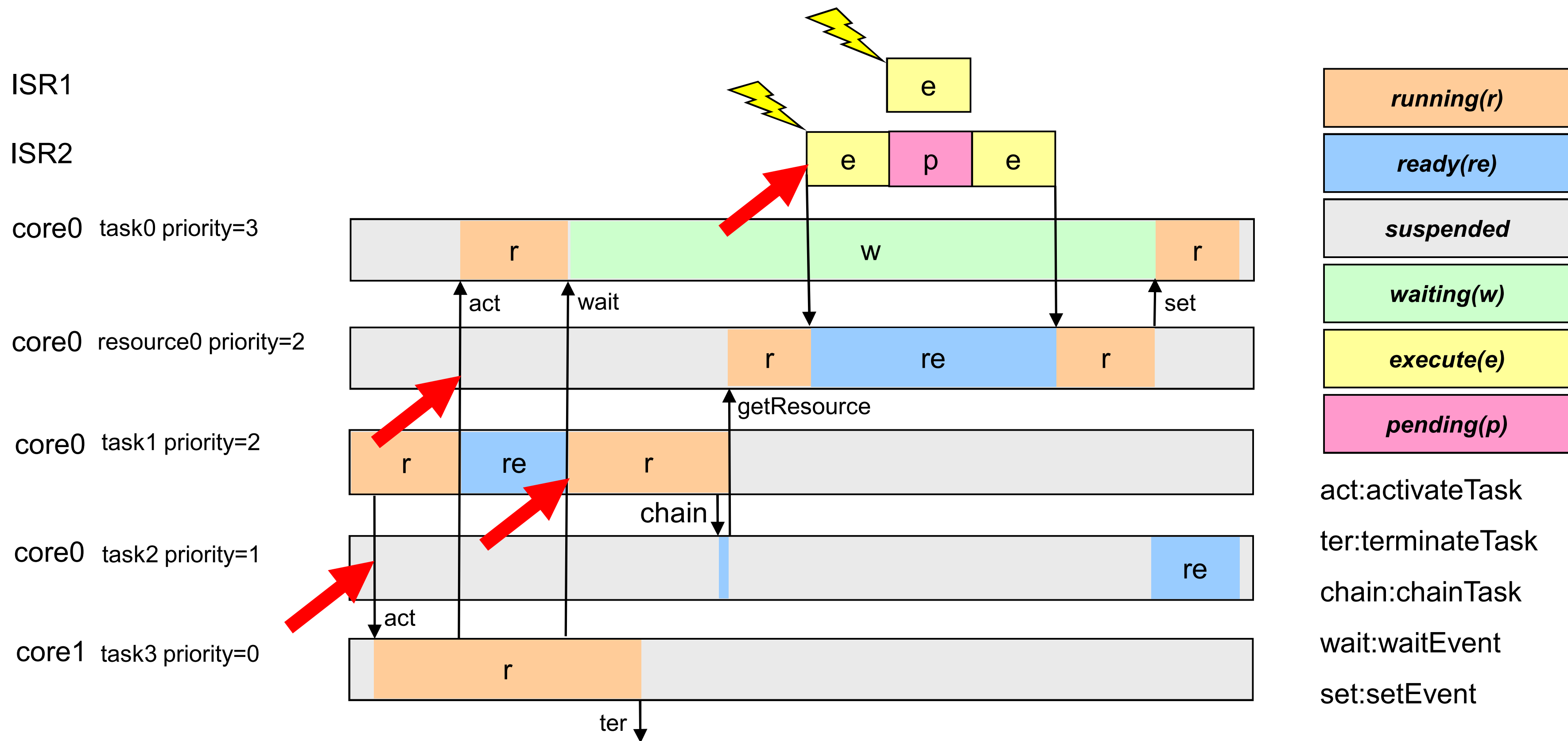
AUTOSAR RTOS: Configurable RTOS



Concepts of RTOS

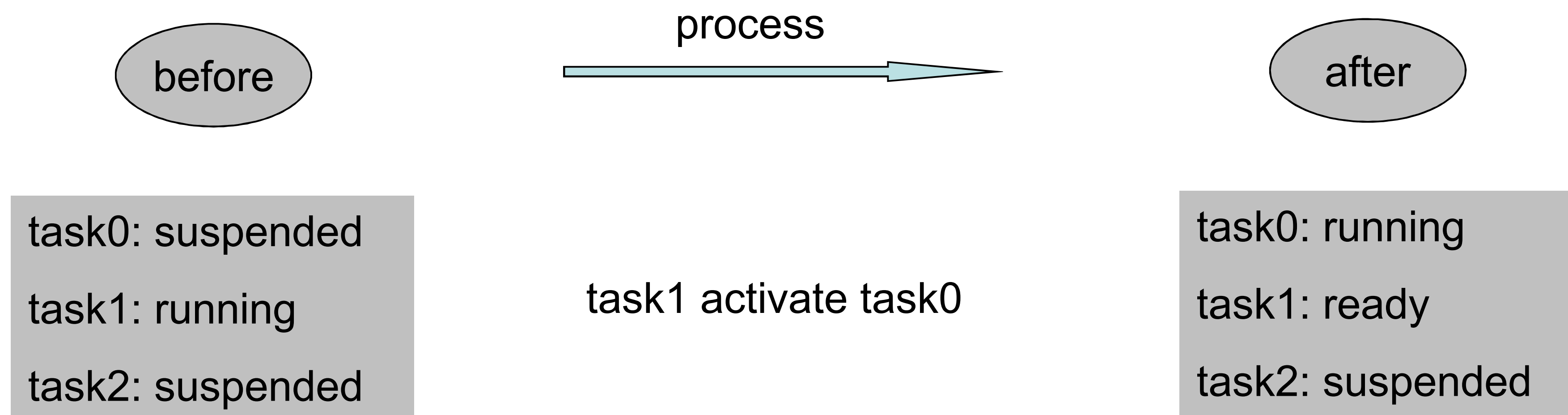
- Task and SVC:
  - Task: Basic unit of programming controlled by an RTOS, refers to an execution context of RTOS management or applications
  - SVC: Request service to the RTOS, activateTask, TerminateTask, ChainTask,…
- Interrupt Service Routine (ISR):
  - ISR category 1 (ISR1) has no effect on the OS
  - ISR category 2 (ISR2) calls API functions and rescheduling is initiated by the generated ISR2 epilogue
- Event:
  - Provide a method for the synchronization of different tasks, or ISR2
- Resource:
  - Used to ensure the shared access
- Critical section and Lock:
  - Critical section means the way of arranging things so that a modification of shared data structure appears atomic
  - Lock is a binary semaphore used to implement critical section

