Model checking applied to practical systems

Ravish Khosla
Dependable Computing Systems Lab
Purdue University

Outline

- Introduction to model checking
- Model checking: Practical Applications
    - Today: Using Model Checking to debug Device Firmware
    - Next Week: Other applications
        - Debug File System Errors
        - Securing e-commerce transactions

PURDUE UNIVERSITY

Introduction to Model Checking, [1]

- Evolved as a successful technology to verify requirements and design for a variety of real-time embedded and safety-critical systems
- Checking design requirements before software development
    - Questions to ask?

Introduction to Model Checking (Contd.)

- Questions to ask:
  - Do they accurately reflect the users' requirements?
  - Are the requirements clearly written, unambiguous and understandable?
  - Are they flexible and realizable for the engineers? For instance, are the requirements suitably modular and well structured to aid in design and development?
  - Can the requirements be used to easily define acceptance test cases to check the conformance of the implementation against the requirements?

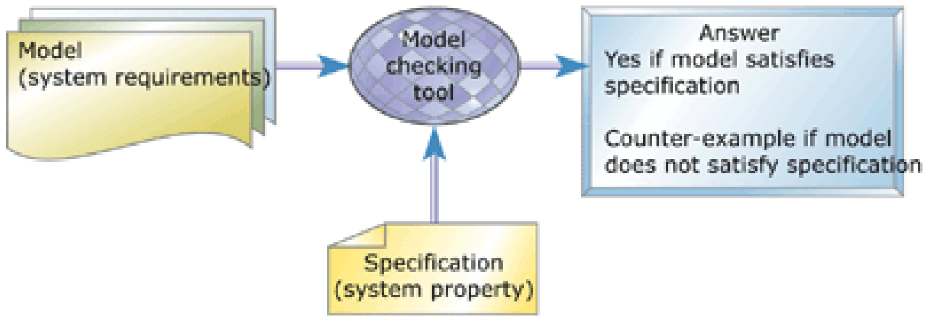
Introduction to Model Checking (Contd.)

- Questions to ask (Contd.)
  - Are the requirements written in an abstract and high-level manner, away from design, implementation, technology platforms and so on, so as to give enough freedom to the designer and developers to implement them efficiently?
- Errors in system requirements
  - Cost is often high, requiring at least rework and maintenance
  - Implementation can lead to incorrect system behavior

Introduction to Model Checking (Contd.)

- Solution
  - Use automated tools to check the quality of various aspects of the requirements and design
  - Enforce a clear, rigorous, and unambiguous formal language
  - Model-checking

Model Checking Approach


Model
(system requirements)
Model checking tool
Answer
Yes if model satisfies specification
Counter-example if model does not satisfy specification
Specification
(system property)

- Inputs
  - System requirement or design (Models)
  - Specifications (Property that the final system is expected to satisfy)
- Outputs
  - Yes if model satisfies specifications
  - Counterexample otherwise: Helps to find errors

How do we define models?

- No single answer for what formal language to use
- Most systems considered are control-oriented rather than data oriented
  - Dynamic behavior is much more important than the structure of and operations on the internal data maintained by the system
  - Finite State Machines (FSM's) is a good notation for defining requirements and design

Model Checking: First application

- S. Kumar and K. Li. Using model checking to debug device firmware. In 5th Symp on Operating Systems Design and Impl. 2002.

Device Firmware

- Device firmware has to be reliable because
  - It is trusted by the operating system
  - It has the ability to write directly into thevphysical memory
  - A stray memory write resulting from a bug can corrupt critical data structures in the operating system and crash the entire machine.

Challenges in developing reliable Device Firmware

- The firmware is implemented using concurrency
  - They might have unforeseen interactions between the different sequential flows of control resulting in race conditions.
- Event-driven state machines are used to express the concurrency
  - Because of their low performance overhead
  - Programming with event-driven state machines in languages like C is difficult because they are not designed to support event-driven state-machines programming
  - Bugs may be introduced by the programmer

Challenges in developing reliable Device Firmware (Contd.)

