

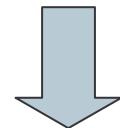
A Verification Framework for Automotive Embedded Systems

Center for Verification and Semantics,
National Institute of Advanced Industrial
Science and Technology (AIST/CVS)

Norio KATO

Background

- Automotive embedded systems employ incremental/variant development.
 - Bugs introduced at **join time** are pervasive.
 - A framework that facilitates to detect such bugs is asked for.
 - Requirements have to be managed consistently.
 - Need to **specify** them to find out bugs.
 - Requirements may change along development.
 - Both old specs and new specs need to hold.



- We propose a framework of development that facilitates both **verification** and **spec management**.

Outline of the Talk

- Background
- Considerations
- Verification Flow
- Example
- Conclusions and Future Directions

Proposed Framework

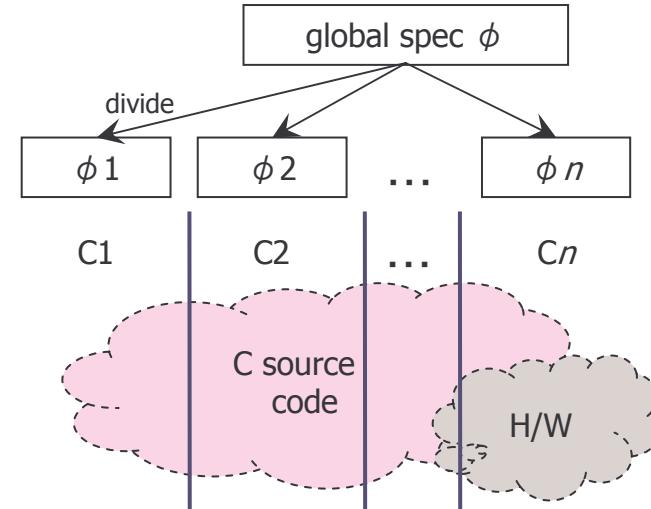
- We propose a framework of development that facilitates both **verification** and **spec management**.
 - designed for use in incremental/variant development
- Considerations
 1. Use of Model Checking
 - suitable for join-time (design) verification
 2. Local Management of Specs
 - to make specs explicit
 3. Handling Model Preciseness
 - to lower verification costs