Typical behavior of RTOS

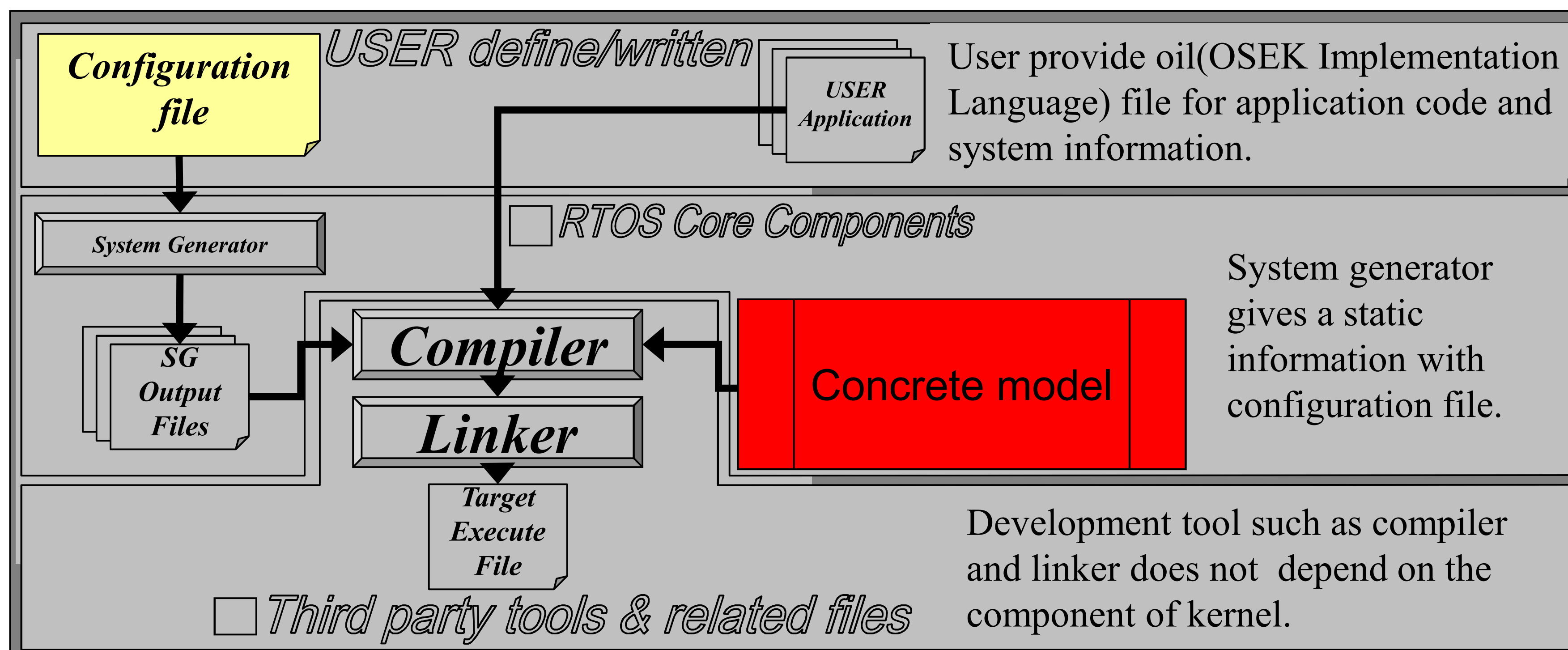


API test case

- API test
  - The system states before and after API process

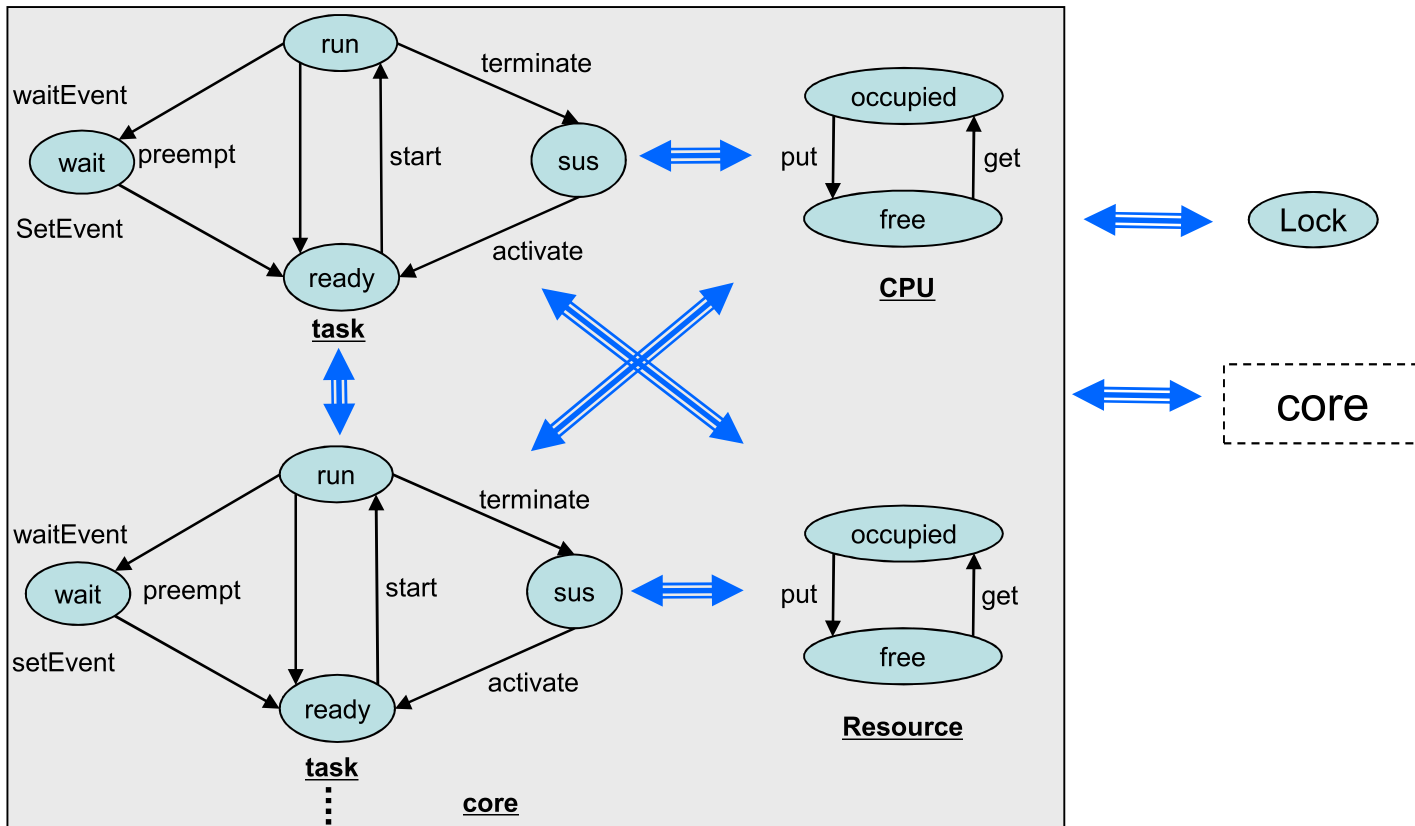


Construct a configurable RTOS model



Abstract model of AUTOSAR RTOS

- Definition is performed in two steps:
  - Define states and transition condition labels for each transition systems
  - Define the interaction communication between the transition systems
    - Asynchronous communication
    - Synchronous communication



Concrete model

- The concrete model must be created with the system configuration. Target model is concrete containing the information of numbers, identities, priorities, accessibilities of tasks, resources, ISR2, events, and etc.
