Dependability and Verification

Systems Software Verification Team

Hajime Fujita, Toshiyuki Maeda,
Motohiko Matsuda, Shuichi Oikawa

Entire goal of DEOS project

- Design and implement
  systems software mechanisms to achieve
  Open Systems Dependability

Goal of our team

- Support dependability of
  the mechanisms
  for Open Systems Dependability
  - If they contain bugs, everything will be lost
    - "Introducing new code" = "introducing new bugs"

Our approach

- Ensure safety of
  the mechanisms for Open Systems Dependability
  by using *program verification* technologies

What is program verification?

- Prove that programs have certain properties by analyzing the programs

- E.g., Java and JVM bytecode type checks
  - They prove memory safety of programs

Advantage of program verification

- We can detect
  problems that may occur at runtime
  without executing them
  - by verifying properties that represent the problems

  - For example, we can prevent the following problems
    - Illegal memory access, buffer overflow,
      malfunction with unexpected inputs,
      virus intrusion, API misuse, etc.

Limitation of program verification

- "Unknown" problems cannot be verified
  - Program verification verifies known properties
  - Remember that Open Systems Dependability is an ability to continuously manage unpredictable failures (Open Systems Failures)
  - What can program verification do for Open Systems Dependability?

Our approach to supporting Open Systems Dependability

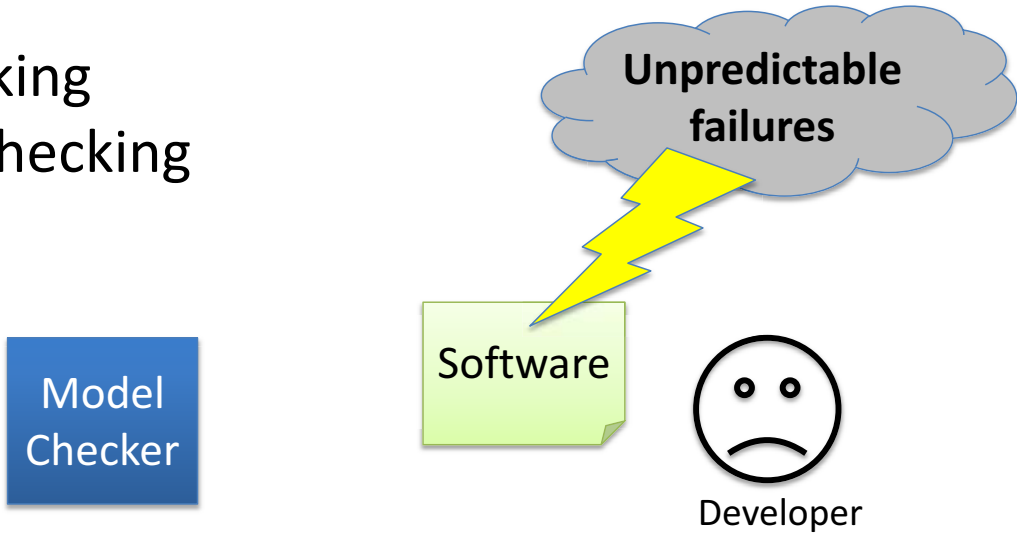
- Utilizing two verification approaches in a complementary way
  - Type checking & model checking
    - Boost up stable and continuous modification of programs in response to Open Systems Failures

Comparison of 2 verification tools

	Type checker	Model Checker
Target safety property	Basic safety (e.g., memory safety, etc.)	Advanced safety (e.g., consistency of locks, correct API usage, etc.)
Target program	C source code Binary executable	C source code
Spec. description	(almost) Unnecessary	Necessary (Describing properties to be verified as specification, etc.)
Verification time	short	long

