P-Bus and DEOS Verification Tools: Safe Extension Infrastructure in Linux Kernel

Yutaka Ishikawa, Hajime Fujita,
Toshiyuki Maeda, Motohiko Matsuda,
University of Tokyo
Shinichi Miura, and Mitsuhisa Sato
University of Tsukuba

An operating system is safe
1) Because all source codes are reviewed by many engineers, and/or
2) Because the system has been running for more than ten years without any troubles

When a new OS function is implemented to be adapted to a new environment or against some threat, how the OS safety is guaranteed.

Overview

- **Major Concerns in P-Bus and DEOS Verification Tools**
  - Operating System is forever modified/upgraded
    - To provide a new dependability against some threat or some malfunctions
    - To provide a new function required by users
    - To run on new computer architectures, e.g., many cores, new devices, …
  - Bugs are often injected in a new kernel module [1]

[1] A. Chou et al., "An Empirical Study of Operating System Error," In Proc. 18th ACM Symp. Operating System Principles (SOSP) , pp. 73-88, 2001.

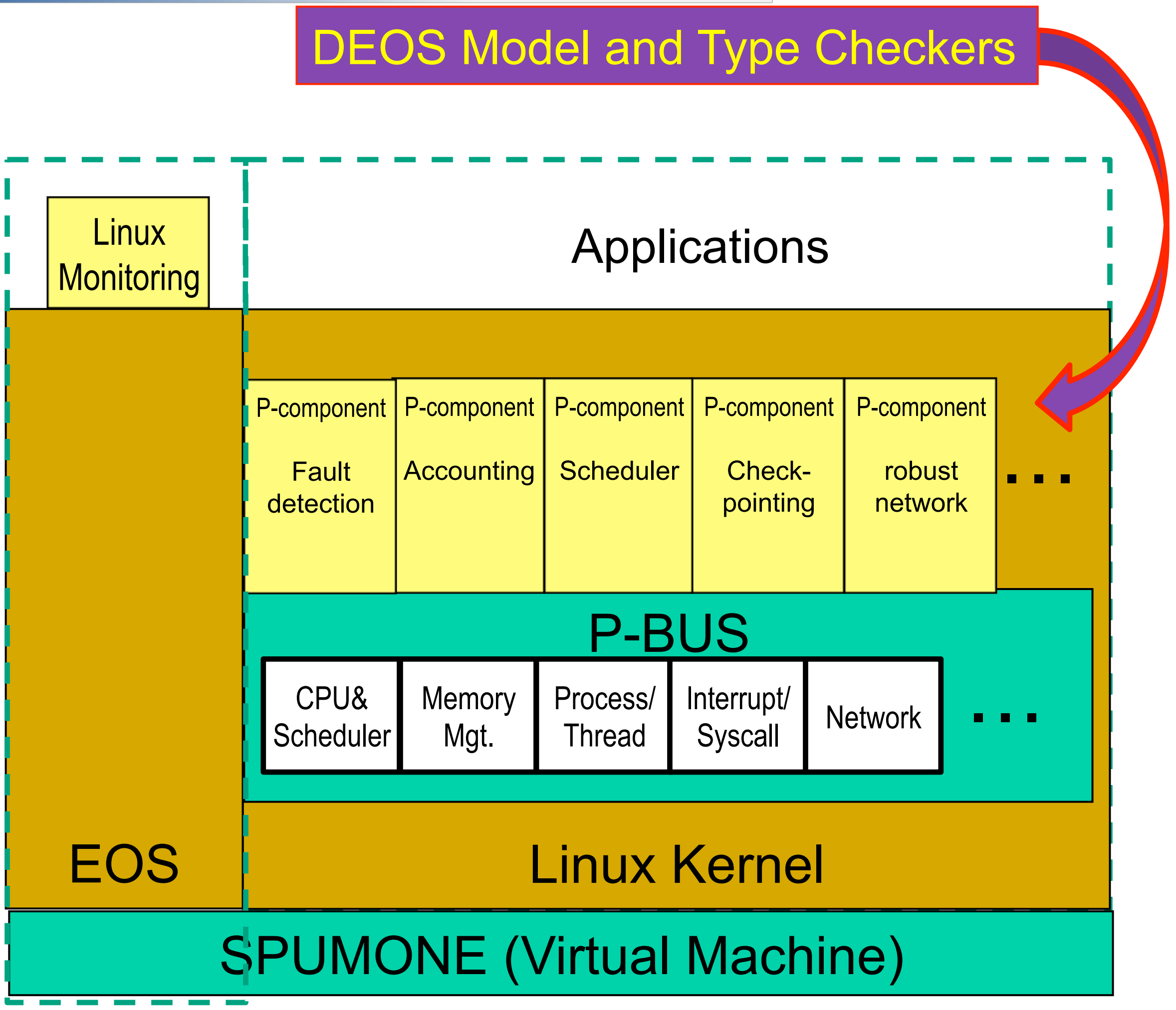
- **Approach**
  - Providing API with formal specification for OS extensions
    - A new extension is implemented using the API
  - Providing verification tools to check if the new OS modules are correct
- **Products**
  - P-Bus
  - DEOS model and type checkers