- Very limited debugging support
  - Often, it is limited to a few memory locations to which the device can write
- Firmware referred in this paper
  - Virtual Memory-Mapped Communication (VMMC) firmware for Myrinet network interface cards
  - It delivers high performance on gigabit networks by migrating as much functionality as possible from the operating system to the network interface card

VMMC Firmware

- Experience of the authors with implementing the VMMC firmware in C
  - While good performance could be achieved, the source code was difficult to write, maintain, and debug
  - Bugs due to race condition remain and occasionally system crashes

Model Checking as an alternative

- Model Checking
  - Seems a promising approach for developing reliable firmware as it can explore all interleaved executions of the concurrent firmware's state machines and checks if the property being verified holds

Model Checking: Some Definitions

- *Global State*
  - It is a snapshot of the entire system at a particular point in execution
- *State space*
  - It is the set of all the global states reachable from the initial global state.
- Since the state space of such systems is finite, the model checkers can, in principle, exhaustively explore the entire state space.

Properties that can be verified

- *Safety* Properties
  - Safety properties are properties that have to be satisfied in specific global states of the system
  - For Example, assertion checking and Deadlocks (A deadlock situation corresponds to the set of all the global states that do not have a valid next state)
- Liveness Properties
  - They refer to sequence of states and are usually specified using temporal logic
  - For example, absence of livelocks

Model Checking

- Advantage
  - Automatic. Violation of property helps in finding the bug
- Disadvantages
  - The state space to be explored is exponential in the number of processes and the amount of memory used
  - Second, the specification language supported by the model checkers provides limited functionality.
  - So, it is not straightforward to translate concurrent programs written in traditional programming languages into the specification language of the model checkers.

Solutions

- Abstraction
  - A model that captures only the details relevant to that property needs to be extracted (especially in large systems)
  - The extraction should preferably be done automatically

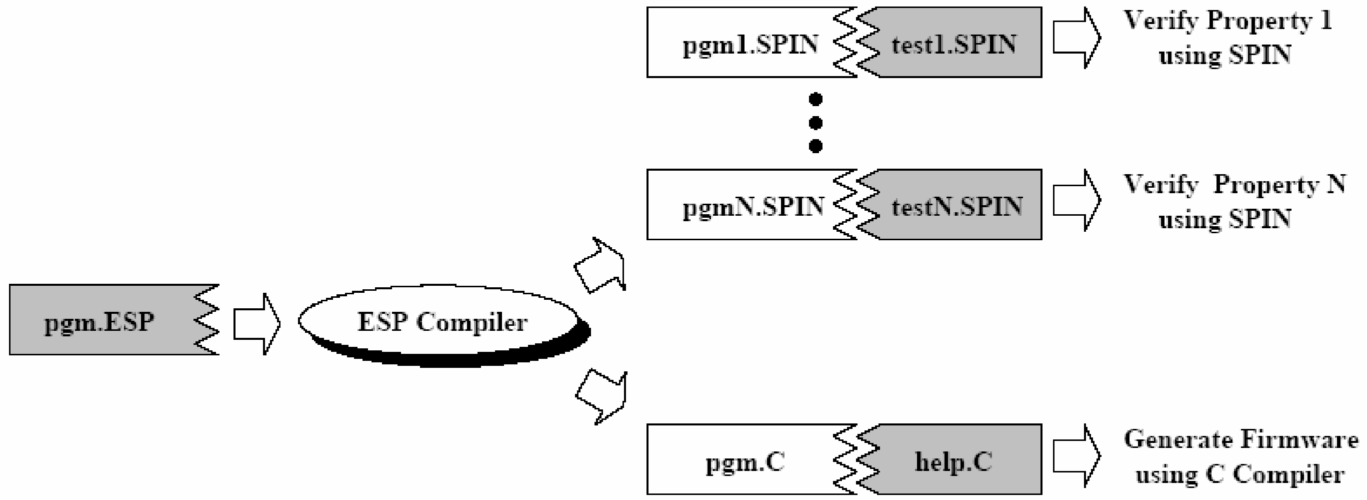
Approach

- Event-driven State-machine Programming (ESP)
    - Language with C-style syntax for programming devices
    - The basic components of the language are processes and channels.
    - Each process represents a sequential flow of control in a concurrent program and communicates with other processes using rendezvous channels.