  - Abstract model can be reused for any RTOS conforming AUTOSAR requirement
  - Test cases from target concrete model correspond the concrete system, so they are executable

```
task[0].priority = 3
task[0].coreID = 0
task[0].preemptive = preemptive
task[0].status = suspended
…
```

Configuration file

Verify the model

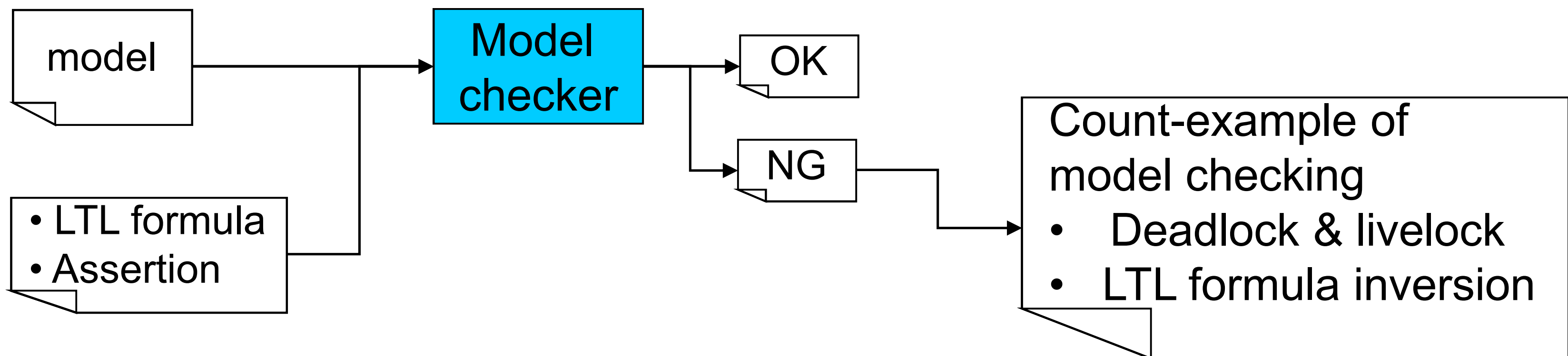
- Deadlock and livelock:
  - Be detected with a model checker automatically
- LTL formula:
  - Reachability
  - Safety

$\neg(task_i.state=ready \wedge task_j.state=running) \wedge (task_i.priority>task_j.priority)$