1. Use of Model Checking

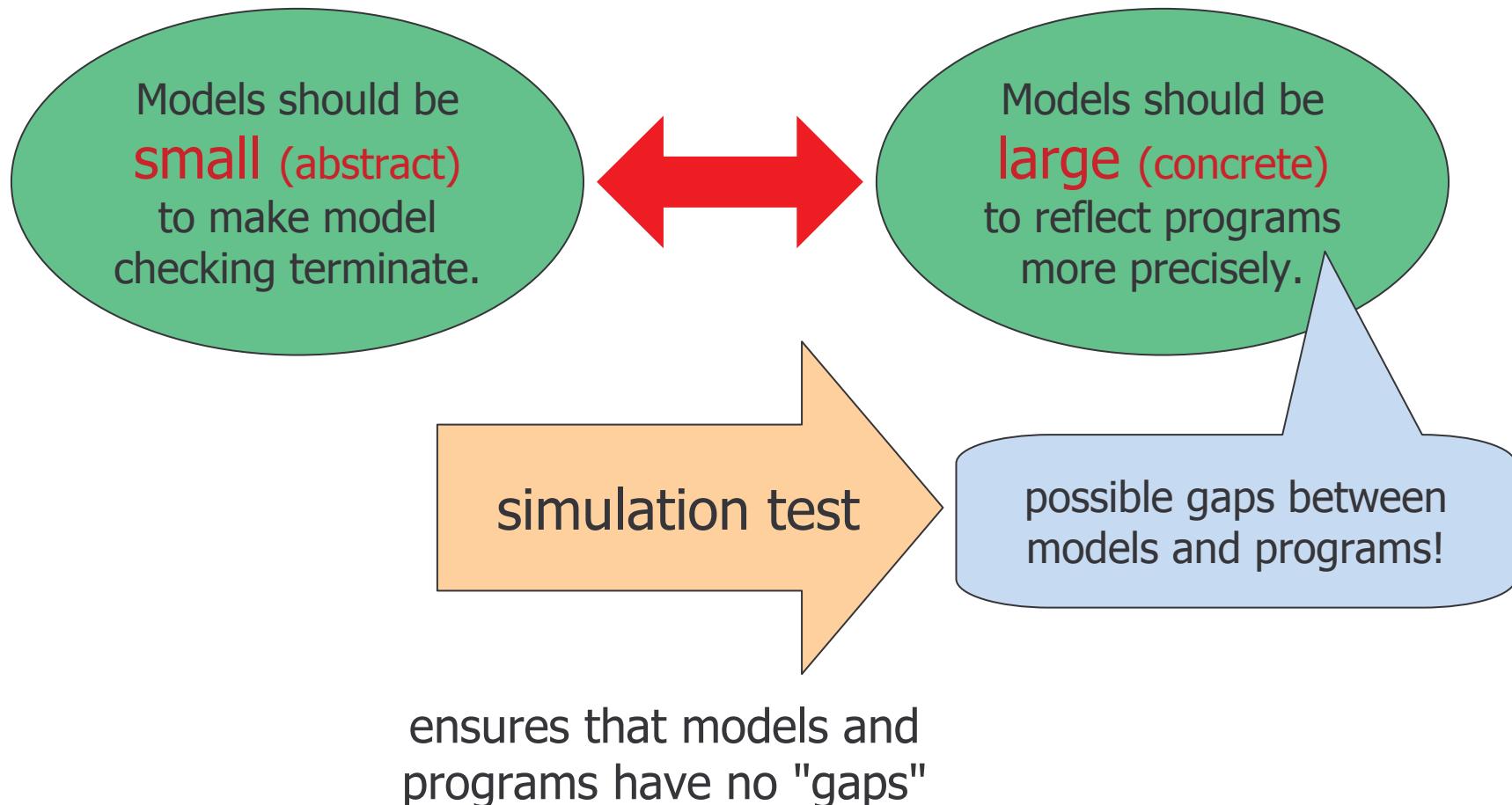
- Advantages
 - exhaustiveness
 - suitable for verifying concurrency (e.g. deadlock)
 - no test case provision is needed
 - tool-supported model composition by interleaving
- Offers a method of **join-time verification**.
 - crucial for variant development
 - otherwise difficult to detect and specify bugs
- Also provides a certain amount of guarantee that the specification holds for implementation.
 - provided that the model is correct!

2. Local Management of Specs

- Divide specification ϕ of the entire system into several specs ϕ_i which are local to components C_i respectively
 - such that $\phi_1 \wedge \dots \wedge \phi_n \rightarrow \phi$ holds
- Effectively manages what must be done in implementing/modifying each part.
 - (Maybe also reduces the total amount of verification time.)

