Dependability and Verification

Systems Software Verification Team

Hajime Fujita, Toshiyuki Maeda,
Motohiko Matsuda, Shuichi Oikawa

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

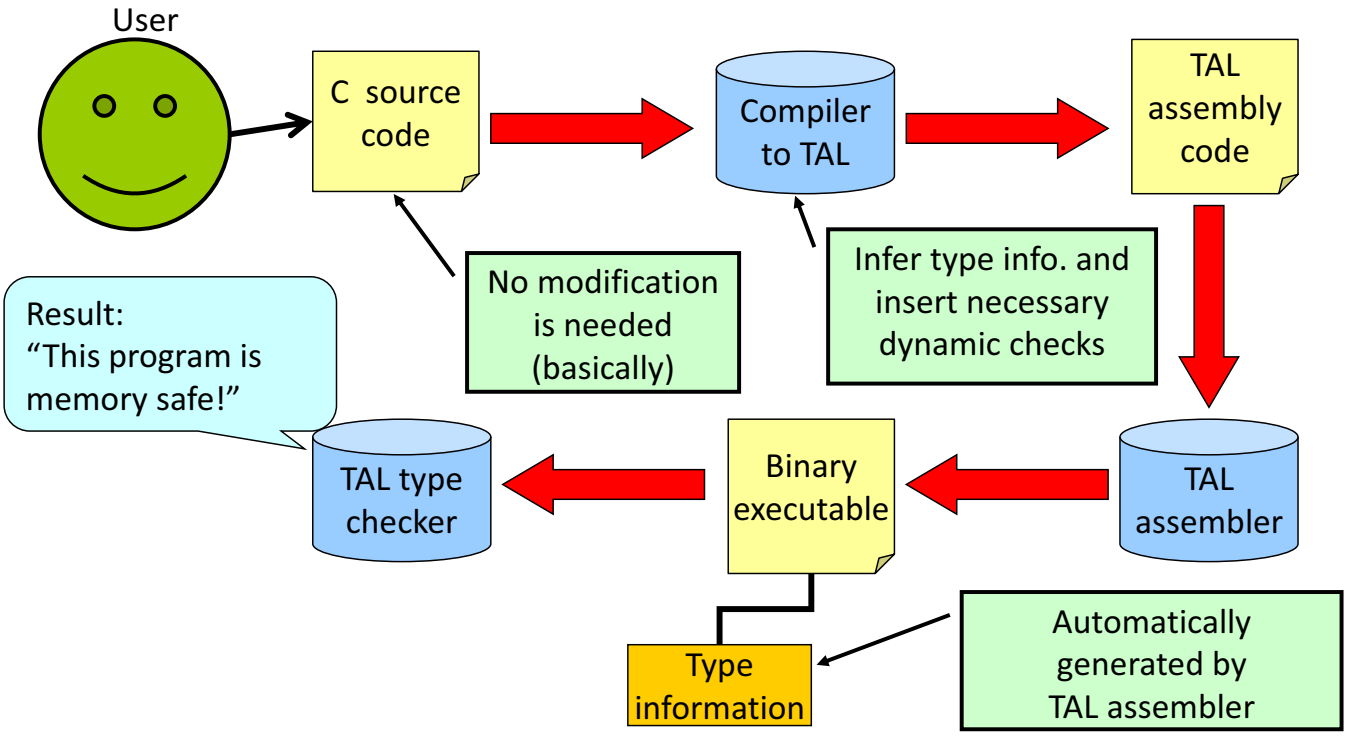
Basic safety

- Properties that can be reduced to simple type safety
  - Memory is consistent with respect to certain types at any time
- E.g.,
  - Memory safety
  - Control-flow safety
  - etc.

