An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Lena Feinbube, Daniel Richter, and Andreas Polze
Operating Systems & Middleware Group
Hasso Plattner Institute at University of Potsdam, Germany
Reality...

- usual assumption:
  - linear relationship between faults, errors, and failures
- but...
  - relation between faults, errors, and failures is complex
  - consequences of a bug are arbitrarily related in time, space, and severity to the cause
  - error state may arise only if multiple faults are activated under certain conditions
  - several error states may necessary for a system failure
  - interaction between multiple software components frequently accounts for software outages
Motivation

- fault injection: testing complex software system’s fault tolerance and overall dependability
  - artificially inject fault & error states into running system
  - observe how well these situations are handled
- one central question: which faults and error states to inject, and when?
- failure cause model: describes what is injected (into running program)
- need for a realistic failure cause model
- faultload representativeness
Motivation

- fault injection testing at interfaces is powerful

**Hovac**

- dependability benchmarking & fault injection tool
- orchestrates fault injection campaigns
- repeatable & configurable
- injection at interface level (function calls to external libraries)
- failure-cause model: misbehavior of external, third-party code
- implementation: dll API hooking (*Detours* library)

Motivation

▪ Common Weaknesses Enumeration (CWE) Database (by Mitre)
  ▪ classify all kinds of software weaknesses
    ▪ i.e. programming language, severity, kinds of error states
  ▪ provides realistic failure data
  ▪ based on experiences of research & industry

▪ realistic fault injection experiments: failure cause models should base on such community-gathered empirical data
Motivation

▪ **our contribution**: error model for dependability benchmarking with Hovac; error classes derived from CWE database

requirements for fault injection error model:

▪ formality
  ▪ existing error descriptions (bug reports, commit messages): anecdotal, textual descriptions of error state leading to failure
  ▪ aim: more formal definition, less specific
requirements for fault injection error model (contd.)

▪ executability
  ▪ possibility to implement for fault injector
  ▪ execution triggers the desired error state
  ▪ ideal: non-intrusive, applicable to arbitrary software, general & application specific error states

▪ realism
  ▪ assess the quality of fault-tolerance mechanisms: only useful if faults and error states correspond to real world problems
An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Agenda

- research gap outline
- error classes derived from CWE database
- abstract formalization of such errors
  - concepts: state, functions, & processes
  - examples
- practical implementation of error classes within our prototype fault injection tool, Hovac
- evaluation of error model
- discussion & future work
Research Gap

what we are looking for: **error models** which are both

- suitable for fault injection (i.e., executable and based on realistic data)
- generalizable
Research Gap

- **bug fixes, or generalized patterns of such fixes**

- **behavioral models**

- **formal grammar-based fault specifications**

- **Common Weakness Enumeration database**

- **Orthogonal Defect Classification**
what we are looking for: **error models** which are both

- suitable for fault injection (i.e., executable and based on realistic data)
- generalizable
Error Model

- A system failure is an event that occurs when the delivered service deviates from correct service. A system may fail either because it does not comply with the specification or because the specification did not adequately describe its function.


- failure cause model (or “fault model”) is complement to program specification
  - what can go wrong?
  - often implicit & not stated explicit
  - aim: explicit error model
Error Model

overview of error classes

▪ computation
▪ environment
▪ timing
▪ race condition
▪ memory
▪ control flow
Error Model

Computation

- variables, in particular computation results of primitive data types, contain a value different from what was expected.
  - Off by One (CWE ID 193)
  - Signed to Unsigned Conversion (CWE ID 195)

Timing

- certain part of the code takes more than the expected time to execute
  - Hovac: call to a library function returns too late
An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Error Model

Control Flow

- input triggers an incorrect execution path through the application
  - unhandled exceptions

Environment

- interaction between the program and its environment is other than expected; unforeseen states in the execution environment or the operating system; programmer’s assumptions regarding the environment are violated
  - Signal Errors (CWE ID 387)
Error Model

Race Condition

- accesses to shared memory are not properly synchronized
  - switch statements (CWE ID 365)
  - data shared between multiple threads (CWE ID 366)
  - signal handlers (CWE ID 364)

Memory

- state of the memory is corrupted.
  - specifically Hovac/C/C++: corruption or leaking of heap & stack memory due to programming mistakes
  - Heap-/Stack-based Buffer Overflow (CWE IDs 121-122)
Basic Formal Model

▪ aim: a description, which is
  ▪ generalizable and applicable to diverse software systems
  ▪ works with automated dependability benchmarking and fault injection