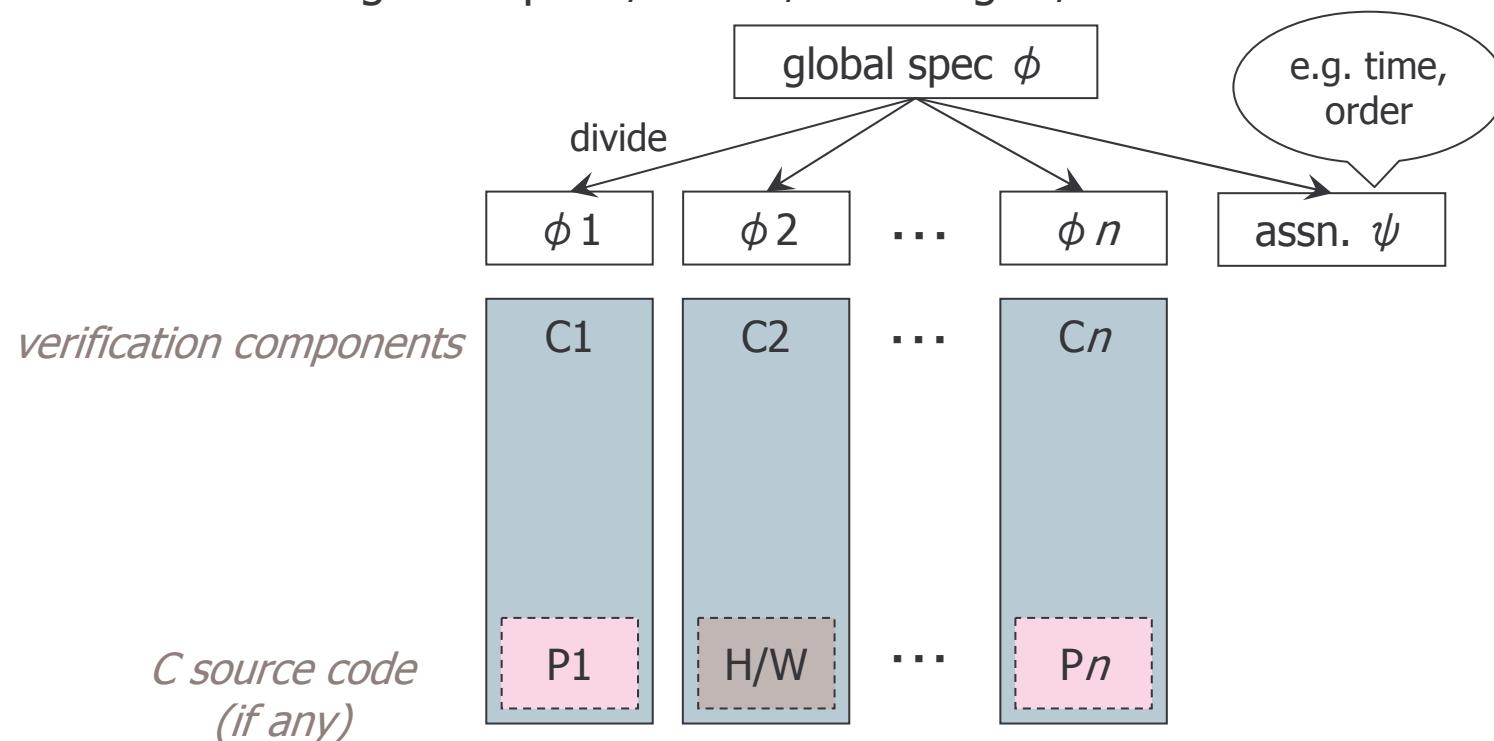


3. Handling Model Preciseness



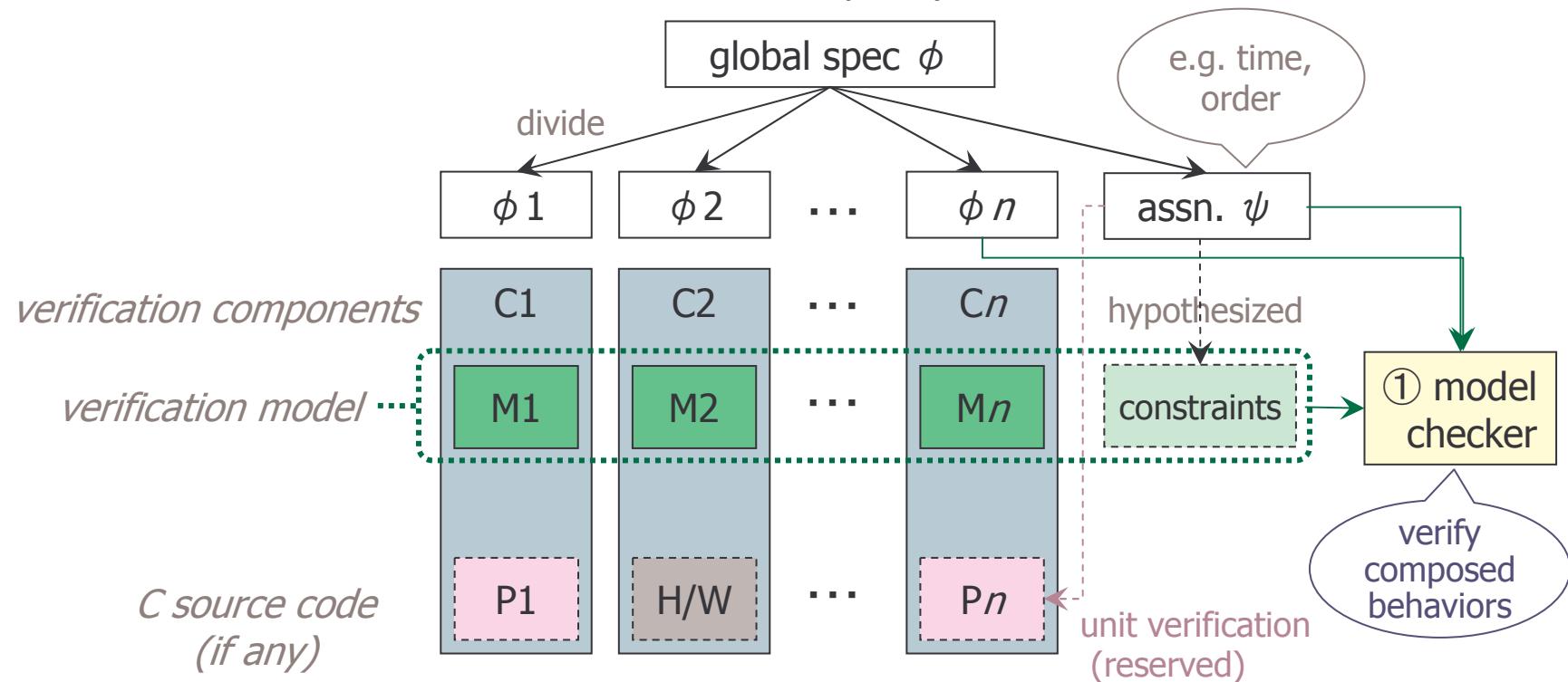
Proposed Verification Flow (1/3)

1. Determine a global spec to verify, say ϕ .
2. (In design time) divide the system into concurrent composition of several verification components C_1 through C_n .
3. Divide the global spec ϕ into ϕ_1 through ϕ_n which are local to C_i 's.



Proposed Verification Flow (2/3)