- Assertion:

$$\neg(\sum_{i=0}^{n} task_i.state = ready > 0 \wedge CPU = free)$$

$$\neg(\sum_{i=0}^{n} task_i.state = running > 1)$$



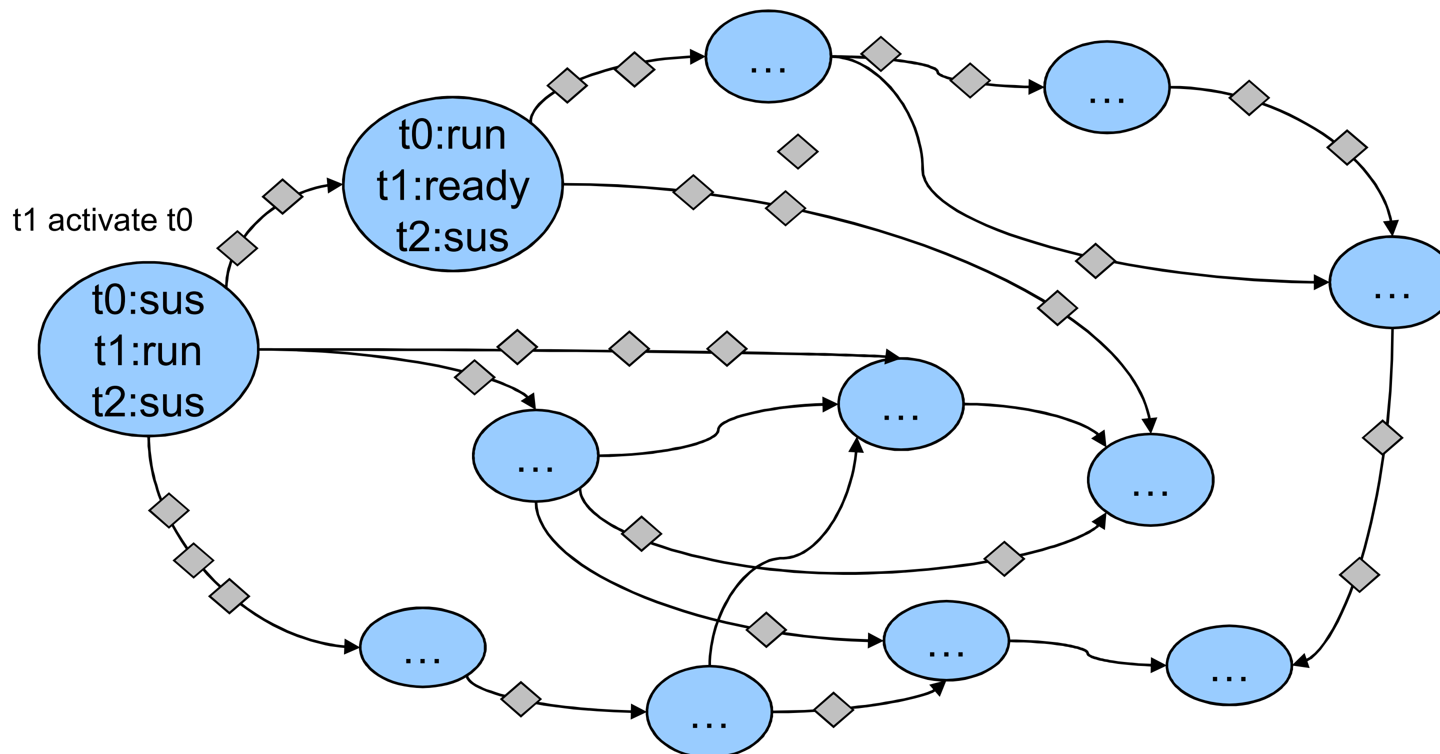
Use the formal model for test

- With the test case generator
  - Extract the heuristic test case set from the formal model
  - Filter the feasible and entire test suite with coverage criteria
- With the test program generator
  - Calculate the test sequences with model checking for the test cases
  - Translate the test sequences into the executable test programs

Generate heuristic test case set

Circle: end point of critical section, a complete transition for SVC (must be atomic)

Diamond: state inside critical section



Generate heuristic test case set

- A heuristic and finite test case set can be extracted from the formal model
  - Counterexample: ***$task_0$=claim*** (claim is not a normal state of task)
  - At the end of critical section

$$T_0\ activate\ T_1\ :$$
$$T_0.run, T_1.sus, T_2.sus \rightarrow T_0.run, T_1.ready, T_2.sus$$

Test selection strategies

The classification tree is used to find all the possible combinations of the conditions from the heuristic test case set, in other words, to group the heuristic set with the criteria and select one case from each group as the member of the target test suite. For example:

C1 : executor and affected task are on the same core.

C2 : executor and affected task are on the different cores.

C3 : executor is activated for several times.

C4 : executor is activated for single time.