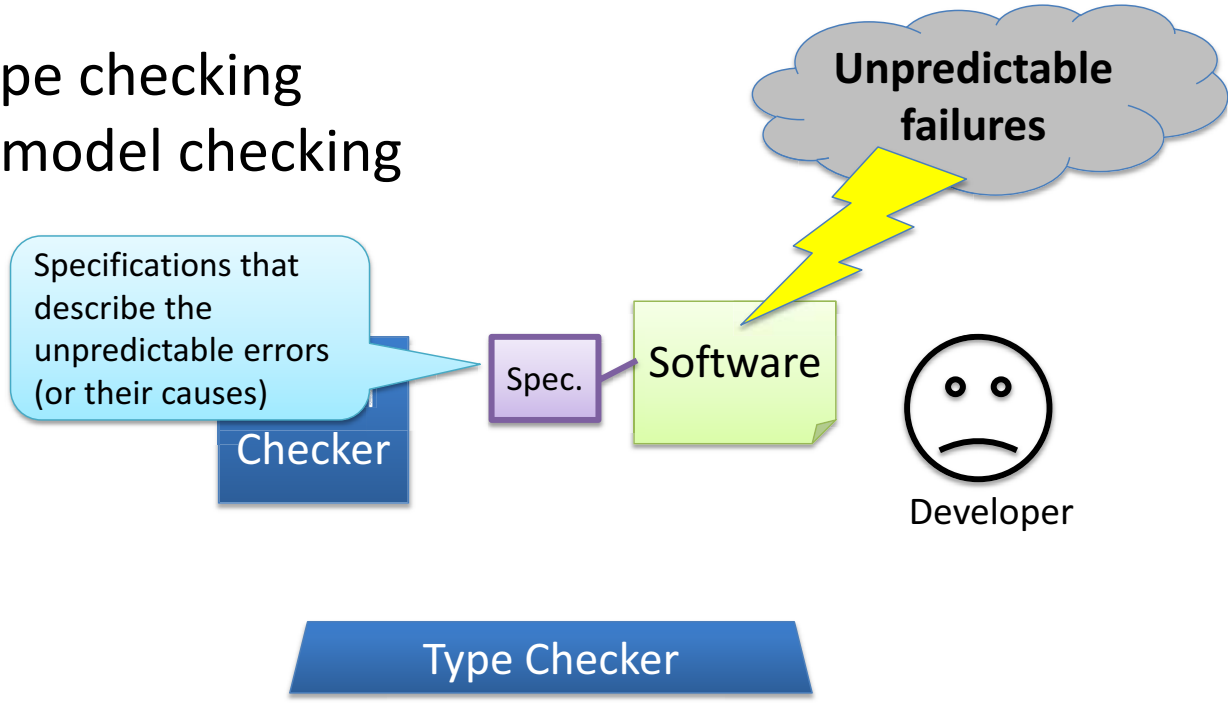
Our approach: utilizing two verification approaches in a complementary way