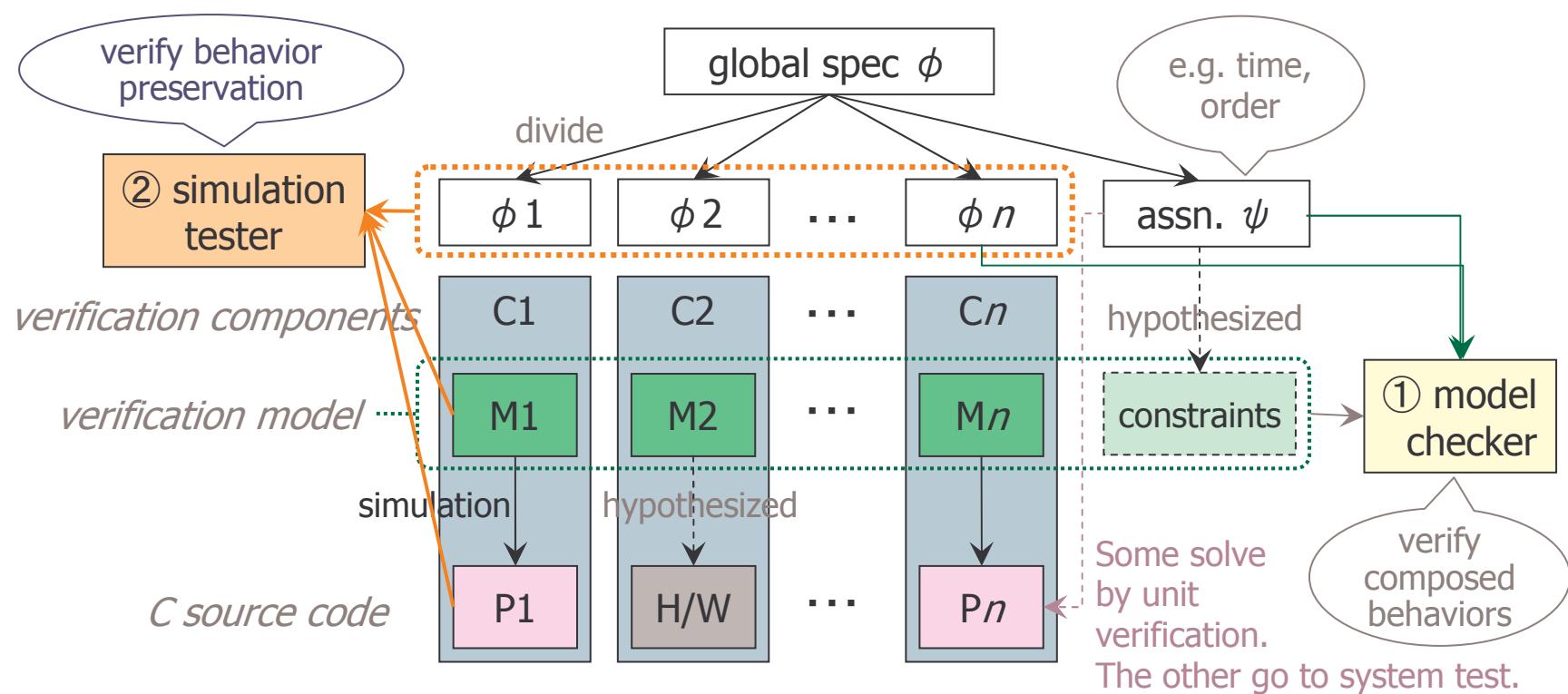
4. Describe the design of each component C_i as a model M_i .
5. Model check specs ϕ_1 through ϕ_n towards the composed model.
 - Amounts to join-time verification.
 - Inside details can be assumed and explicitly reserved for unit verification.



Proposed Verification Flow (3/3)

6. Verify that the model simulates the program.

- Redo this step every time after the implementation is changed.
- Among assumptions, those which are hard to verify (such as realtime constraints) should go to system test.