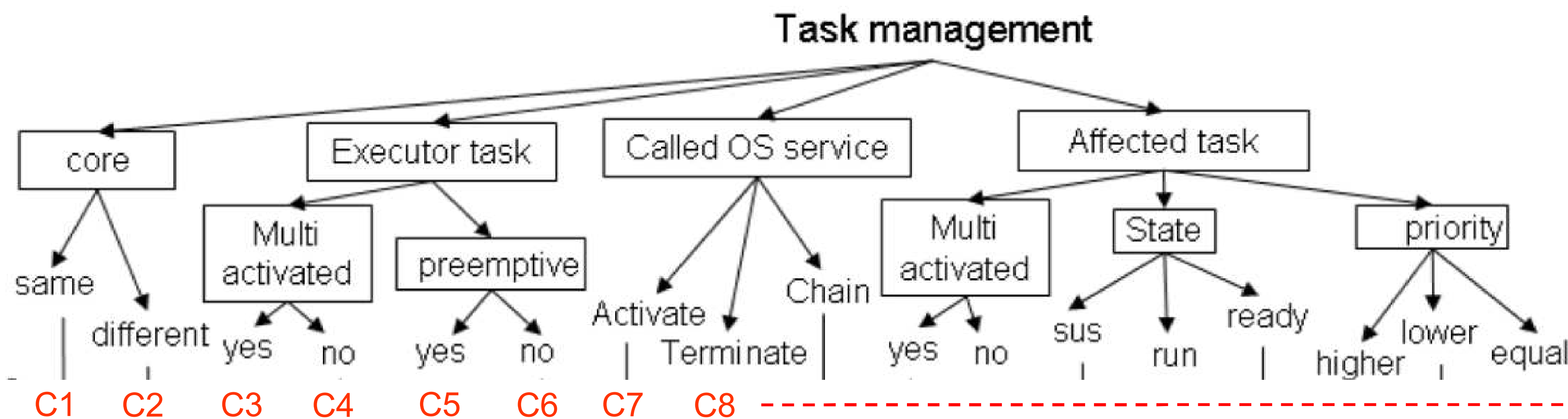
…

Case4: T0 activate T1 : T0.run, T1.sus → T0.run, T1.ready

Case5: T0 activate T2 : T0.run, T2.sus → T0.run, T2.ready

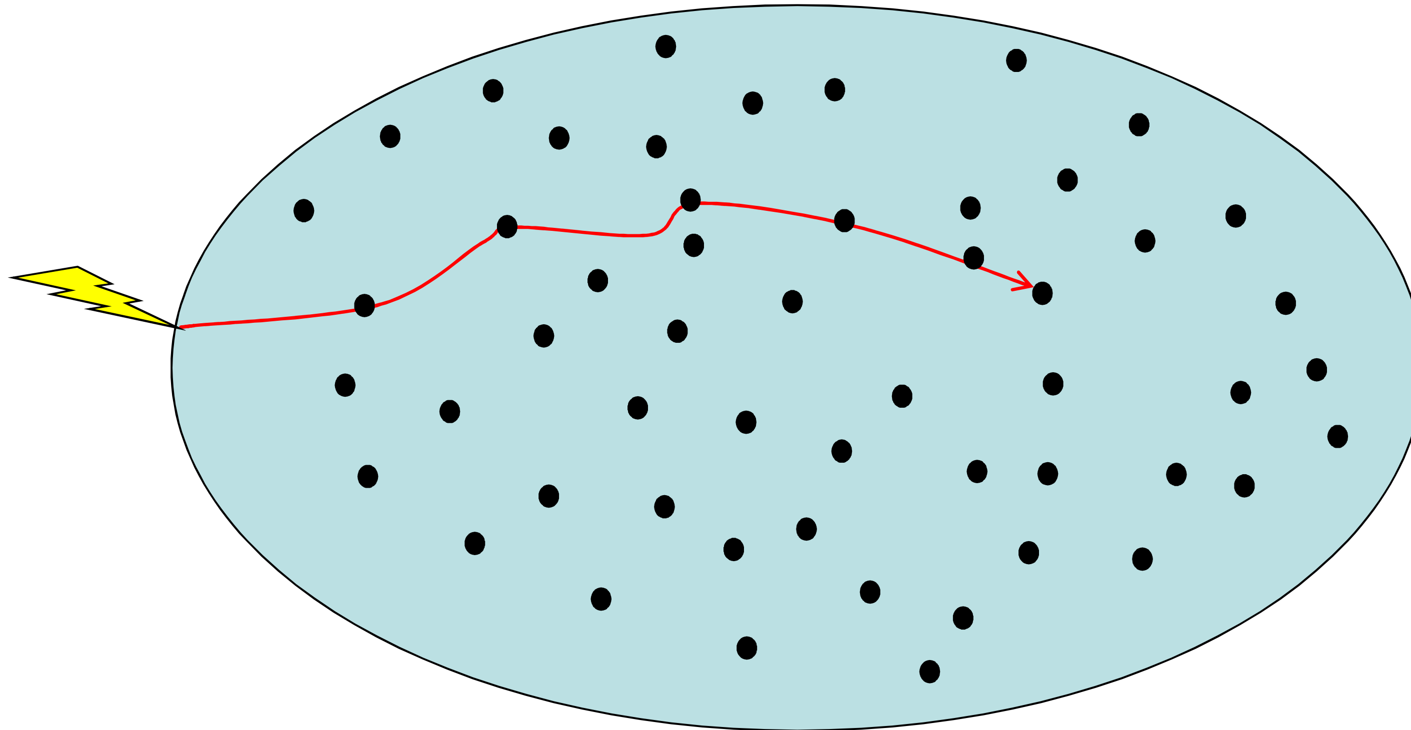
Case6: T0 activate T1 : T0.run, T1.sus, T2.sus → T0.run, T1.ready, T2.sus

Case7: T0 activate T1 : T0.run, T1.sus, T2.ready → T0.run, T1.ready, T2.ready



Test program generation

- Find a sequence of operations to set up the internal state before the test case can be applied
  - Test case is used as counterexample
  - Model checking for finding appropriate paths (test sequences)



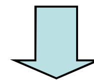
The complete and useful test suite

The entire test case set

Test program generation

- Translate the path from model checker to the program which can be executed on ECU

$$T_0 \ activate \ T_1 : T_0.run, T_1.sus \rightarrow T_0.run, T_1.ready$$
$$T_0 \ chain \ T_1 : T_0.run, T_1.ready \rightarrow T_0.sus, T_1.run$$
$$T_1 \ chain \ T_0 : T_0.sus, T_1.run \rightarrow T_0.run, T_1.ready$$



