Zing: A Model Checker for Concurrent Software

Tony Andrews[*] Shaz Qadeer[*] Sriram K. Rajamani[*]
Jakob Rehof[*] Yichen Xie[†]

January 27, 2004

Technical Report
MSR-TR-2004-10

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

The ZING project is an effort to build a flexible and scalable model checking infrastructure for concurrent software. The project is divided into four components: (1) a modeling language for expressing concurrent models of software systems, (2) a compiler for translating a ZING model into an executable representation of its transition relation, (3) a state explorer for exploring the state space of the ZING model, and (4) model generators that automatically extract ZING models from programs written in common programming languages. We believe that such an infrastructure is useful for finding bugs in software at various levels: high-level protocol descriptions, work-flow specifications, web services, device drivers, and protocols in the core of the operating system. The next section illustrates ZING using an example. In the remainder of this paper, we give an overview of the modular software architecture of Zing, the novel algorithms implemented in the Zing state explorer, and our ongoing efforts to automatically generate Zing models from C and C# programs.

2 Example

We illustrate using ZING to find a concurrency error in a Windows device driver. The driver under consideration, like every Windows driver, has to handle races between PNP Stop and other dispatch routines that handle the IRP's of the driver. The driver code should satisfy the invariant that during the execution of a dispatch routine, the PNP Stop routine should not be allowed to stop the device. It turns out that this invariant is violated in one particular interleaving of events between the dispatch routine and the PNP Stop routine. PNP stands for *plug-and-play*, the feature that allows devices to attach and detach when the OS is running, and IRP stands for *I/O Request Packet* which are units by which the OS communicates I/O operations to the device driver.

The details of the synchronization are shown in the ZING model in Figure 1. The synchronization between the threads is quite complicated. It is implemented using three objects: a boolean flag called `driverStoppingFlag` in the device extension, an event called `stoppingEvent`, and a reference count field `pendingIo` in the device extension. The reference count `pendingIo` is initialized to 1, and the `stoppingFlag` is set to false during driver entry. The dispatch routine first checks the `driverStoppingFlag` to see if the PNP Stop is imminently going to stop the device. If `driverStoppingFlag` is true then it fails the IRP. If `driverStoppingFlag` is false then it increments the reference count `pendingIo`, does whatever work needs to be done by the dispatch routine, and then decrements the reference count. After decrementing the reference count, if the reference count reaches 0, then it signals the event `stoppingEvent`. The PNP Stop routine sets `driverStoppingFlag` to true and decrements the reference count `pendingIo`. After decrementing the reference count, if the reference count reaches 0, then its signals the event `stoppingEvent`. Finally, it waits for the event `stoppingEvent` and after the event arrives, it stops the device.

```
static void PNP_Stop (DEVICE_OBJECT BtDevice){
  DEVICE_EXTENSION deviceExtension = BtDevice.deviceExtension;
  deviceExtension.driverStoppingFlag = true;
  IoDecrement(deviceExtension);
  KeWaitForStoppingEvent(deviceExtension);
  deviceExtension.stopped = true;
}
static void Dispatch(DEVICE_OBJECT BtDevice){
  DEVICE_EXTENSION deviceExtension =  BtDevice.deviceExtension;
  int status;
  status = IoIncrement (deviceExtension);
  if(status > 0){
     // do work here
     assert(!deviceExtension.stopped);
  }
  BCSP_IoDecrement(deviceExtension);
}
static int IoIncrement(DEVICE_EXTENSION e){
  int status;
  bool driverStopping = e.driverStoppingFlag;
  if(driverStopping == true) status = -1;
  else{
     InterlockedIncrementPendingIo(e);
     status = 1;
  }
  return(status);
}
static void IoDecrement(DEVICE_EXTENSION e){
  int pendingIo;
  pendingIo = InterlockedDecrementPendingIo(e);
  if(pendingIo == 0) KeSetEventStoppingEvent(e);
}
```

Fig. 1. A ZING model of a device driver.