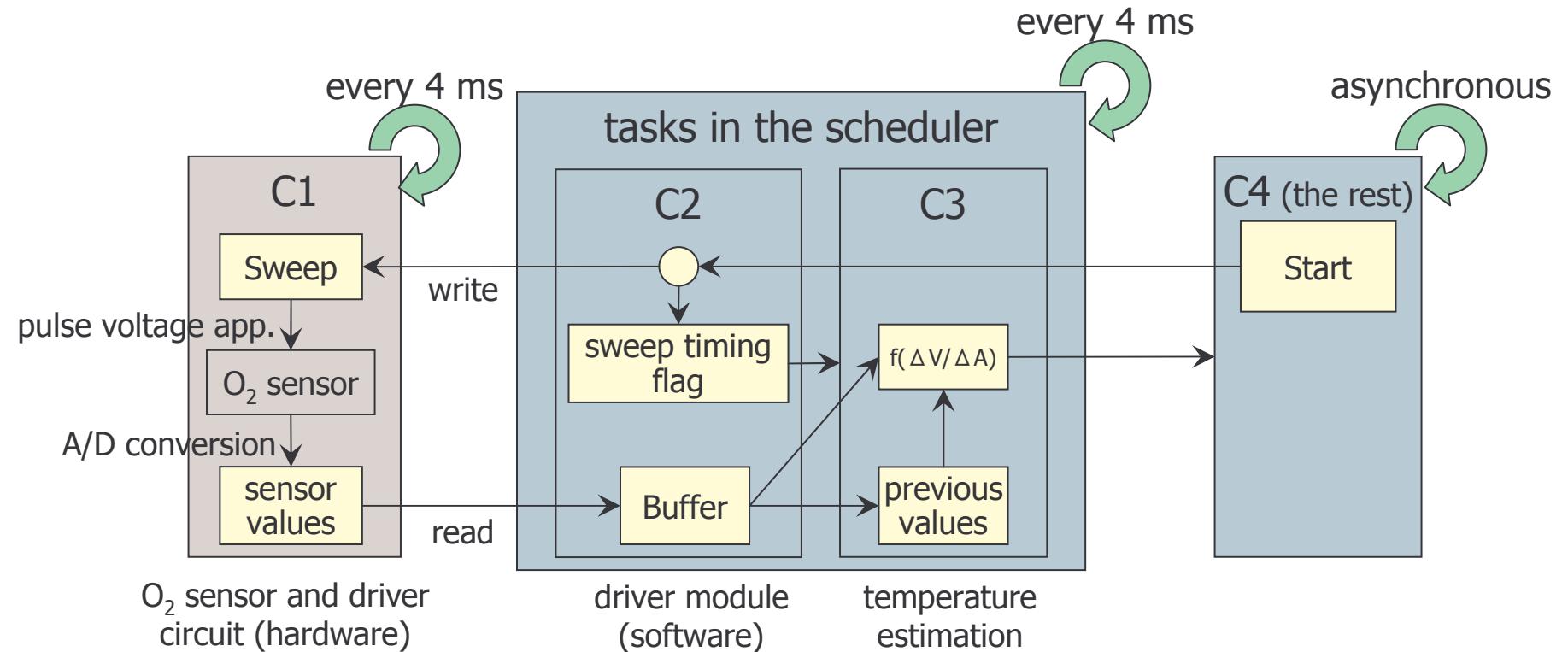


Verification Flow (summary)

- Within a single development scene:
 - verification of design and implementation
 - ① Verify that the design enjoys the given spec.
 - by model checking
 - ② Verify that the design corresponds to the implementation.
 - by simulation testing
 - These ensure that the implementation enjoys the given spec.
 - Within a development cycle:
 - verification of incremental/variant development
 - Detect "gaps" which are common to get introduced between design and implementation.
 - by simulation testing

Example Run of the Verification Flow

- Global spec = "the correct temperature is estimated"
- Objectives: to see whether the following are achieved:
 - clarification of the assumed functionality (or spec) of each part
 - join-time verification by model checking the composed behaviors



Detailed Explanation of the System

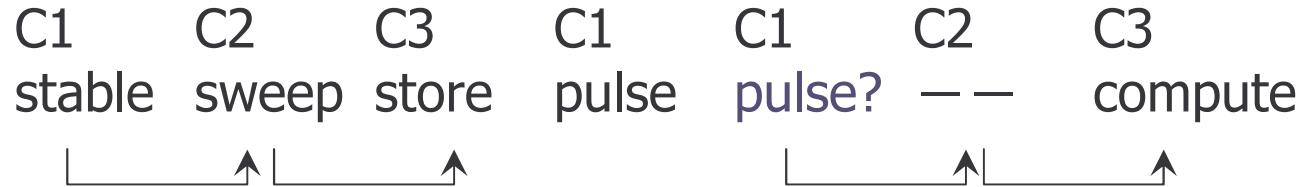
- The components of the system:
 - C1: O₂ sensor and driver circuit (hardware)
 - Updates the sensor values (voltage and current) every 4 ms.
 - Right before the update, applies a pulse voltage if Sweep is set.
 - Temperatures can be estimated from the sensor value changes.
 - C2: driver module (software)
 - Sets Sweep to true when Start is turned on.
 - Writes the current sensor values to Buffer.
 - C3: temperature estimation (software)
 - Reads from Buffer to compute the temperature.
 - C4: the rest
 - Sets Start to true asynchronously.
- Assumption
 - C2 and C3 are sequentially scheduled every 4 ms.