Outline of Talk

- P-Bus 1.0
- DEOS Verification Tools
  - Model Checker
  - Type Checker
- Case Study
- Related Work
- Rethinking of Our Approach
- Summary

An Overview of P-BUS

- P-BUS
  - Abstraction of Kernel Functions
  - API with formal specification for programming P-Component
  - Linux Kernel Module
- P-Component
  - implements an additional kernel function to enhance dependability
    - E.g., fault recovery/avoidance mechanisms, monitoring/ tracing
  - implements new device drivers
  - runs under the kernel mode
    - Not implemented by a user process such as micro kernels
  - is statically verified by DEOS tools
- Linux kernel
  - is a minimum Linux kernel



- SPUMONE & EOS
  - monitors the Linux activity to detect malfunctions and to recover the OS

P-Bus Design Philosophy

Linux

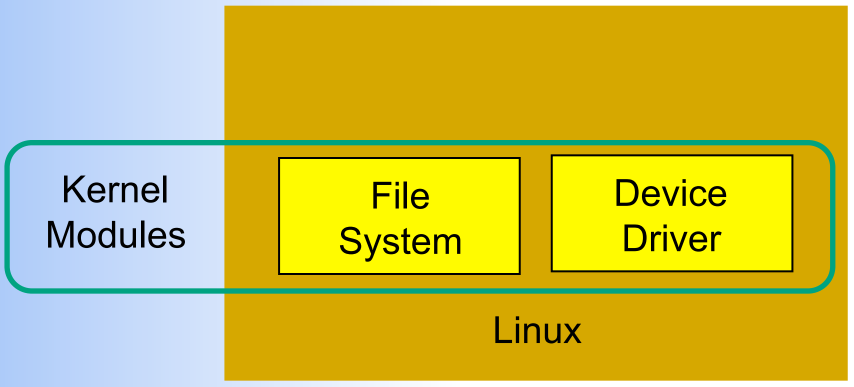
- APIs for Kernel Extensions
  - No documentations
    - Programmers misunderstand how to use APIs provided by Kernel
  - Varying
    - Different minor versions may differ different APIs !

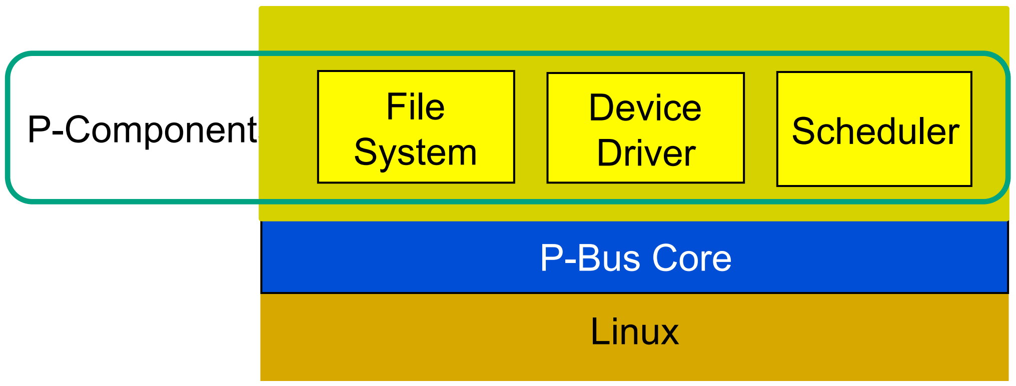
P-Bus

- Abstraction of Kernel Function
- API with formal specifiation for programming P-Component

P-Component

- implements an additional kernel function to enhance dependability
- implements new device drivers
- is statically verified by DEOS tools