Advanced safety

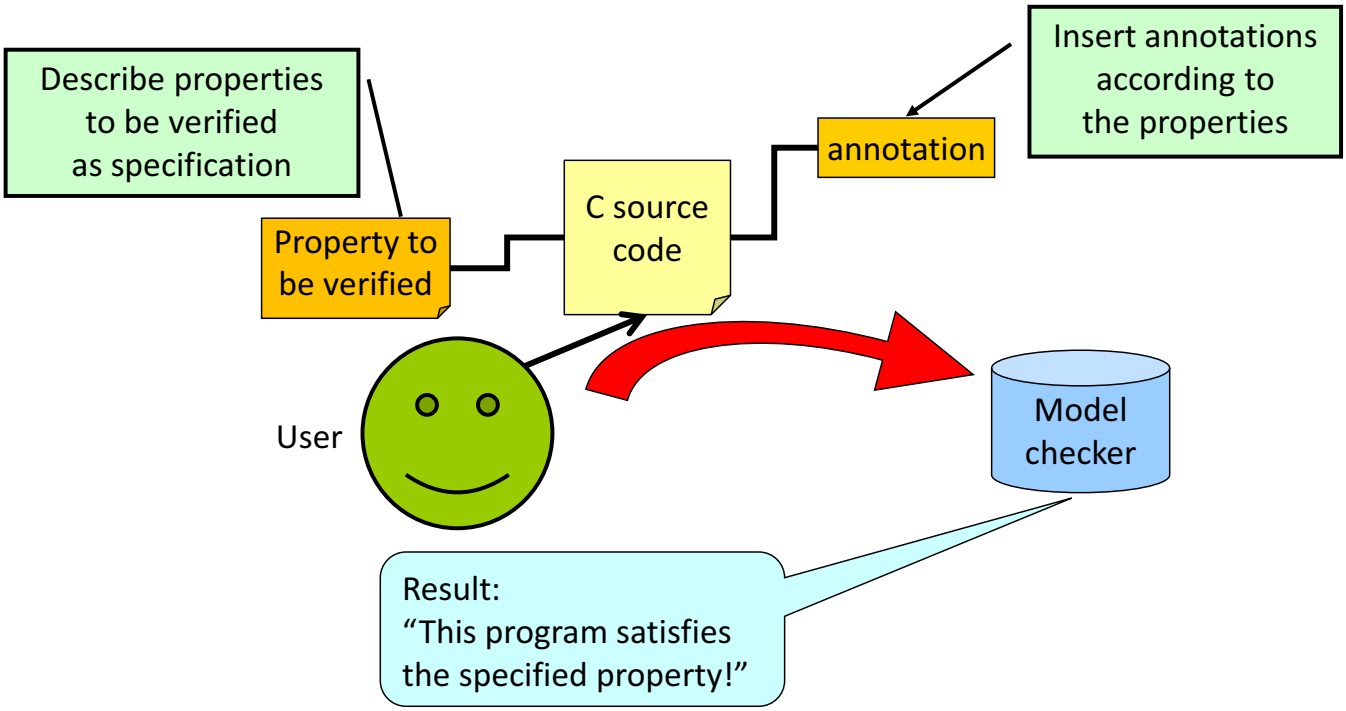
- Properties that can be reduced to graph reachability
  - Program execution never reaches wrong states from the initial state
- E.g.,
  - Consistency of locks
  - Consistency of contexts
  - Correct API usage
  - etc.

Overview of type checking



TAL (= Typed Assembly Language)
Type-check is possible at the level of assembly/machine languages

Overview of model checking



Example of specification description

Pre-conditions of the function

The lock must not be held by the running thread

```
/*@
    $requires (*mtx == PSPC_MTX_UNKNOWN);

    $ensures ($result == 0 || $result == EBUSY);

    $ensures ($result == 0
                ==> *mtx == PSPC_MTX_EX_OWNED);
 */
static inline int pbus_nbmtx_extrylock (pbus_nbmtx_t *mtx);
```

Return value must be 0 (success) or EBUSY (busy)

If success, the running thread holds the lock

The function that tries to acquire a mutex lock exclusively

Post-conditions of the function

Our specification language CSCL is defined as a dialect of JMA and ACSL specification languages.

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

Properties that can be covered by P-Bus specification

- Type check
  - Memory safety
    - Never perform illegal memory access
  - Control-flow safety
    - Never perform illegal code execution
- Model check
  - Lock consistency
    - Never release a lock that is not held
    - Never acquire a lock twice without releasing it, etc.
  - Execution context consistency
    - Never sleep (block) while holding a lock, etc.

Properties NOT covered by P-Bus specification

- Resource consumption safety
  - Absence of memory leaks, etc.
- Timing constraints
  - Ensuring real-time processing, etc.
- etc.

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

Bug 1: Missing lock release

(found by our 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 our 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" is overwritten, but …

No valid pointer is assigned to "priv"!

Bug 3: Accessing uninitialized timers

(found by our 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!

Comparison with other tools

	Target property	Target program	Necessity of C code modification	User-defined property	Specification language	Verification time
Our Type Checker	Basic safety	C and binary executable	No	No	N/A	Short
CCured	Basic safety	C	Yes (partially)	No	N/A	Short
Fail-Safe C	Basic safety	C	No	No	N/A	Short
Deputy	Basic safety + α (static array bounds checking etc.)	C	Yes	Yes (partially)	Dependent Types	Short
Our Model Checker	Advanced safety	C	Yes	Yes	ACSL-based assertion lang.	Long
BLAST	Advanced safety	C	Yes	Yes	State machine-based lang.	Long
SDV	Advanced safety	C	No(?)	No	N/A	Long
SPIN	LTL property (beyond safety)	Promella	Yes	Yes	LTL-based lang.	Long

Basic safety = memory safety, etc.

Advanced safety = consistency of locks, correct API usage, etc.

Related work: Type Checking Based 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
- Our 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

Related work: Model Checking Based 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
- Our 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

Conclusion

- 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

- Prototypes of a type checker and a model checker have been designed and implemented
  - They could find several bugs in a P-Component

Publications

- Refereed international conferences
  - Takahiro Kosakai, Toshiyuki Maeda and Akinori Yonezawa, “Compiling C Programs into a Strongly Typed Assembly Language”, In Proc. of ASIAN'07, pp. 17-32, 2007.
  - 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”, In Proc. of STFSSD 2009.
  - 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, 2009.
  - Toshiyuki Maeda and Akinori Yonezawa, “Typed Assembly Language for Implementing OS Kernels in SMP/Multi-Core Environments with Interrupts”, In Proc. of SSV 2010.
  - Toshiyuki Maeda and Akinori Yonezawa, “Extended Alias Type System using Separating Implications”, In Proc. of TLDI 2011. To appear.