Specification Dividing and its Effects

- Global spec ϕ : "the correct temperature is estimated"
 - ϕ 1: "apply a pulse voltage if Sweep is set"
 - ϕ 2: "transfer to Buffer sensor values before/during a sweep"
 - ϕ 3: "read the values to compute at **appropriate timings**"
 - ϕ 4: no conditions
 - assn. ψ : certain time constraints (e.g. Start only if stable)
- Clarification of specs (esp. on variables) is enforced.
 - 1) Which timing is **appropriate**?
 - "the point which the sweep timing flag is turned off"
 - 2) When the estimated temperature is ready to be read?
 - 3) When and who cancels the Sweep flag?

Model Checking with SPIN

- Aimed at join-time verification
 - Details inside a component are simply assumed.
 - They are to be verified at unit level.
 - e.g. "correct values are propagated and computed"
- The global spec ϕ
 - $\square(\text{temperatureDone} \rightarrow \text{temperatureOk})$
- Counterexamples exist if Sweep may cancel too early:



- Adding the following assumption makes ϕ hold:
 - "C1 and {C2,C3} run alternately"
- Alternatively, the design may be modified to delay cancels.

Some Model Fragments (1/2)

- Some time constraints are described not as logical formulas but as a process.

```
/* management of time constraints */
active proctype timer() {
    c1Ready = true;
    !c1Ready ->
    do :: true ->
        c1Ready = true;
        if :: true -> skip
            :: true -> c2Ready = true;
            !c2Ready ->
            c3Ready = true;
            !c3Ready ->
        fi;
        !c1Ready ->
        startOk = bufferBef && prevBef;
    od
}
```

```
#define c2start c2_loop@c2_start
active proctype c2_loop() {
    do :: c2Ready ->
        c2_start:
        ...
        c2Ready = false;
    od
}
active proctype c4_loop() {
    do :: true ->
        if :: !start && startOk ->
            start = true;
        :: ...
    fi
    od
}
```

```
ASSN="[]<>c2start && [](sweep -> (!c2start U sensorDur)) "
SPEC="[](temperatureDone -> temperatureOk)"
```

Some Model Fragments (2/2)

- Sensor values are abstracted with respect to whether they came to exist before/during a sweep.

```
active proctype c2_loop() {
    do :: c2Ready ->
        c2_start:
            atomic {
                bufferVTemp = sensorV;
                bufferATemp = sensorA;
                bufferBefTemp = sensorBef;
                bufferDurTemp = sensorDur;
            }
            bufferBef = false;
            bufferDur = false;
            bufferV = bufferVTemp;
            bufferA = bufferATemp;
            bufferBef = bufferBefTemp;
            bufferDur = bufferDurTemp;
            ...
            c2Ready = false;
    od
}
```

```
active proctype c1_loop() {
    do :: c1Ready ->
        ...
        atomic {
            sensorV = adconv(analogV);
            sensorA = adconv(analogA);
            sensorBef = !sweep;
            sensorDur = sweep;
        }
        ...
        c1Ready = false;
    od
}
```

Conclusions

- Our framework for development can facilitate both **verification** and **spec management**.
 - Amenable to incremental/variant development.
 - because the system is modeled and managed as concurrent composition
 - Enforces clarification of specs.
 - Because specs have been written explicitly, it is clear what to do in implementing/modifying programs.
 - (This comes from the use of formal methods.)
 - Specs hold also for the implementation.
 - once simulation checking tools are available

Future Directions

- Investigate practical issues by experiments.
 - what to do when the checker fails
 - what to do when models/programs have changed
 - offer the verification flow as a guideline
- Develop simulation checking tools.
 - automated tools for a restricted class of programs
 - semi-automated tools for more general programs
 - possibly connected to interactive proof assistants