Goals of ESP

- It should provide language support that makes it easier to develop device firmware.
- Second, it should allow the use of model checkers like Spin to extensively test and debug the firmware
- Third, the compiler should be able to generate efficient executables to run a single processor.

ESP Approach



pgm1.SPIN
test1.SPIN
Verify Property 1 using SPIN
pgmN.SPIN
testN.SPIN
Verify Property N using SPIN
pgm.ESP
ESP Compiler
pgm.C
help.C
Generate Firmware using C Compiler

Figure 1: The ESP approach. The shaded regions represent code that has to be provided by the programmer.

- Programmer supplied SPIN code: Generates external events such as network message arrival as well as specifies the properties to be verified

ESP Approach: Difference with previous Approaches

- Domain Specific language
  - Any detail in the ESP program that is necessary to check a particular property can be retained in the extracted model. In contrast, general-purpose languages like C, C++, and Java have language features (complex pointer manipulation, exceptions etc.) that are difficult to translate into the specification language of the model checker

ESP Approach: Difference with previous Approaches (Contd.)

- Support for abstraction
  - To avoid the state-space explosion associated with detailed models, traditional languages have been designed to encode only the control structure of the program—the data manipulation is implemented externally in C.
  - In contrast, the ESP language provides support for both control structure and data manipulation.

SPIN Model checker

- Spin is a on-the-fly model checker
  - It does not build the global state machine before it can start checking for the property to be verified. So, in cases where the state space is too big to be explored completely, it can do partial searches.
- Three modes for state-space exploration
  - Exhaustive mode:
    - The entire state space is explored in the exhaustive mode.

Modes of Spin model checker

- Bit-state Hashing Mode:
  - For larger systems, the *bit-state hashing* mode performs a partial search using significantly less memory.
  - It exploits the fact that state spaces are usually sparse and uses a hash function to obtain a much more compact representation for a state.
  - However, since the hash function can map two states onto the same hash, a part of the state space may not be explored.
  - This technique often allows very high coverage (> 98 %) while using an order of magnitude less memory.

Modes of Spin model checker

- Simulation Mode:
  - The *simulation* mode explores single execution sequence in the state space
  - A random choice is made between the possible next states at each stage
  - Uses very little memory

Extracting models using a compiler

- ESP compiler generates 3 types of models:
  - Detailed:
    - Contain all details: Often too much state to be able to perform effective state space exploration
  - Memory-Safety Models:
    - They are used to check for memory allocation bugs in the program. These models are essentially detailed models with some additional Spin code inserted to check for validity of memory accesses

Extracting models using a compiler (Contd.)

- Abstract models:
  - They omit some of the details that are irrelevant to the particular property being verified.
  - These models can have significantly smaller state than the detailed models.
  - They can be used to check for system-wide properties like absence of deadlocks.

How abstraction works

- The programmer specifies the abstractions
  - Control over the abstraction process
  - The ESP compiler uses the abstractions specified by the programmer conservatively when generating the abstract models
  - Consequently, a bug in the ESP program will always be present in the abstract model.

Specifying Abstractions

- Two types of abstractions
  - Replacing Types
  - Dropping Variables

Replacing Types

- It allows a complex type to be replaced by a much simpler type.
- This can be done either by specifying an alternative type for the variables individually or by specifying an alternative type in a type declaration.

Replacing Types (Contd.)

- Example
  - If original program contained

```
type replyT = record of {
  caller: int,
  last: bool,
  addr: int,
  size: int
}
```

  - Abstraction can be

```
replace type replyT = record of {
  caller: int,
  last: bool
}
```

  - Replacement is a Supertype: Fields get dropped

Reduction of state by Replacing Types

- Significant reduction possible. Examples:
  - Reducing the size of an array: Affects only the performance and not the correctness of the program
  - The code to implement the retransmission protocol accepts packets that are implemented as a union of the different types of packets that have to be sent.
  - The content of the packet makes little difference to the correctness of the retransmission code, hence the complex datatype representing packets can often be replaced by a simple type in the abstract model.