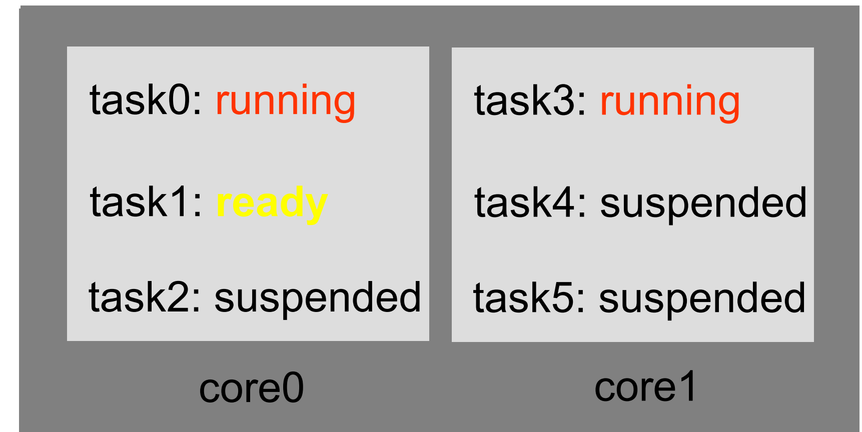
```
void TASK_OsTask0 (void)
    switch(TestIndex)
        case 1 :
            Activate(task1);
            Check(task0,task1);
            TestIndex++;
            break;
        case 2 :
            Chain(task1);
            Check(task0,task1);
            TestIndex++;
            break;
```

```
void TASK_OsTask1 (void)
    switch(TestIndex)
        case 3 :
            Chain(task0);
            Check(task0,task1);
            TestIndex++;
            break;
```

Bug analysis

- When bugs exist, the user can analyze them with a trace along the test sequence step-by-step

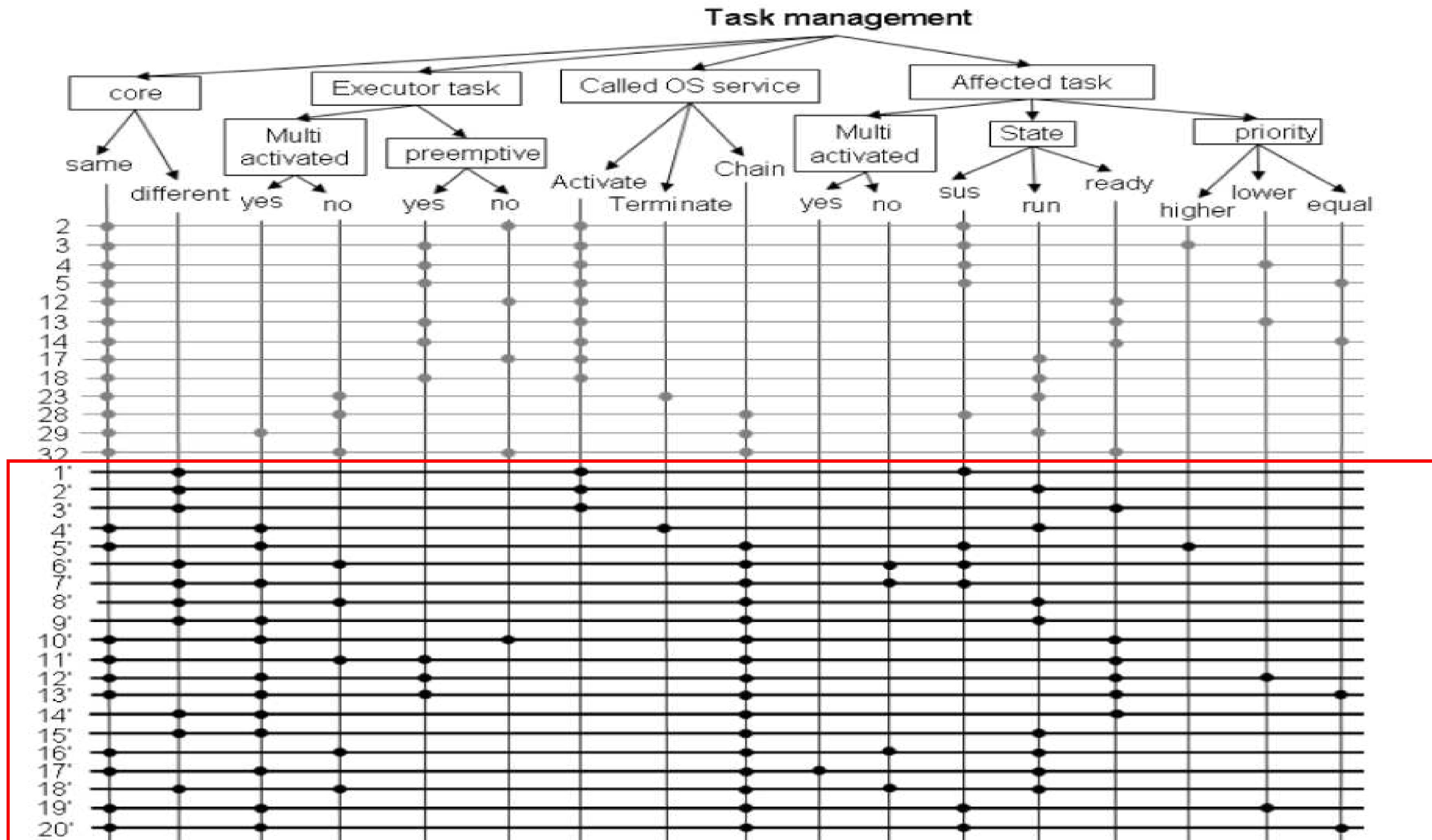
Action: task1 ActivateTask task_3
task status: 0:sus 1:run 2:sus 3:sus 4:sus 5:sus
Action:task1 TerminateTask
task status: 0:run 1:ready 2:sus 3:run 4:sus 5:sus
Difference: task1 is ready in model, but is sus in ECU
Action: task3 ActivateTask task0
task status: 0:sus 1:run 2:sus 3:run 4:sus 5:sus



