Verification of Windows NT kernel drivers using Zing model checker

Tomáš Matoušek

http://tm.matfyz.cz

CHARLES UNIVERSITY IN PRAGUE
Faculty of Mathematics and Physics

Outline

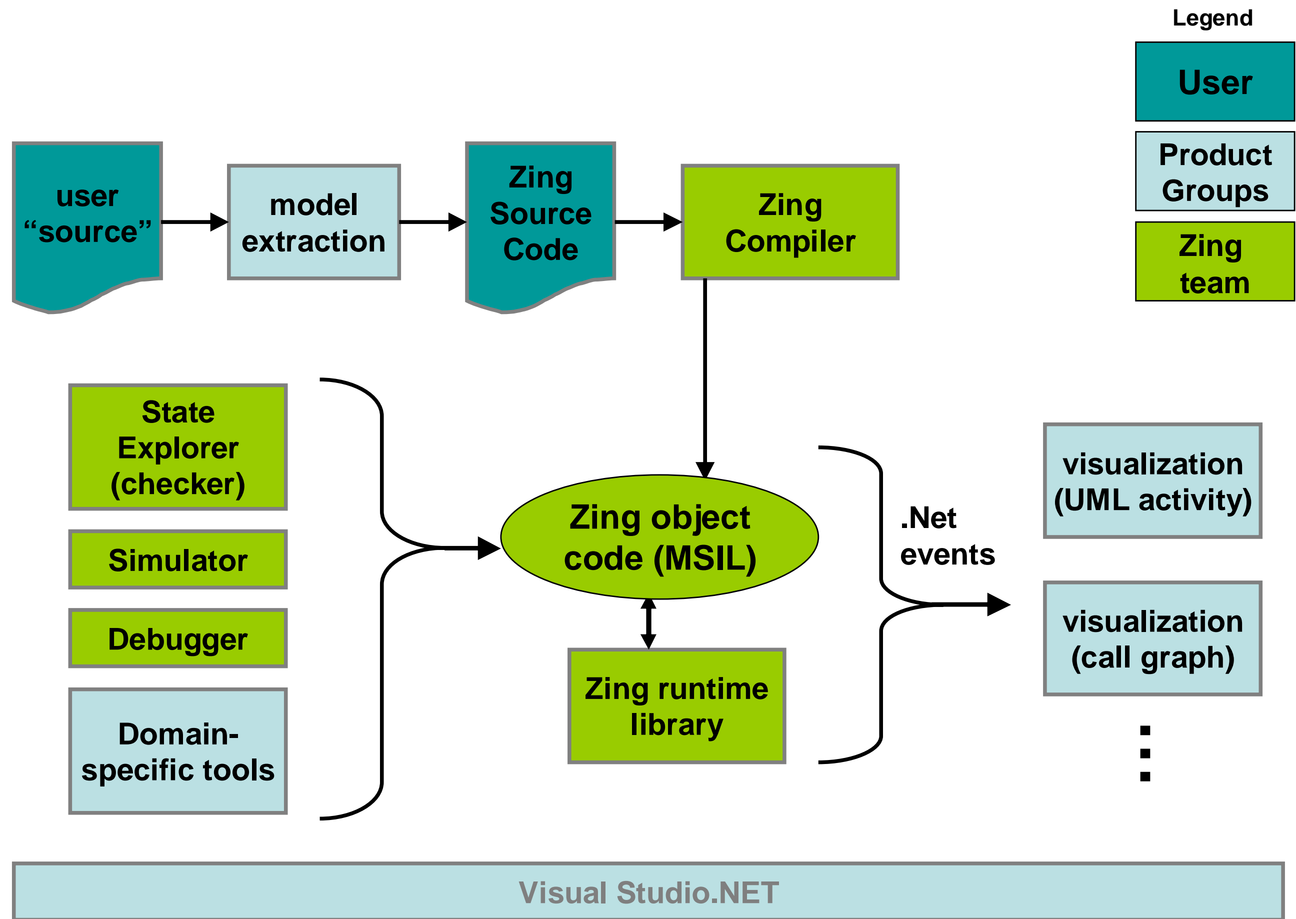
- Zing Model Checking System
- Example: Dinning Philosophers
- Kernel Driver Environment
- Driver Environment Specification Language
- Demo: Verification of IRP handling in Serial Port Driver