▪ complementary approach to software verification.
  ▪ abstract specifications and invariants which a program must obey to function correctly: success space
  ▪ error states: explore the failure space
Basic Formal Model

- characterization of software error states in an abstract fashion, while using a minimal amount of machine-, language- and hardware architecture-dependent modelling concepts

- basic building blocks:
  - state
  - functions
  - processes

- static (only properties of current state needed)
- state sequences, environment state
An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Basic Formal Model

- **state** (internal & environment): set $R$ of resources
- resource: $r = (Resource\,State, Ownership) \in R$
- ownership: set of resource owners
  - $p_i \in P$ (processes in system)

- data in memory: $D \subset R$
  - $Resource\,State$: $s = \langle s_j, s_{j+1}, \ldots, s_k \rangle$
    - $j \geq 0 \land k \leq m$ (range of addressable memory)

- pre-defined states:
  - $scheduled(p \in P)$; next function call from process $p$
  - $output(r_1; r_2)$; resource state of $r_1$ is written to $r_2$
An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Basic Formal Model

- **functions**: type of events, which transfers input data to output data (= modifies state):
  \( f : I \rightarrow O \) where \( I, O \subseteq R \)
  - granularity can be arbitrary
  - error states are assumed to be observable only after events, i.e., at function boundaries

- pre-defined functions:
  - \textit{acquireResource}\((\langle s, O \rangle, p \in P)\); adds a process to the ownership of a resource: \( (s, O) \rightarrow (s, O \cup \{p\}) \)
  - \textit{releaseResource}\((\langle s, O \rangle, p \in P)\); removes a process from the ownership of a resource: \( (s, O) \rightarrow (s, O - \{p\}) \)
Basic Formal Model

- **processes**: sequential compositions of functions executed one after another

- multiple processes can run concurrently
  - within a process, functions are strictly ordered
  - assumption: at each time instant only one event occurs.
  - concurrency exists, but behavior is equivalent to a sequential system without hardware parallelism

- set of processes $P$ is fixed
Examples

- usage of first order logics quantifiers combined with Linear Temporal Logic (LTL) predicates
  - LTL is commonly used to denote properties of paths over time, or state sequences in a software system
- allows to express that a boolean fact or condition holds
  - Next – in the next state
  - Eventually/Finally – in some state in the future,
  - Globally – in all future states of the current execution path
Examples

Race Condition

- concurrent accesses to shared memory are not properly synchronized
  - all cases where the outcome can differ depending on the interleaving of two processes
  - here: sharing of one resource between processes

∃ r, out ∈ R, res ∈ ResourceState, p₁ ∈ P, s ∈ t_r :
 t_r |= r = (res, {p₁, p₂}) ∧ Next scheduled(p₁) ⇒
 ¬Eventually output(s, out) ∧
 t_r |= r = (res, {p₁, p₂}) ∧ Next scheduled(p₂) ⇒
 Eventually output(s, out)
Examples

Memory Leak

- allocated memory that cannot be used because the reference to it has been lost

\[ \exists p \in P : t_r \models \text{Exists } \neg \text{Globally } (r = (*, \{e\}) \land \text{Next } r = (*, \{p \in P \}) \Rightarrow \text{Eventually } r = (*, \{e\})) \]

- or -

\[ \exists p \in P : t_e \models \text{Exists acquireResource}(r, p) \Rightarrow \neg \text{Eventually releaseResource}(r, p) \]
Examples

Buffer overflow

- a memory region, or buffer, is written beyond its boundaries and no bounds checking was performed

- any operation (event) which modifies state according to this pattern constitutes a buffer overflow:

\[
(s_j, s_{j+1}, \ldots, s_k, \{p\}) \rightarrow
(s'_j, s'_{j+1}, \ldots, s'_k, \{p\}), (s'_j, \ldots, s'_{k'}, \ast)
\]

\[k' > j' > k \lor j' < k' < j\]
Implementation

Hovac

---

**Dllhooks.h**
- Attach hooks to detour to modified function:
  1. Modify input arguments
  2. Original function call
  3. Delay/ Memory Corruption/ Insert Race
  4. Modify result value
  5. Return result

---

https://github.com/laena/hovac
Implementation

- based both on formal model and on CWE DB
- selected CWE entries ("weaknesses") describe instances of our error classes
- for each error class, different errors – one per CWE entry – are implemented

- "static" errors (only operate on the current state)
  - activation takes place before or after the intercepted function call, function call itself takes place as usual

- "dynamic" errors
  - lambda passed into the activate function is relevant
Computation error (static)

- C++ template classes, an instantiation is implemented per type, containing the modification of arguments and return values