- Type checking & model checking



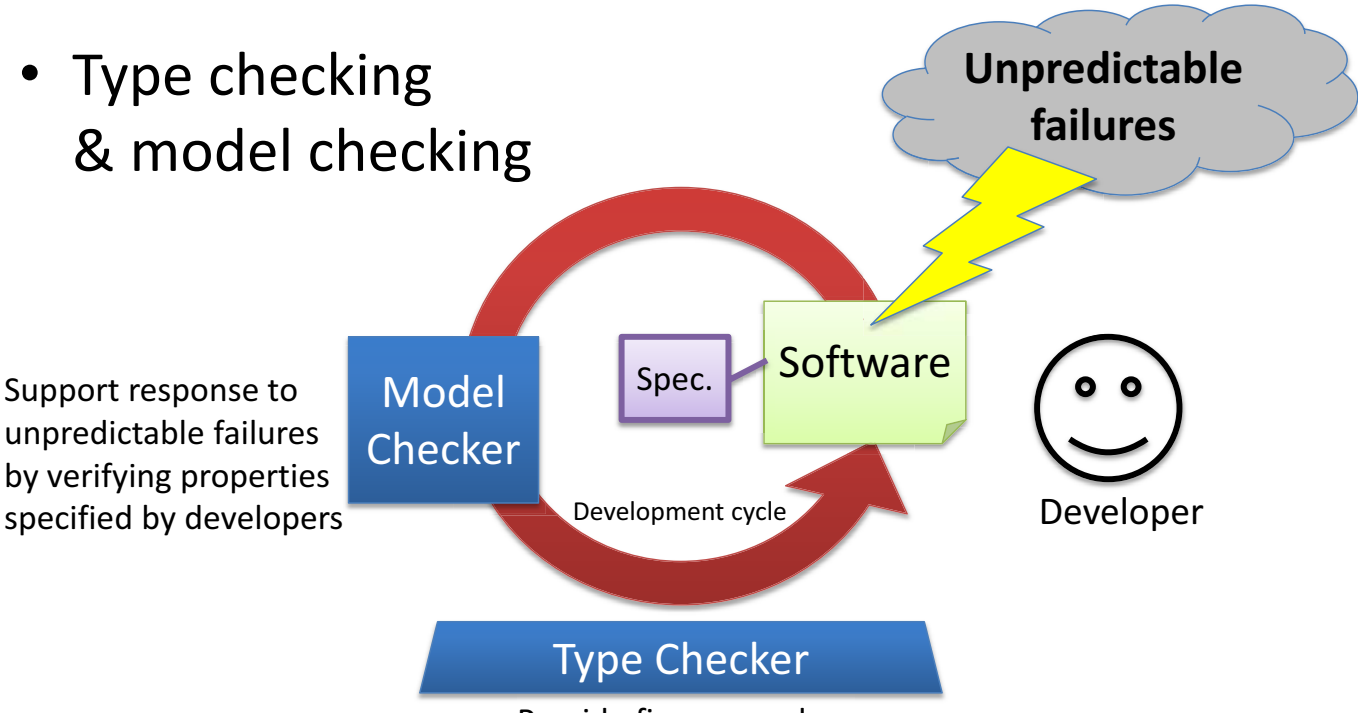
Our approach: utilizing two verification approaches in a complementary way

- Type checking & model checking



Our approach: utilizing two verification approaches in a complementary way

- Type checking & model checking



Verifications in the DEOS process: Imaginary scenario

- An automatic ticket gate goes wrong
  - End users noticed that they couldn't pass through the gate
- Then, station staff/engineers try to grasp the situation
  - Automatic doors don't work?
  - Contactless card readers don't work?
  - etc…
- If the situation is within the scope of D-Case of the gate, it should tell what to do in the situation
- Otherwise…

Verifications in the DEOS process: Imaginary scenario

- The situation is not covered by D-Case of the gate, that is,
  - it is an unexpected situation, and
  - it cannot be recovered/avoided by runtime dependability mechanisms
- Then, the developers of the gate try to infer factors that cause the situation
  - They can utilize the D-Case diagram of the gate and logs gathered by monitoring mechanisms
- If the inferred causes are software-related issues, then…

Verifications in the DEOS process: Imaginary scenario

- The developers express the causes as specifications and describe (or revise) the specifications in a specification language

- Then, the developers try to fix programs used in the gate according to the specifications
  - During program modification, type checking can prevent the developers from introducing subtle bugs (e.g., segmentation faults)

Verifications in the DEOS process: Imaginary scenario

- After fixing the programs, the developers can utilize model checking in order to check whether the inferred causes are really solved