An invariant we want the driver code to have is that while a dispatch routine is doing work, the PNP Stop routine should not be allowed to stop the device. The invariant is encoded using an assertion in the ZING model. Suppose the dispatch thread has just finished checking the `driverStoppingFlag` and ascertained that it is false. Just before it increments the reference count, if the PNP Stop IRP gets fired and it sets the `driverStoppingFlag` to true, decrements the reference count to 0, and stops the device. In this particular interleaving, the dispatch thread continues execution after incrementing the reference count, unaware that the device has been already stopped. The ZING state explorer is able to produce this exact interleaving automatically from analyzing the ZING model for the driver. The ZING browser shows this error trace just like a debugger, allowing the user to step through all the events that led to the assertion violation.

The developer proposed the following fix for this problem: Change the dispatch routine to first increment the reference count and then check the `driverStoppingFlag` (and decrement the reference count back, if the flag is set). ZING was able to validate the fix by proving that the desired invariant holds in all possible interleavings between the dispatch routine and the PNP Stop IRP.

3 The ZING modeling language and compiler

ZING provides several features to support automatic generation of models from programs written in common programming languages. ZING supports a basic asynchronous interleaving model of concurrency with both shared memory and message passing. In addition to sequential flow, branching and iteration, ZING supports function calls and exception handling. New threads are created via asynchronous function calls. An asynchronous call returns to the caller immediately, and the callee runs as a fresh process in parallel with the caller. Primitive and reference types, and an object model similar to C# or Java is supported, although inheritance is not supported. ZING also provides features to support abstraction and efficient state exploration. Any sequence of statements (with some restrictions) can be bracketed as atomic. This is essentially a directive to the state explorer to not consider interleavings with other threads while any given thread executes an atomic sequence. Sets are supported, to represent collections where the ordering of objects is not important (thus reducing the number of potentially distinct states Zing needs to explore). A choose construct that can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, or object references is provided. An example ZING model that we extracted from a device driver, and details of an error trace that the ZING explorer found in the model can be found in our technical report [1].

The ZING compiler translates a ZING model into a ZING object model, which can be executed to produce transitions between ZING states. A ZING state has the following three components:

- **Stacks**. Each thread in the ZING runtime state has its own stack and each stack consists of stack frames where the local variables and return addresses are being stored.
- **Global storage**. Static class variables are stored in a shared global storage area. The number of global variables and their types are determined at compile time.
- **Heap**. ZING supports dynamic allocation of objects. Dynamically allocated objects are placed on the heap, which is implemented by a variable sized array of heap objects indexed by pointers.

A ZING model is compiled into an MSIL assembly that manipulates the state representation according to the semantics of the model. The MSIL assembly can be loaded and invoked from within the state explorer. In a ZING model, process stacks and heap can grow without limit at runtime due to recursive function calls and dynamic allocation. This flexibility comes at a price: ZING is not guaranteed to terminate on every model. However, we can still systematically explore a portion of the state space and look for errors.

4 ZING state explorer

The ZING state explorer executes the ZING object model to explore the state space of the corresponding ZING model. Starting from the initial state, the state explorer systematically explores reachable states in a depth-first manner. State transitions are carried out by invoking appropriate methods in the MSIL assembly produced by the ZING compiler. The biggest technical challenge with ZING, as with any model checker, is scalability. We have implemented several techniques that reduce the time and space required for state exploration.

Efficient state representation. We observe that most state transitions modify only a small portion of the ZING state. By only recording the difference between transitions, we greatly cut down the space and time required to maintain the depth-first search stack. To further cut down on the space requirements, the state explorer stores only a fingerprint of an explored state in its hash table. We use Rabin's finger-printing algorithm [3] to compute fingerprints efficiently.