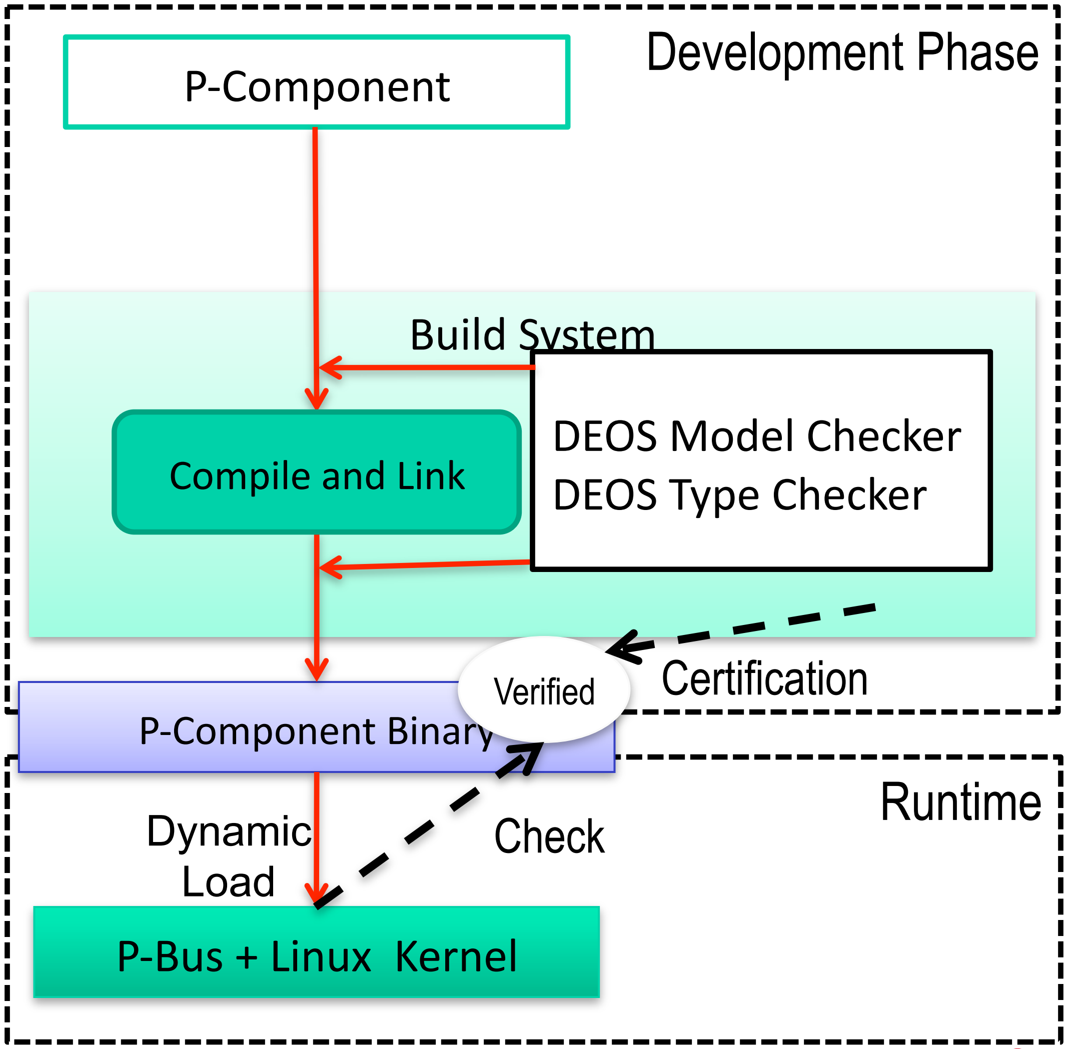


An Example of P-Bus Interface and Verification

int pbus_bmtx_extrylock(pbu_bmtx_t *mtx)
tries to hold a blocking mutex.