Dropping Variables

- Some variables that do not affect the validity of a property being checked can be dropped altogether.
- Example:
  - A table that keeps track of the mapping between virtual and physical addresses in the main memory might not be relevant when checking the firmware for deadlocks.

Extracting Abstract Models

- Phases:
  - First the compiler performs a typechecking phase during which the compiler determines a type for every expression in the original program (without taking the abstractions into account).
  - The model generator phase can apply the abstractions to each of the statements of this fully-typed program independently.

Problems

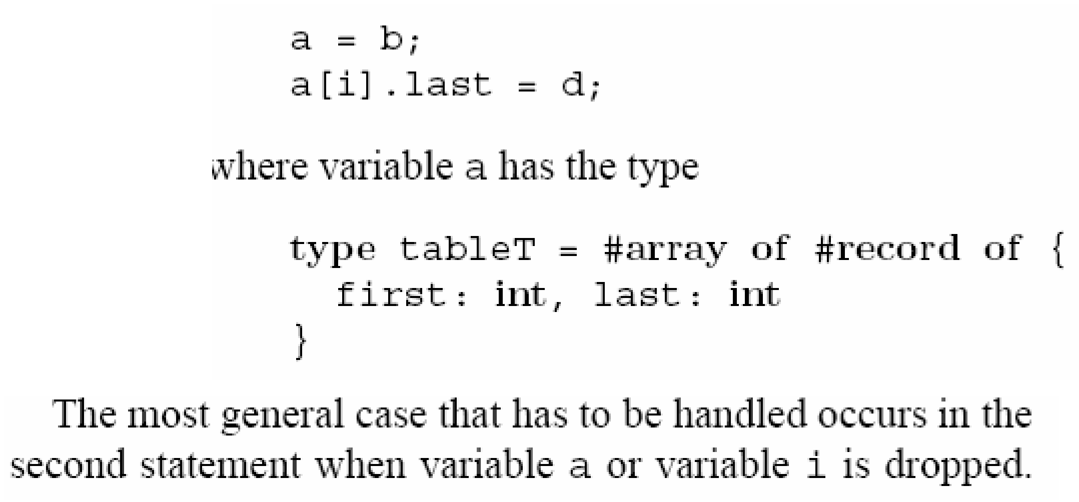
- The abstractions specified can cause some of the expressions in a statement to have an indeterminable value.
  - In these situations, the ESP compiler uses nondeterminism to make conservative approximations that strictly generalizes the scope of model checking.
- Handling of expressions
  - Left and right expressions

Left Expressions

- A left-exp is an expression that is used to determine a memory location to which a value will be stored. (In left side of assignment statements)
- In the simplest cases, when a left-exp becomes undeterminable, the statement can be simply discarded during model extraction.
  - For instance, if the variable *a* is dropped by the abstraction, the first statement *a* = *b*; becomes irrelevant and can be discarded. This is because the only side effect of that statement is to the variable *a*.

Left Expressions (contd.)

- Consider the following:



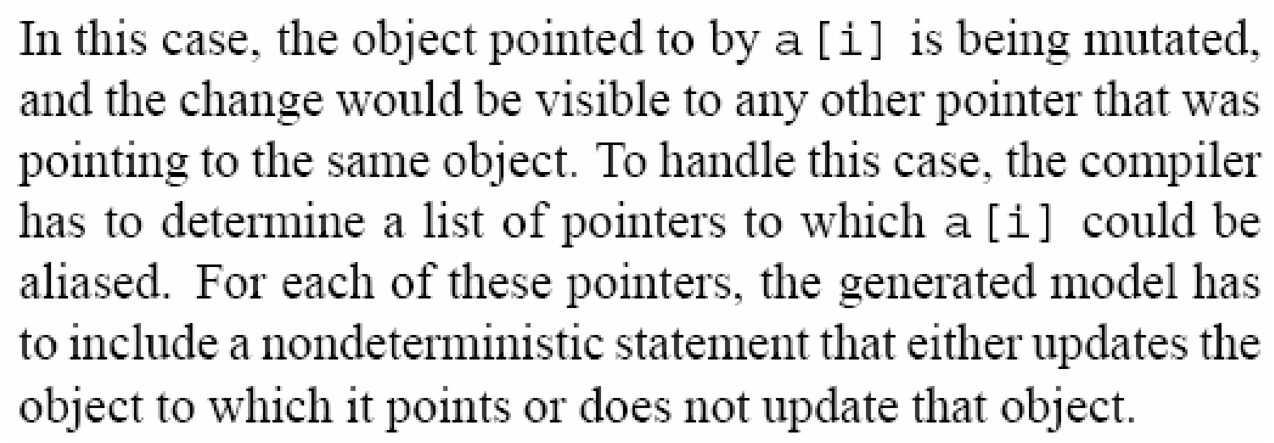
```
a = b;
a[i].last = d;
```

where variable a has the type

```
type tableT = #array of #record of {
  first: int, last: int
}
```

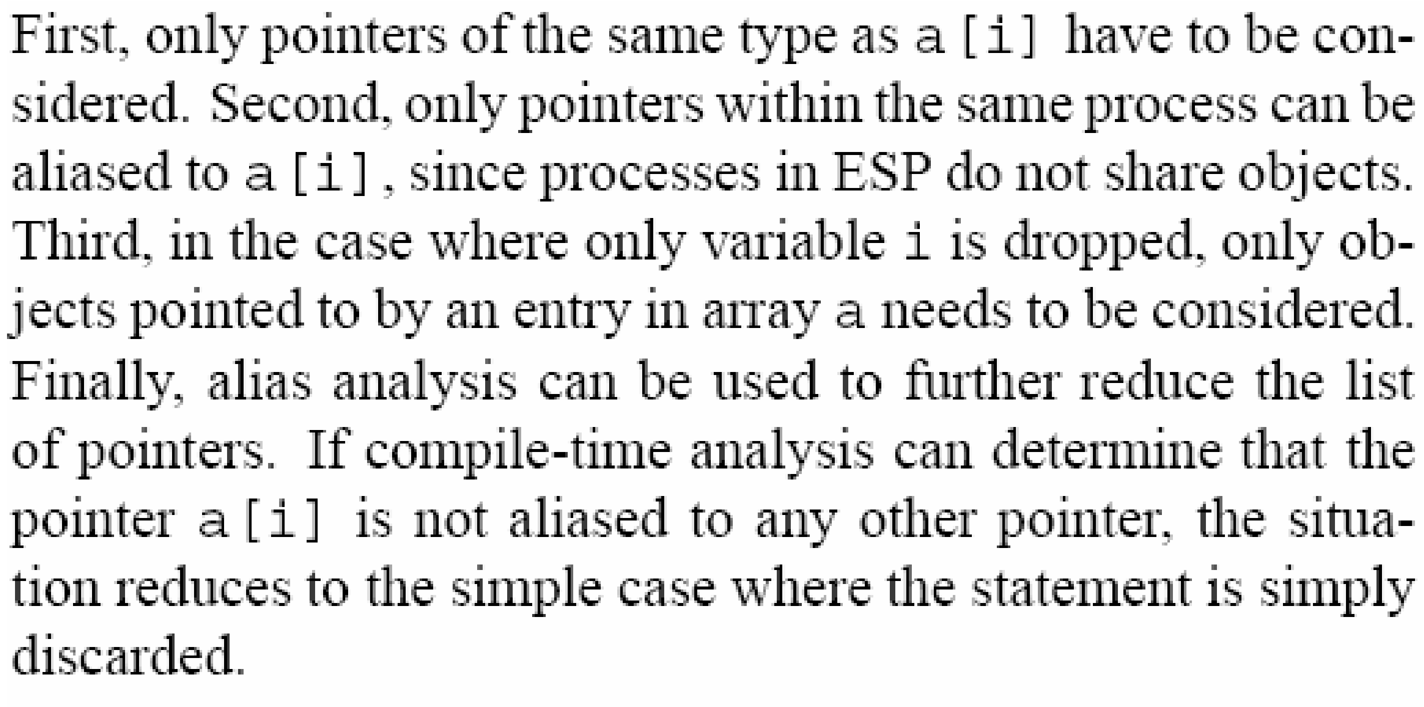
The most general case that has to be handled occurs in the second statement when variable a or variable i is dropped.