Overview

- MSR group
  - Tony Andrews, Sriram Rajamani, Shaz Qadeer, Jakob Rehof

- built on .NET Framework
- Zing Modeling Language
  - basics similar to Promela
  - supports common procedural language constructs
    - classes, methods, exceptions, dynamic memory allocation
  - goal: to make it easier to extract models from the code
- extractors for common languages (C#, C++, MSIL)
  - not available today, work in progress
- model checker
  - checks assertions only
  - temporal logics not supported yet

Zing Architecture



Features

- control flow
  - loops: while, foreach
  - methods
  - exceptions
- concurrency
  - asynchronous call – process creation
  - static fields – shared memory
  - channels – communication via send/receive
- data
  - value types
  - reference types (heap)
    - no inheritance
    - sets, arrays, objects
- model checking specific features
  - atomic brackets
  - non-deterministic choices
  - assertions, assumptions
  - traces, events, annotated statements

Types

- simple (values)
  - predefined: bool, byte, int
  - enum
  - structure
- complex (references)
  - array
    - currently fixed size only
  - set
    - unbounded unordered collection of distinct items
  - channel
    - unbounded queue
  - class
    - no inheritance, overloading, ctors, access modifiers, nesting
  - generic **object** type
    - any complex type placeholder

State Data

- unbounded heap
- unbounded per-process stacks
- global data (static fields)

Processes

- entry points
  - static parameter-less methods
  - marked with **activate** keyword
- new process creation
  - asynchronous calls (**async** keyword before the call)
  - methods with input parameters only

Select Statement

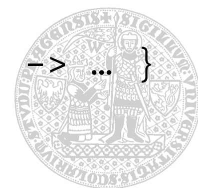
```
select_stmt ::= 'select' qualifier* '{' (join_stmt_list '->' block)+ '}'
```

- basic qualifiers
  - first
    - selects the first satisfied join statement to execution
    - non-deterministic choice among satisfied statements otherwise
  - end
    - makes the select statement a correct end state

- join list
  - a list of join statements connected by '&&' operator

- basic join statements
  - `wait(condition)`
    - blocks until the condition is satisfied
  - `receive(channel, data)`
    - blocks until the channel receives some data

```
select { wait(sizeof(ch) >= 2) && receive(ch, m1) && receive(ch, m2) -> … }
```



Non-determinism

- select without 'first' qualifier
  - choice among satisfied join statements

- choice
  - choose(bool)
    - true-false choice
  - choose(*enum*)
    - choice among the items of enum
  - choose(*low*..*high*)
    - choice from an integer range
  - choose(*variable*)
    - choice among items contained in the variable of type array or set

Example

- Dinning Philosophers

Verification of Kernel Drivers Using Zing

- inputs
  - source code of the kernel driver
  - specification of the kernel environment
    - kernel abstraction
    - properties imposed on drivers
- process
  - C code analysis and slicing driven by the specification
  - C to Zing conversion
  - combination with predefined Zing partial model
    - particular partial model chosen according to the property being checked
  - slicing on the Zing model
- output
  - a Zing model realizing an interaction of the driver with the environment related to the property being checked

Windows Kernel Driver

- a module operating in kernel mode

- provides interface
  - initialization: DriverEntry
  - work
    - for each connected device
      - AddDevice
      - asynchronously in more or less non-deterministic order: IrpDispatch, ISR, DPC, APC, IRP completion routines, routines synchronized with ISRs, …
      - IrpDispatch(RemoveDevice)
  - finalization: Unload

  - not all routines have to be implemented by the driver
  - modeled by partial models
    - e.g. sequential model – provided routines are called synchronously

- requires interface
  - kernel API functions (KeXxx, IoXxx, MmXxx, PoXxx, …)

Rules

- drivers should comply with many rules
  - stated in Driver Development Kit (DDK)
  - a few of them checked by Static Driver Verifier (SDV)
  - English description

- rules on using kernel functions
  - “If a driver specifies a Tag on a call to IoAcquireRemoveLock, the driver must supply the same Tag in the corresponding call to IoReleaseRemoveLock.”
- rules on routines of provides interface
  - “The DriverEntry routine should save a copy of the registry path string, not the pointer itself, since the pointer is no longer valid after DriverEntry returns.”