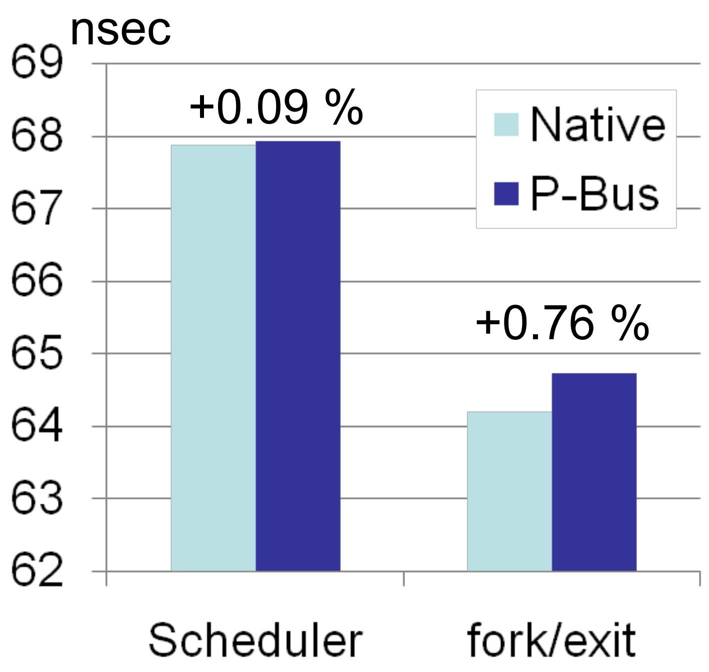
```
/*@ requires context == PBUSV_CTX_PROCESS;
    requires \valid(mtx);
    requires  *mtx != PBUSV_UNINITIALIZED;
    assigns   *mtxt;
    ensures  \result == 0 || \result == EBUSY;
    ensures  \result == 0 ➔ *mtx == EX_LOCKED;
    ensures  \result == EBUSY ➔ *mtx == \old(*mtx);
*/
```

Context	Process Context only
May block or not	No
Pre-conditions	mtx must be initialized by pbus_bmtx_init
Return value	0 on success EBUSY on failure
Post-conditions	mtx shall be locked on success, otherwise mtx is kept unchanged



P-Bus: Overhead

- **Extendibility**
  - **Network Driver**
    - **RI2N, high-bandwidth and fault-tolerant network with multi-link Ethernet [Miura08]**
  - **Schedulers**
    - **EDF, Earliest Deadline First, scheduler**
    - **Gang scheduler**
      - **A group of processes , a parallel job, runs simultaneously in a multi-core computer**
- **Overhead**
  - **Scheduler**
    - **Comparing with the schedule function**
  - **Fork/exit system call**



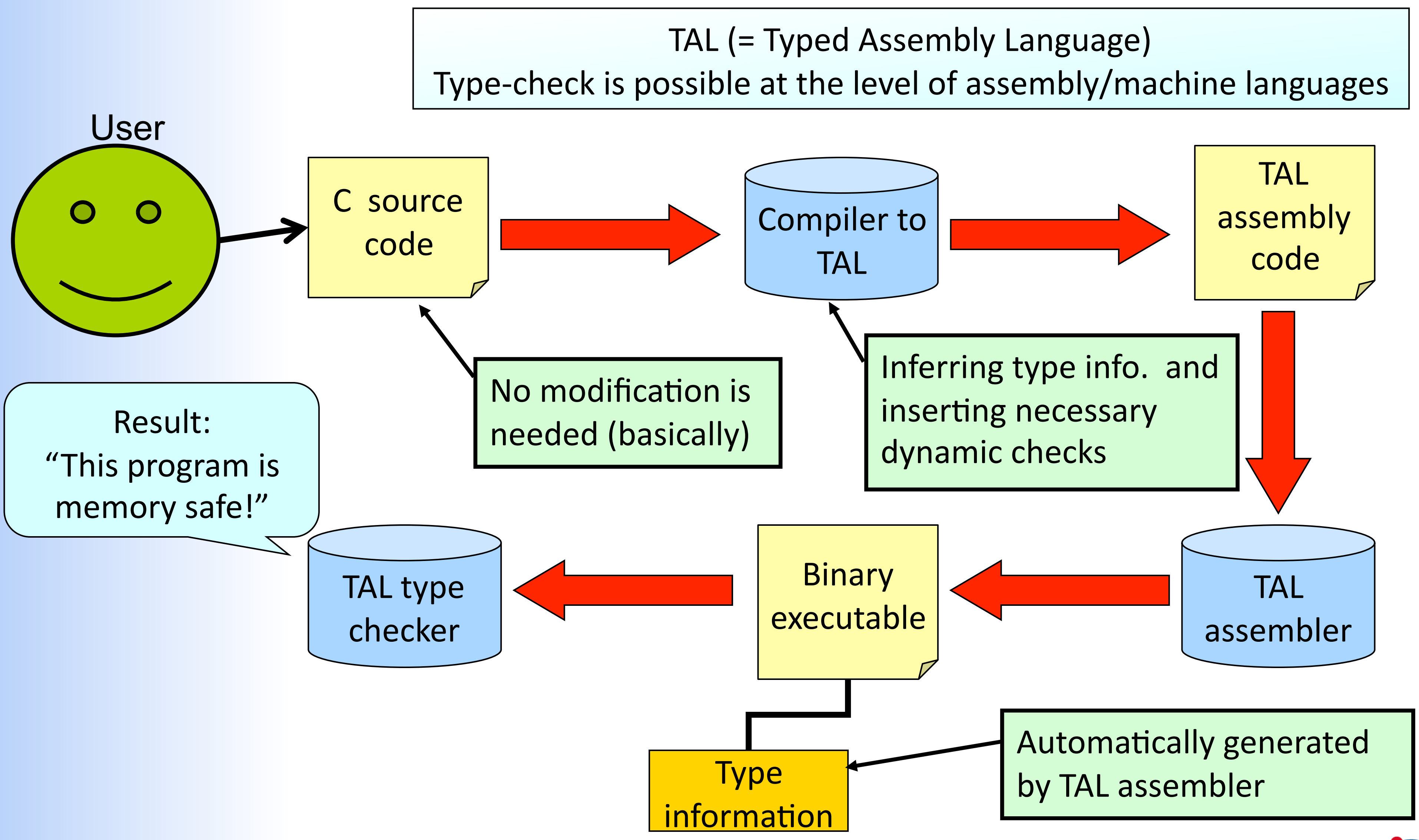
Linux 2.6.24.7
Dual Core AMD Opteron Processor 175 (2.2GHz)