Symmetry reduction. A ZING state comprises the thread stacks, the global variables, and a heap of dynamically allocated objects. Two states are equivaluent if the contents of the thread stacks and global variables are *identical* and the heaps are *isomorphic*. When the state explorer discovers a new state, it first constructs a canonical representation of the state by traversing the heap in a deterministic order. It then stores a fingerprint of this canonical representation in the hash table.

Partial-order reduction. We have implemented a state-reduction algorithm that has the potential to reduce the number of explored states exponentially without missing errors. This algorithm is based on Lipton's theory of reduction [8]. Our algorithm is based on the insight that in well-synchronized programs, any computation of a thread can be viewed as a sequence of transactions, each of which appears to execute atomically to other threads. During state exploration, it is sufficient to schedule threads only at transaction boundaries. If programmers follow the discipline of protecting each shared variable with a lock, then these transactions can be inferred automatically [4]. These inferred transactions reduce the number of interleavings to be explored, and thereby greatly alleviate the problem of state explosion.

Summarization. The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. This is not the case for concurrent programs. ZING has an implementation of a novel model checking algorithm for concurrent programs that uses procedure summarization as an essential component [9]. Our method for procedure summarization is based on the insight about transactions mentioned earlier. We summarize within each transaction; the summary of a procedure comprises the summaries of all transactions within the procedure. The procedure summaries computed by our algorithm allow reuse of analysis results across different call sites in a concurrent program, a benefit that has hitherto been available only to sequential programs.

Compositional reasoning. Stuck-freedom is an important property of distributed message-passing applications [10, 5]. This property formalizes the requirement that a process in a communicating system should not wait indefinitely for a message that is never sent, or send a message that is never received. To enable compositional verification of stuck-freedom, we have defined a conformance relation $\leq$ on processes with the following substitutability property: If $I \leq C$ and P is any environment such that the parallel composition $P \mid C$ is stuck-free, then $P \mid I$ is stuck-free as well. Substitutability enables a component's specification to be used instead of the component in invocation contexts, and hence enables model checking to scale. We have adapted the ZING state explorer to implement a *conformance checker* to verify the relation $I \leq C$, where I and C are ZING models.

5 Automatic model generation

ZING is designed as a backend to model-generation tools that automatically extract behavioral models from concurrent systems written in common programming languages. Abstraction can offer huge reductions in state space needed to be explored. As an example, in the SLAM project [2], when we check if an acquired lock is eventually released, we can typically construct an abstraction that involves just the device drivers control flow graph and one bit of information as to whether the lock is held or not. This abstraction has a state space linear in the size of the driver, and hence can be used to check for proper lock usage on very large device drivers. We are currently buidling several automatic extractors from common programming languages, and we are investigating whether abstractions can be generated using iterative refinement for concurrent programs using ZING.

6 Related work

Software model checking is an active area of current research. ZING is an explicit-state model checker, in the spirit of SPIN [7], JPF [13] and BOGOR [11]. In comparison with SPIN, ZING supports several features like objects and function calls, which make it a more amenable target for automatic extraction of models from programming languages. In comparison with JPF and BOGOR, we implement newer algorithms for state-space reduction, such as reduction and summarization. Like BOGOR, one of our design goals is to keep the architecture flexible and open. Unlike any of these related efforts, we support a notion of conformance between two ZING models [5]. This feature of ZING is related to, but distict from, the refinement checking feature of the FDR model checker [12, 6].

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. Technical report, Microsoft Research, 2004.
2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
3. A. Broder. Some applications of Rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152, 1993.
4. C. Flanagan and S. Qadeer. Transactions for software model checking. In *SoftMC 03: Software Model Checking Workshop*, 2003.
5. C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck free conformance. Technical report, Microsoft Research, 2004.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
7. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
8. R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.
9. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255. ACM, 2004.
10. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 166–179. Springer-Verlag, 2002.
11. Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
12. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
13. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICASE 00: Automated Software Engineering*, pages 3–12, 2000.