- we need a formal description
  - of the rules
  - of the kernel environment

Driver Environment Specification Language (DeSpec)

- an object oriented specification language
  - inspired by Spec#
- type abstractions
  - abstractions of C types used by drivers
- temporal logic patterns
  - defined by Bandera
    - e.g. {P} leads to {Q} <scope pattern>
  - several new added
    - e.g. {P} corresponds to {Q} <scope pattern>
  - temporal logic itself not supported
    - patterns are expressive enough
    - can be added easily
    - but it would make specifications less readable

Driver Source Code

- drivers written in C
- kernel functions are “object oriented”
  - groups of functions working on the same structure
- C headers

```
void IoInitializeRemoveLock(IO_REMOVE_LOCK *Lock, ULONG AllocateTag,
                            ULONG MaxLockedMinutes, ULONG HighWatermark);
NTSTATUS IoAcquireRemoveLock(IO_REMOVE_LOCK *Lock, void *Tag);
void IoReleaseRemoveLock(IO_REMOVE_LOCK *Lock, void *Tag);
void IoReleaseRemoveLockAndWait(IO_REMOVE_LOCK *Lock, void *Tag);
```

- DeSpec

```
class IO_REMOVE_LOCK
{
  void IoInitializeRemoveLock(instance, ...);
  NTSTATUS IoAcquireRemoveLock(instance, object tag);
  void IoReleaseRemoveLock(instance, object tag);
  void IoReleaseRemoveLockAndWait(instance, object tag);
}
```

ellipsis stands for arguments which are not interesting now

defines the parameter treated as “this”

`void*` is replaced with `object` which means the tag have to point on some allocated object not on a field etc.

Pointers and Macros

- pointer operations are unverifiable in general
- however
  - interesting code doesn't use pointer arithmetic at all or use predefined kernel macros for "standard" pointer operations
  - uninteresting code can be sliced out
  - structures allocated on the stack can be moved to the heap
  - fields pointed to by interesting pointer can be boxed
  - if a pointer can neither be converted to an object nor be sliced out

    $\Rightarrow$ the property being checked is declared unverifiable

- some kernel "functions" are, in fact, macros
  - shouldn't be expanded to check rules related to them
  - we need a special preprocessor which will preserve them
    - or pre-preprocessor replacing declarations in header files
  - such macros are specified as normal methods in DeSpec

Rules

- DDK says
  - “A driver must initialize a remove lock with a call to IoInitializeRemoveLock before using the lock.”
- DeSpec says

```
class IO_REMOVE_LOCK
{
  void IoInitializeRemoveLock(instance, ...);
  NTSTATUS IoAcquireRemoveLock(instance, ...);
  void IoReleaseRemoveLock(instance, ...);
  void IoReleaseRemoveLockAndWait(instance, ...);

  rule
    { IoInitializeRemoveLock() }
    precedes
    { any() }
    globally;
}
```

“this” is quantified implicitly by universal quantifier since the rule is an instance rule

f(args) in a rule matches a function call in the source code; the value of f(args) expression is true iff the function has been called

any stands for any method declared in the abstraction. Equivalent to:

```
{ IoInitializeRemoveLock() || IoAcquireRemoveLock() ||
IoReleaseRemoveLock() || IoReleaseRemoveLockAndWait() }
```

Source Code Events

- express particular actions being performed by the code
  - method entry `M(args)::entered`
  - method return `M(args)::returned`, or `M(args)` for short
  - successful return `M(args)::succeeded`
  - method failure `M(args)::failed`
  - field reading `F::read`
  - field writing `F::written`
- accessing method result
  - with result propagation `a = M(args)::result`
  - without result propagation `a := M(args)`
- implementation in the model
  - method and field state variables added

Example

DDK:

"Drivers that manage their own queues of IRPs should check Cancel routine pointer that IoSetCancelRoutine returns to determine whether the cancel routine has already started."

```
class IRP
{
  FunctionPointer IoSetCancelRoutine(instance, ...);

  [Conditional(!Driver.UsesKernelIrpQueue)]
  rule
    forall (FunctionPointer routine)
    {
      IoSetCancelRoutine()
    }
    leads to
    {
      // binds "routine" to a source code variable and requires
      // this variable to be read afterwards;
      // if the variable is not bound the condition is not satisfied:
      (routine := IoSetCancelRoutine()) && routine::read
    }
    globally;
}
```

an attribute which makes checking of the rule dependent on a condition

rule state variable; recall: "this" is quantified implicitly

the value of this expression is true iff the function has returned

Method Contracts

- preconditions
  - conditions under which the method can be called
  - checked by an assertion in the resulting model
- postconditions
  - conditions which holds when the method returns
  - modeled by
    - assumption if the method specifies a kernel function
    - assertion if the method specifies a function of provides iface
- blocking pre/post conditions
  - process is blocked until the condition is satisfied
- argument and return type abstractions
  - non-nullable arguments, return values
  - parts of precondition and postcondition respectively

Method Body

- method body
  - can restrict method output values
  - can model some more complex property explicitly
  - can model the real behavior of the method

```
class FAST_MUTEX
{
  synthetic KIRQL OldIrql;
  synthetic KTHREAD Owner;

  void ExInitializeFastMutex(instance)
  {
    Owner = null;
  }

  void ExAcquireFastMutex(instance)
    requires Owner==null otherwise wait;
  {
    oldIrql = cpu.Irql;
    Owner = cpu.Thread;
    cpu.Irql = KIRQL.APC_LEVEL;
  }
  …
}
```

synthetic fields are used in the resulting model but don't correspond to fields specified in a source code

cpu refers to variables bound to the current cpu (such as Irql and Thread).

blocking precondition

KIRQL is an enum abstracting kernel type of the same name containing levels of IRQL.

Kernel Method Model

```
return_type Method(args)
{
  assert(precondition);

  atomic
  {
    select { wait (blocking_precondition) -> ; }

    Method_state = entered;

    // used by old(var) opearator in body or postcondition
    old_var1 = var1;
    old_var2 = var2;
    …

    // outputs which are not assigned in the method body before read
    // are chosen non-deterministically according to their abstractions
    result = choose(abstraction);
    out_param1 = choose(abstraction);
    out_param2 = choose(abstraction);
    …

    <method body>
  }
  atomic
  {
    select { wait (blocking_postcondition) -> ; }

    Method_state = IsSuccessful(result) ? succeeded : failed;

    assume(postcondition);
    return result;
  }
}
```

Driver Method Model

```
return_type Method(args)
{
  Method_state = entered;

  // used by old(var) opearator in a body or a postcondition
  old_var1 = var1;
  old_var2 = var2;
  …

  // a model of the method body extracted from C code
  <method body>

  assert(postcondition);

  Method_state = IsSuccessful(result) ? succeeded : failed;
  return result;
}
```

notes

- preconditions are missing
  - method is called from kernel ⇒ the kernel (partial model) is responsible for preconditions
- method body is not enclosed in atomic brackets
  - driver code cannot be assumed to be well synchronized (the kernel code can)
  - in fact, driver code is verified for this property

Example

DDK:

"Device driver's StartIo routine is responsible for calling IoGetCurrentIrpStackLocation."

```
static class Driver
{
  void @StartIo(_,IRP irp);

  rule
    forall (IRP irp)
    { IoGetCurrentIrpStackLocation(irp) }
    exists between
    { @StartIo(_,irp)::entered }
    and
    { @StartIo(_,irp)::succeeded };
}
```

@ sign marks identifiers which may be named differently in the driver source code – a mapping has to be manually defined (a heuristic is also possible).