Outline of Talk

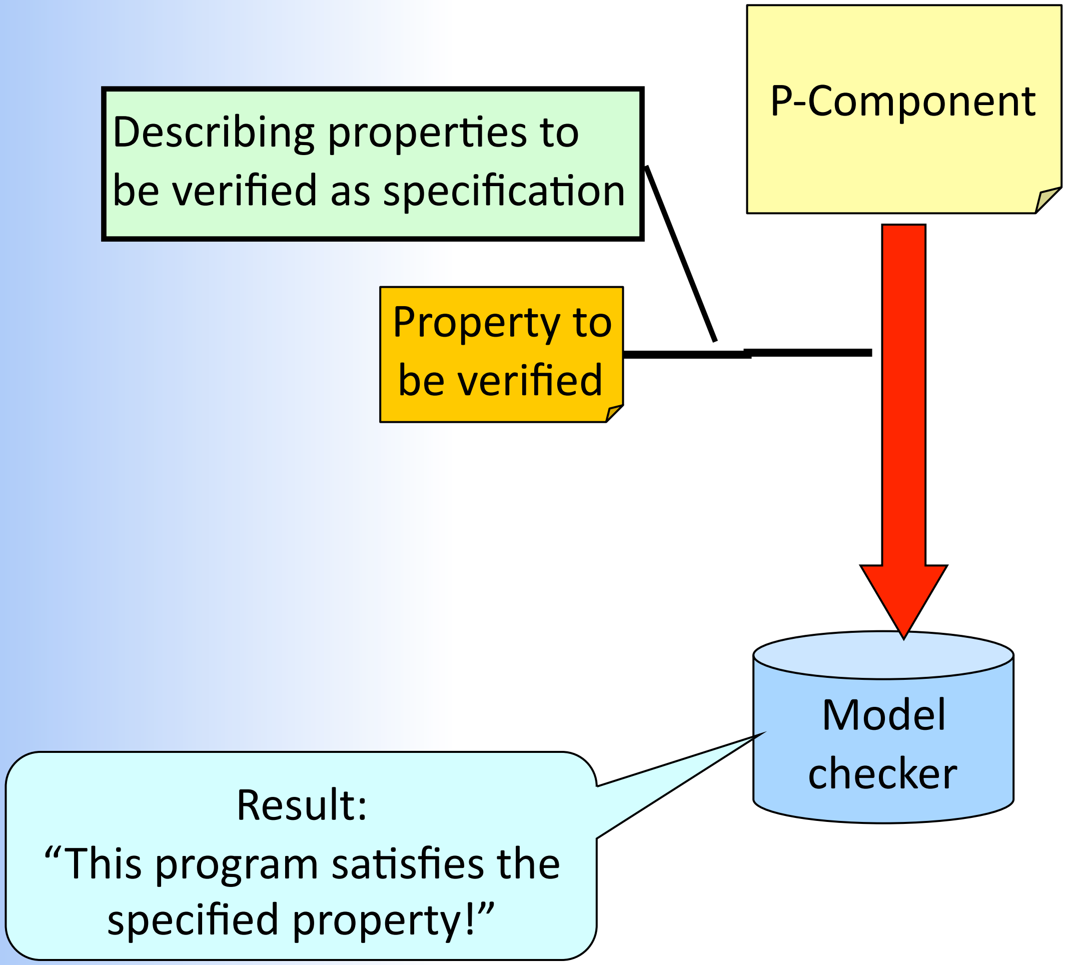
DEOS Type and Model Checkers

	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

Overview of DEOS Type Checker


TAL (= Typed Assembly Language)
Type-check is possible at the level of assembly/machine languages
User
C source code
Compiler to TAL
TAL assembly code
No modification is needed (basically)
Inferring type info. and inserting necessary dynamic checks
Result: “This program is memory safe!”
TAL type checker
Binary executable
TAL assembler
Type information
Automatically generated by TAL assembler

Overview of DEOS Model Checker


Describing properties to be verified as specification
P-Component
Property to be verified
Model checker