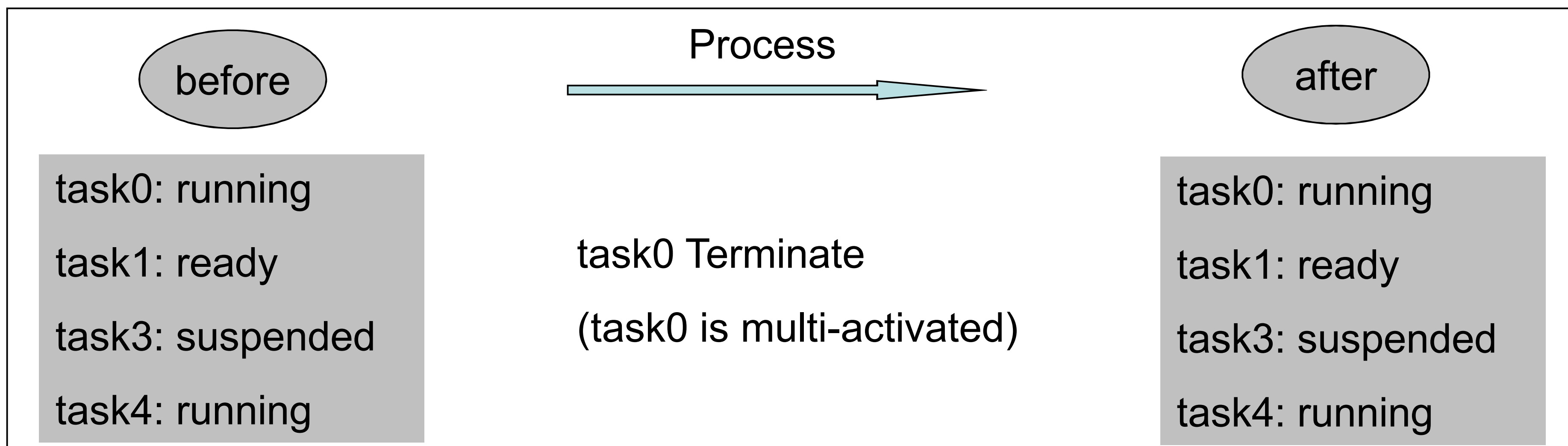
TerminateTask task1 ??