- we want to check successfulness
  - it requires use of "between" pattern scope
  - otherwise we may use simpler pattern

```
{ IoGetCurrentIrpStackLocation(irp) && @StartIo(_,irp)::entered }
exists globally;
```

Data Abstractions

- a single kernel structure or function can be abstracted by more DeSpec abstractions
  - depends on what we need to check
  - if we checked all rules together the model would be too large
- namespaces group specifications used together
  - special namespace: Default
  - top-level abstractions implicitly placed to this namespace
  - contains default abstractions
    - used for abstracting uninteresting kernel functions and structures
    - as simple as possible
- data abstractions
  - top-level: enums, ranges, classes, embedded classes
  - nested: structs, unions

Enums and Ranges

```
enum POOL_TYPE
{
    Paged = { PagedPool, PagedPoolCacheAligned }
    NonPaged = { NonPagedPool, NonPagedPoolMustSucceed,
                 NonPagedPoolCacheAligned, NonPagedPoolCacheAlignedMustS },
}

enum KIRQL
{
    PASSIVE_LEVEL = 0,
    APC_LEVEL = 1,
    DISPATCH_LEVEL = 2,
    DIRQL = 3..26,
    HIGHER_LEVELS = 27..31
}

enum NTSTATUS
{
    Success = STATUS_SUCCESS,
    Unsuccessful = others
}

enum IrpStatus abstracts NTSTATUS
{
    Success = STATUS_SUCCESS,
    Pending = STATUS_PENDING,
    Unsuccessful = others
}

range AllIrqls = 0..31;
```

abstracts from distinguishing among specified values

abstracts from distinguishing among values not stated

if we want to name an abstraction differently from the abstracted type we use abstracts keyword to link the abstraction to the type

defines a range of admissible values

Structures, Unions, Properties

- structure
  - only for qualified name mapping (e.g. `irp->IoStatus.Information`)
  - flattened: fields moved to owning class
- union
  - a structure with hidden discriminator set after a write op. and asserted before a read op.
- property
  - allows to add pre/post conditions and modeling code to field read/write operations

```
class IRP
{
  …
  union AssociatedIrp
  {
    object! SystemBuffer { get; }
    IRP! MasterIrp { get; }
  }

  struct IoStatus
  {
    NTSTATUS Status;
    uint Information;
  }

  KPROCESSOR_MODE RequestorMode { get; }
  …
```

```
  …
  union Tail
  {
    union Anonymous
    {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
    }

    ETHREAD Thread
    {
      set requires Driver.IsHigher;
      get;
    }
    …
  }
  …
}
```

Embedded Classes

- enforced by macro performing a “standard” pointer operation

```
CONTAINING_RECORD(field_address,owning_type,field_name)
```

- returns the object whose field address is specified
- used for implementation of linked lists (for example)
- C
  - linking pointers embedded into the owning class
- DeSpec
  - fields separated into “embedded class” – a class with an owner
  - owning class sets its ownership up in its synthetic constructor (like Java inner class)

```
class IRP
{
   … union Tail { … struct Overlay { LINK_ENTRY ListEntry; } … } …

   synthetic IRP() { Tail.Overlay.ListEntry = new LINK_ENTRY(this); }
}

embedded class LIST_ENTRY
{
  LIST_ENTRY Flink,Blink;
  …
  synthetic LIST_ENTRY(object instanceOwner) { owner = instanceOwner; }
}
```

Demo and Examples

- demo
  - verifying IRP handling in Serial Port driver
- examples
  - Irp.des
  - RemoveLock.des
  - Lists.des

Conclusion

- DeSpec seems to be expressive enough to describe a majority of rules stated in DDK
  - including SDV rules
  - approx. a third of them already written in DeSpec
- Zing doesn’t support temporal logic yet
  - many patterns can be verified using assertions only
  - Is there any theoretical limitation which would disallow implementation of never claim in Zing?
- major issues
  - C code analysis
  - model extraction
  - slicing

References

Zing Language Specification, Zing User Guide
http://research.microsoft.com/zing

Windows Driver Development Kit
http://www.microsoft.com/whdc/devtools/ddk/default.mspx

Spec# Programming System
http://research.microsoft.com/specsharp

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, Yichen Xie
Zing: A Model Checker for Concurrent Software.
Technical Report, MSR-TR-2004-10, January, 2004.

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, Yichen Xie
Zing: Exploiting Program Structure for Model Checking Concurrent Software
CONCUR 2004

M. Barnett, K. R. M. Leino, and W. Schulte.
The Spec# Programming System: An Overview.
Microsoft Research, May 2004.