Result: "This program satisfies the specified property!"

```
int pbus_bmtx_extrylock(pbu_bmtx_t *mtx)
    tries to hold a blocking mutex.
```

```
/*@ requires context == PBUSV_CTX_PROCESS;
    requires \valid(mtx);
    requires  *mtx != PBUSV_UNINITIALIZED;
    assigns   *mtxt;
    ensures  \result == 0 || \result == EBUSY;
    ensures  \result == 0 ➔ *mtx == EX_LOCKED;
    ensures  \result == EBUSY ➔ *mtx == \old(*mtx);
*/
```

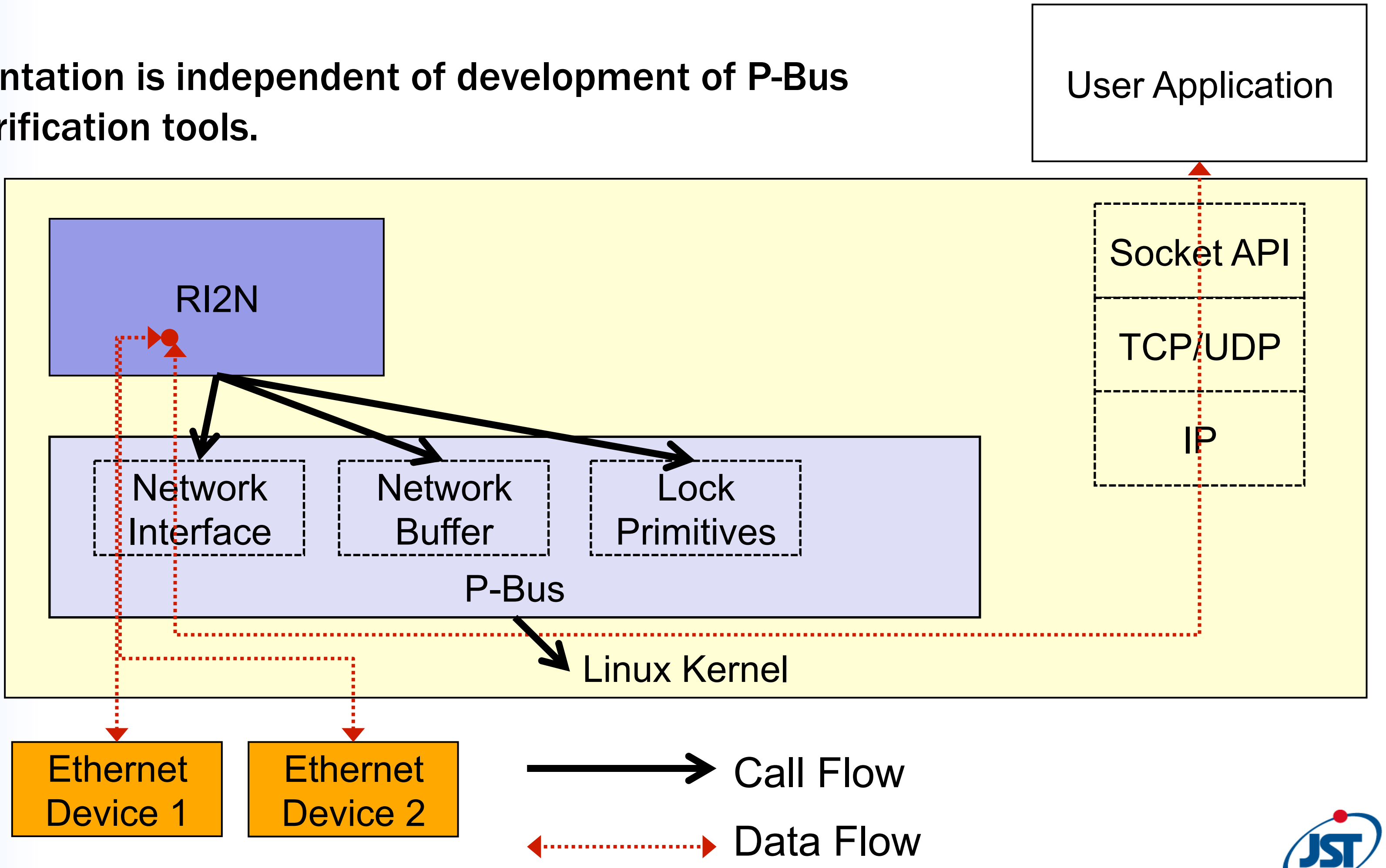
Context	Process Context only
May block or not	No
Pre-conditions	mtx must be initialized by pbus_bmtx_init
Return value	0 on success EBUSY on failure
Post-conditions	mtx shall be locked on success, otherwise mtx is kept unchanged