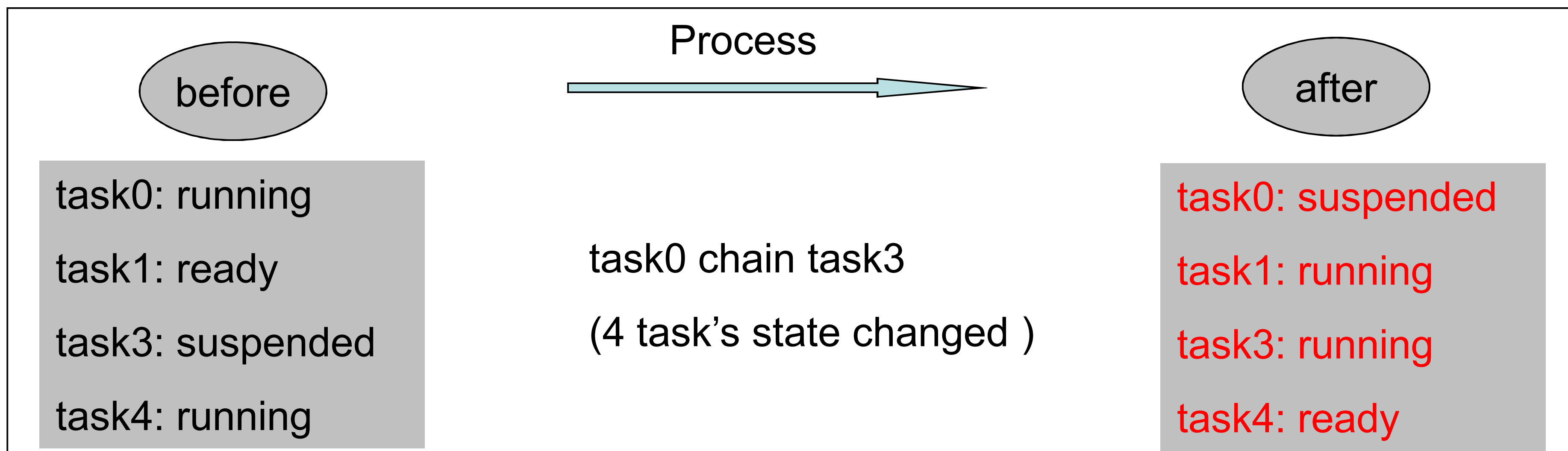
Experiment results

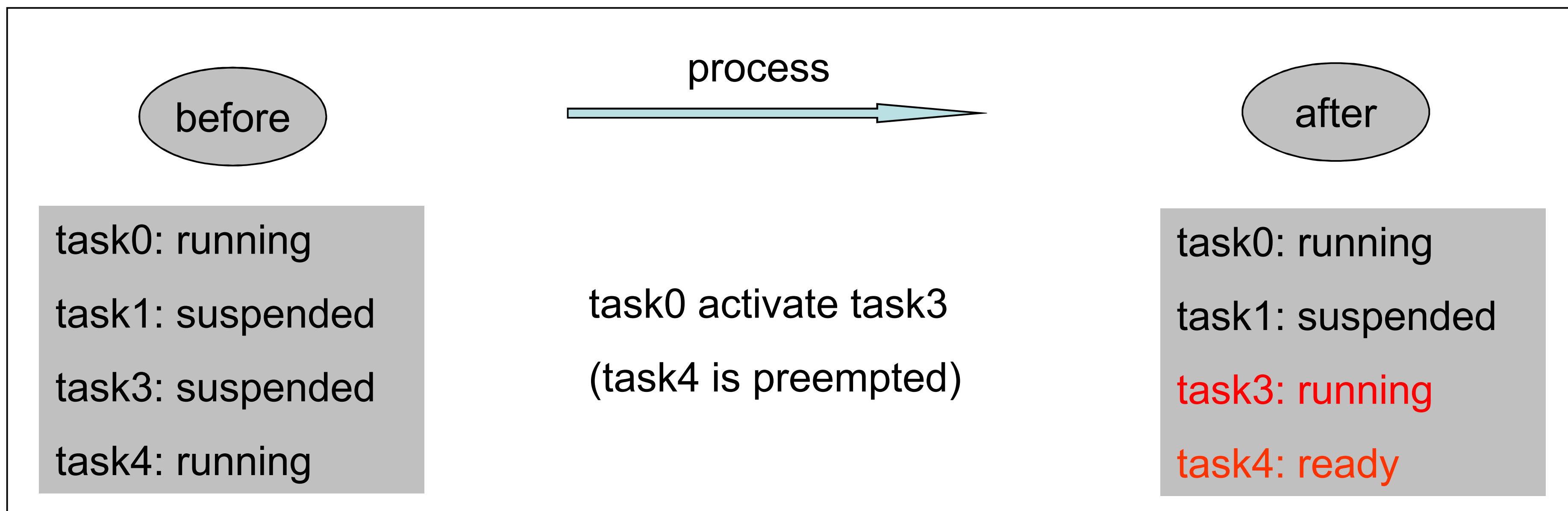
- 20 new cases were found
  - Almost all cases when the executor and affected tasks were on different cores
  - Several complicate cases when the executor and affected tasks were on the same core

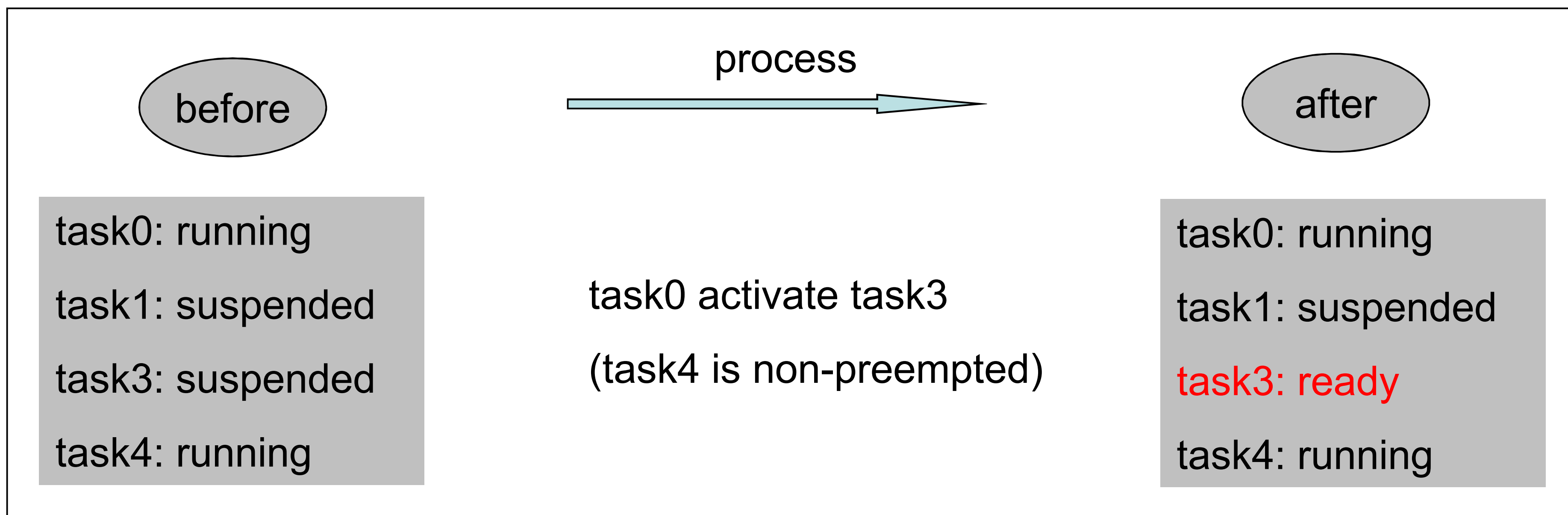


Typical new test cases






before
process
after
task0: running
task1: suspended
task3: suspended
task4: running
task0 activate task3
(task4 is preempted)
task0: running
task1: suspended
task3: running
task4: ready


before
process
after
task0: running
task1: suspended
task3: suspended
task4: running
task0 activate task3
(task4 is non-preempted)
task0: running
task1: suspended
task3: ready
task4: running

Contribution and future work

- Contribution
  - Greatly improve the quality
    - It extracts test cases from a model more completely
    - It is based on the behavior of the system, it does not need to consider complex test scenarios
  - Greatly cut the cost
    - It includes automatic test case generation and test execution completely automatically
    - Easy to be reused
    - Easy to be changed
    - Easy for bug analysis
- Future work
  - Complete the model for timing features
  - A smart performance test is desired by industry

END

Please contact me if you have any questions.

ling.fang@siat.ac.cn (I changed my work from Japan to China)