  - If not, fix the programs again

Verifications in the DEOS process: Imaginary scenario

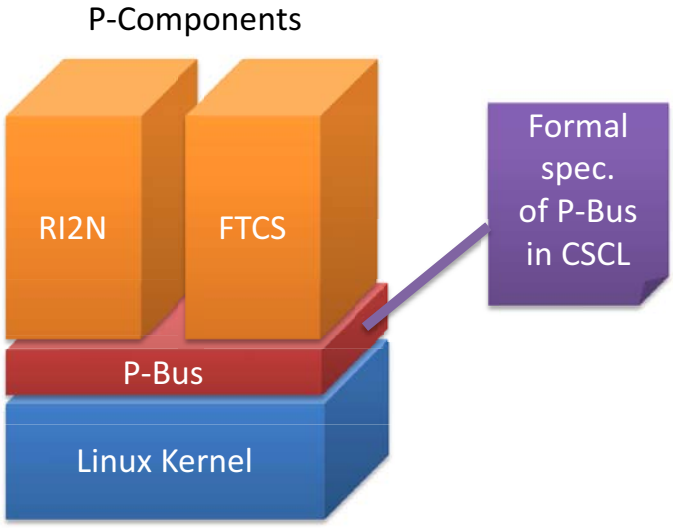
- If the inferred causes are surely solved, the developers conduct several tests and performance benchmarks
  - in order to make sure that the situation never occurs
    - If the situation still occurs, repeat the process from inferring the factors that cause the situation

- Finally, the D-Case diagram is updated in order to reflect the experiences obtained through the development process

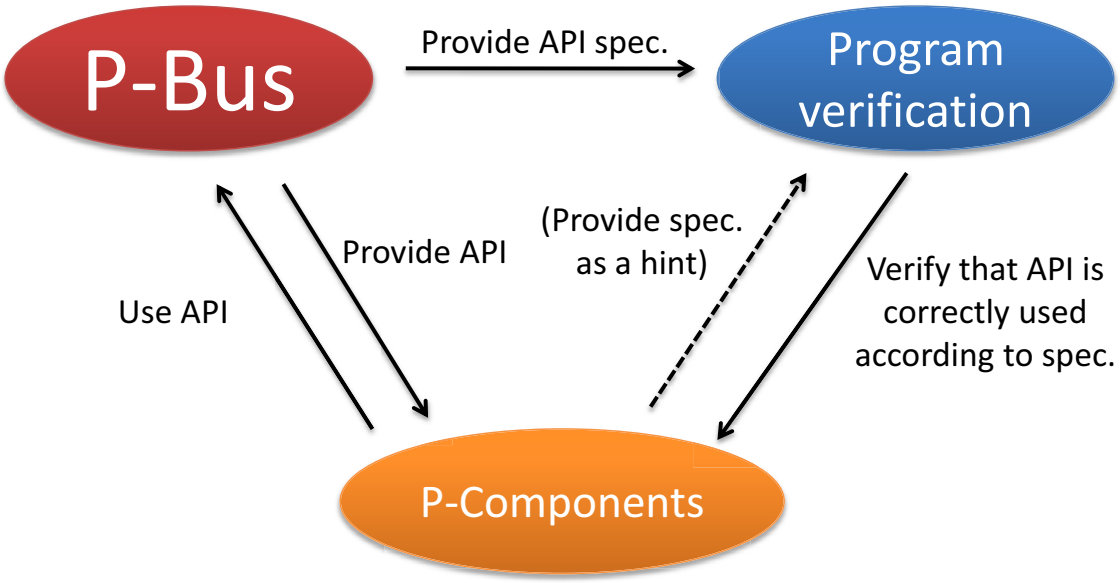
Application to systems software mechanisms for Open Systems Dependability

- Design and implement an OS kernel API (P-Bus) for kernel extensions (P-Components)

- Formally define specifications for P-Bus using our specification description language (CSCL)



Overview of P-Bus, P-Components, and program verification



Case study: checking RI2N P-Component

- RI2N P-Component
  - Multi-link Ethernet for high-bandwidth and fault-tolerant network
  - About 3000 lines of code
    - Slight modification of source code is required
    - It took up to half an hour to perform model checking

How many bugs did we find?

- 3 bugs
  - 2 with our model checker
    - Missing lock release
    - Accessing uninitialized timers
  - 1 with our type checker
    - Accessing unallocated memory
    - They could not be found
      by a certain commercial static analysis tool

Conclusion

- Goal of Systems Software Verification team is to support dependability of DEOS mechanisms for Open Systems Dependability
- In the DEOS process,
  two verification approaches are utilized
  in a complementary way
  in order to tackle Open Systems Failures
  - Type checking and model checking