Left Expressions (contd.)


In this case, the object pointed to by `a[i]` is being mutated, and the change would be visible to any other pointer that was pointing to the same object. To handle this case, the compiler has to determine a list of pointers to which `a[i]` could be aliased. For each of these pointers, the generated model has to include a nondeterministic statement that either updates the object to which it points or does not update that object.

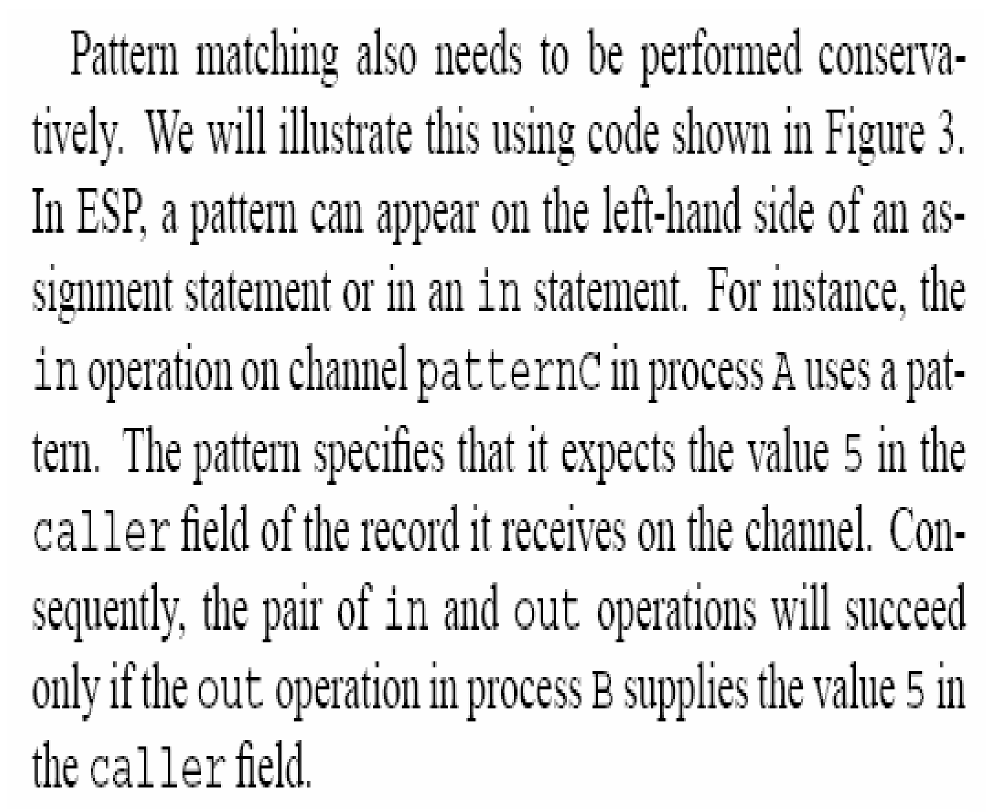
- This can dramatically increase state space
  - Techniques to narrow list of pointers to which *a*[*i*] can be aliased