Outline of Talk

- P-Bus 1.0
- DEOS Verification Tools
  - Model Checker
  - Type Checker
- Case Study
- Related Work
- Rethinking of Our Approach
- Summary

RI2N P-Component

- RI2N is a fault-tolerant network developed at University of Tsukuba.
- The fault-tolerance is implemented with redundant network devices
- The implementation is independent of development of P-Bus and DEOS verification tools.



Case Study: How many bugs have we found ?

- Three Bugs
  - Two Bugs found by DEOS model checker
    - Missing lock release
    - Accessing uninitialized timers
  - One Bug found by DEOS type checker
    - Accessing unallocated memory

Bug 1: Missing lock release (found by DEOS model checker)

```
static int ri2n_add_slave(pbus_netif_t *netif,
                          pbus_netif_t *slave_netif) {
  struct ri2n_priv_t *priv = anlab_netif_private(netif);
  …
  pbus_net_giant_lock();

  root = priv->chl_list;
  if (root == NULL) {
    priv->chl_list = root =
      pbus_alloc(sizeof(struct ri2n_list),
                 PBUS_ALLOC_NOWAIT | PBUS_ALLOC_ZERO);
    if (root == NULL) {
      ri2n_error_msg("pbus_alloc fault\n");
      return 1;
    }
  …
```

A lock is acquired here, but …

Forgot to release the lock!

Bug 2: Accessing unallocated memory (found by DEOS type checker)

```
static int ri2n_priv_init(pbus_netif_t *netif) {
  struct ri2n_priv_t *priv = pbus_netif_private(netif);
  …
  pbus_nbmtx_init(&priv->tablock);
  …
}

int ri2n_setup(void) {
  pbus_netif_t *pbus_netif;
  …
  if (0 != pbus_create_netif(
           &ri2n_netif_ops,
           &ri2n_proto_handler,
           &ri2n_netif_param, &pbus_netif)) {
    …
  }
  …
  rval = ri2n_priv_init(pbus_netif);
}
```

The memory pointed by “priv” has not been correctly allocated, but it works because the area has not been used for other purposes

No valid pointer is assigned to “priv”!

Bug 3: Accessing uninitialized timers (found by DEOS model checker)

```
void ri2n_cleanup(void) {
  …
  pbus_timer_cancel(&ri2n_buf_timer);
  …
}

int ri2n_setup(void) {
  …
  rval = ri2n_priv_init(pbus_netif);
  if (0 != rval) {
    ri2n_error_msg("ri2n_priv Initialize() fault\n");
    ri2n_cleanup();
    return -1;
  }
  …
  pbus_timer_init(&ri2n_buf_timer,
                  &ri2n_buf_timer_ops, NULL);
  …
}
```

A timer is accessed here, but …

The timer may not be initialized in error paths!