  - examples (CWE ID): Weakness Class: Incorrect Calculation (682), Off by One (193), Integer Overflow or Wraparound (190), Incorrect Conversion between Numeric Types (681)
sample code: Off by One computation error type

```cpp
template<typename Type>
class OffByOne : public ComputationError<Type> 
{
    public:
        OffByOne(std::string cweID = std::string("193")) : ComputationError<Type>(cweID) {}

        Any activate(AnyList arguments, LambdaPtr = nullptr) const override 
        {
            for (std::reference_wrapper<Type> arg : typeCastArguments(arguments))
                arg.get()++;
            return Any();
        }
};
POCO_EXPORT_CLASS(ComputationError<unsigned int>)
```
Environment error (static)

- execution environment is shared due to the API hooking approach – it can be manipulated programmatically
  - examples (CWE ID): Signal Error (387), Improper Privilege Management (269), Information Exposure Through Environmental Variables (526)
Timing error (dynamic)

- timing of a call to a third-party library is delayed artificially. C++ provides versatile means to do so using its thread support library.
  - examples (CWE ID): Excessive Iteration (834), Loop with Unreachable Exit Condition (835), Uncontrolled Recursion (674)
Race condition (dynamic)

- erroneous behavior in a new thread spawned by Havoc - for race conditions dependent on environment state and input data, no race conditions between multiple third-party libraries

- examples (CWE ID): Weakness Class: Concurrent Execution using Shared Resource with Improper Synchronization (362), Race Condition within a Thread (366), Time-of-check Time-of-use (TOCTOU) Race Condition (367), Context Switching Race Condition (368)
Implementation

**Memory error (dynamic)**

- manipulate heap and stack memory before or after the detoured function call (e.g. call `malloc` with an excessive size parameter)
  - examples (CWE ID): Allocation of Resources without Limits (770), Stack-based Buffer Overflow (121-122), Logging of Excessive Data (770), Out-of-bounds Write (787)
Implementation

Control flow error *(dynamic)*

- exception injection, to test exception handling mechanisms
  - examples *(CWE ID)*: Always-Incorrect Control Flow Implementation (670), Incorrect Behavior Order (696), Incorrect Control Flow Scoping (705)
Implementation

- sample code: exception injection

```cpp
template<typename Type>
class UncaughtException : public ControlFlowError
{
    public:
        UncaughtException() : ControlFlowError("248" /* cweID */) {};

        Any activate(AnyList arguments, LambdaPtr lambda) const override {
            std::string what = "uncaught exception";
            throw Type(what.c_str());

            (*lambda)();
        }
};

POCO_EXPORT_CLASS(UncaughtException<std::runtime_error>)
```
Evaluation

- formality:
  - our model is based on formal considerations
  - most error classes can be represented statically (just a snapshot of the running software and not a potentially infinite sequence of states needs to be considered)

- executability:
  - implementation conforming to our C++ AbstractError interface
  - the interface turned out to be versatile and expressive enough for all our needs.
  - our architecture allows for simple development of extension DLLs
Evaluation

- **realism**
  - based on CWE database, which contains empirical knowledge from real world industrial software systems
  - implementations of errors in the different classes based on a structured search of CWE database.
  - we used “C++” keyword to search, but additional not classified as C or C++-language relevant, also need to be considered
An Error Model for Multi-threaded Single-node Applications, and Its Implementation

Discussion & Future Work

- error model for fault injection
- based on community knowledge of software problems (CWE database)
- consists of a formalization of concepts needed for describing error states in a running system...
- ...as well as an implementation thereof
Discussion & Future Work

- further evaluation & extension of our error model
- provide an **automated deduction of error implementations** from bugs
- model excludes **probability and frequency** over time of error states
- integrate **profiling and field failure data**
- error model is **limited** to the application layer of a single, potentially multithreaded compute node
- extend error model to **cloud software systems**
  - distributed nature, complexity of virtualized software stack
Summary

- fault tolerance of complex software systems can be assessed experimentally using fault injection
- to become an effective & systematic testing strategy: requires a realistic and well-defined failure cause model
- failure cause models are frequently incomplete, informal, and implicit or application-dependent
### Summary

- We present a **formal error model** tailored for multi-threaded single-node applications.

- We provide a **formal error model** based on Common Weakness Enumeration (CWE) database of real world software problems to derive classes of error states:
  - Static (i.e., detectable from a snapshot of the system)
  - Dynamic (i.e., dependent on history of previous states).

- We show how to **implement** our error model so that it becomes **executable** in our fault injection tool, Hovac [https://github.com/laena/hovac](https://github.com/laena/hovac).