# of False Positive is 1

- Locks in memory heap could not be handled correctly by our model checker

```
static void ri2n_buf_timer_fn(pbus_timer_t *timer) {
  …
  for (i = 0; i < RI2N_HASHLEN; i++) {
    …
    do {
      if (ptr->cont != NULL) {
        …
        pbus_nbmtx_exlock(&node->lock);
        …
        pbus_nbmtx_exunlock(&node->lock);
        …
      }
    } while (ptr != root);
  }
  …
}
```

It seems that the lock is acquired and released correctly, but the current model checker does not take care of pointer variables

Outline of Talk

Related Work: Model Checking Tools

- BLAST (Thomas A. Henzinger et al., EPFL)
  - Properties reducible to graph reachability can be verified
    - Properties can be specified by users
      - State-machine based specification language
  - C source code can be verified directly
    - Lazy predicate abstraction approach: more expensive, less conservative
- SDV (Microsoft)
  - Properties reducible to graph reachability can be verified
    - Properties cannot be specified by users
  - C source code can be verified directly
    - Predicate abstraction approach: less expensive, more conservative
- SPIN (Gerard J. Holzmann et al., Bell Labs ?)
  - Properties described in LTL (Linear Temporal Logic) can be verified
    - Properties can be specified by users
  - C source code cannot be verified directly
- DEOS Model Checker
  - Properties reducible to graph reachability can be verified
    - Properties can be specified by users
      - Assertion based specification language (a dialect of ACSL)
  - C source code can be verified directly
    - Predicate abstraction approach: less expensive, more conservative

Related Work: Type Checking Tools

- CCured (George Necula et al., UCB)
  - Memory safety is ensured through type inference
  - A little modification of C source code is (typically) required
- Fail-Safe C (Yutaka Oiwa, AIST)
  - Memory safety is ensured by inserting dynamic checks
  - No modification is required basically
- Deputy (Jeremy Condit et al., UCB)
  - Memory safety + α (invariants about null-terminated pointers etc.) is ensured through type checking of dependent types and inserting dynamic checks
  - Explicit type annotations are required basically
- DEOS Type Checker
  - Memory safety is ensured by inserting dynamic checks
  - Memory safety of generated assembly code can be verified through type checking
  - No modification is required basically

Outline of Talk

- P-Bus 1.0
  - .....
- DEOS Verification Tools
  - Model Checker
  - Type Checker
- Case Study
- Related Work
- Rethinking of Our Approach
- Summary

Rethinking P-Bus/P-Component (1/2)

- Original Design Philosophy
  - P-Bus APIs define the basic kernel functions and extension capabilities
    - API for extensions
      - Device drivers, scheduler, and so on
      - The API is different than API for customization/extension provided by original Linux
    - API for basic kernel operations
      - Locking /unlocking semaphore, sleep/wakeup, and so on
  - P-Bus APIs are defined with formal specification
    - A kernel module implemented with the P-Bus API is called a P-Component
    - A P-Component is validated using the DEOS verification tools

- Issues in P-Bus 1.0
  - It is assumed that all extensions are described using P-Bus APIs
    - This approach is something like defining the specification of a new micro kernel inside Linux kernel
  - Actual Linux extensions are based on extension capabilities provided by Linux kernel with patching
    - P-Bus does not assume such a case
  - P-Bus approach is creation of a new world in the Linux kernel, that might not be accepted by the Linux community

Rethinking P-Bus/P-Component (2/2)

- P-Bus 2.0
  - Because the Linux kernel provides APIs for customization/extension, the specification of those APIs is formally defined
    - VFS, network/block/character device interface
    - Socket interface
    - Netfilter interface
    - …
  - API for basic operations used by extended modules is formally defined. This is the same as P-Bus 1.0
    - Locking /unlocking semaphore, sleep/wakeup, and so on

Summary

- P-Bus/P-Component and DEOS type and model checkers have been introduced
- A result of the case study shown in this presentation demonstrates that our approach is effective and contributes safety of OS modules
- However, P-Bus and DEOS verification tools prove limited correctness of OS modules. Functional properties of OS modules cannot be validated unlike the seL4 approach

References

- Yutaka Ishikawa, et.al, "Towards an Open Dependable Operating System,“ IEEE 12th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2009 (Invited Paper).
- Toshiyuki Maeda and Akinori Yonezawa, "Writing an OS Kernel in a Strictly and Statically Typed Language", Lecture Notes in Computer Science 5458, pp. 181-197, May 2009.
- Motohiko Matsuda (Univ. of Tokyo), Toshiyuki Maeda (Univ. of Tokyo) and Akinori Yonezawa (Univ. of Tokyo), "Towards Design and Implementation of Model Checker for System Software", The 1st International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009), Tokyo, Jan. 2009.
- Takahiro Kosakai, Toshiyuki Maeda and Akinori Yonezawa, "Compiling C Programs into a Strongly Typed Assembly Language", In Proc. of ASIAN'07, Lecture Notes in Computer Science 4846, pp. 17-32, Dec. 2007.
- etc.