Techniques to reduce state space

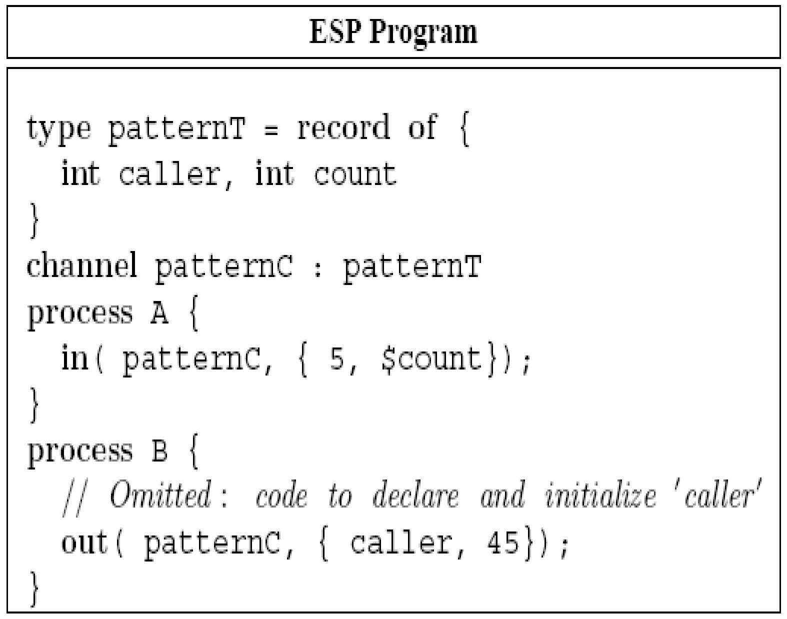


First, only pointers of the same type as `a[i]` have to be considered. Second, only pointers within the same process can be aliased to `a[i]`, since processes in ESP do not share objects. Third, in the case where only variable `i` is dropped, only objects pointed to by an entry in array `a` needs to be considered. Finally, alias analysis can be used to further reduce the list of pointers. If compile-time analysis can determine that the pointer `a[i]` is not aliased to any other pointer, the situation reduces to the simple case where the statement is simply discarded.

Another Example



Pattern matching also needs to be performed conservatively. We will illustrate this using code shown in Figure 3. In ESP, a pattern can appear on the left-hand side of an assignment statement or in an `in` statement. For instance, the `in` operation on channel `patternC` in process `A` uses a pattern. The pattern specifies that it expects the value `5` in the `caller` field of the record it receives on the channel. Consequently, the pair of `in` and `out` operations will succeed only if the `out` operation in process `B` supplies the value `5` in the `caller` field.



ESP Program

```
type patternT = record of {
  int caller, int count
}
channel patternC : patternT
process A {
  in( patternC, { 5, $count});
}
process B {
  // Omitted: code to declare and initialize 'caller'
  out( patternC, { caller, 45});
}
```

Another Example (Contd.)

Suppose the programmer uses the abstraction in Figure 3 on the ESP program in the same figure. The abstraction drops the `caller` field on the channel. In the absence of the `caller` field in the extracted model, there is no way to determine if the pattern matching on the channel should succeed. Therefore, a nondeterministic statement is inserted in process `A` before the `in` operation on the channel. The `in` operation will succeed only if the variable `match` is set to `OK`. This captures both cases: the case in which the `in` operation on channel `patternC` would have succeeded and the case in which it would not have succeeded.

Abstraction

```
replace type patternT = record of {
  int count
}
```

Extracted Spin Model

```
mtype = { OK, DONT_SEND, DONT_RECV};
typedef patternT { int nil; int count; };
chan patternC = [0] of { mtype, patternT };
proctype A() {
  mtype match; int count;
  // Nondeterministically pick a value for 'match'
  if
  :: skip -> match = OK
  :: skip -> match = DONT_RECV
  fi;
  patternC ? eval(match), 0, count
}
proctype B() {
  patternC ! OK, 0, 45
}
```

Right Expressions

- Appear on right side of assignment statements
  - Generate values to be used
  - During abstraction, the value of a right-exp expression can become undeterminable. Ideally, these expressions should be replaced by one that nondeterministically returns a valid value of the type of the expression. This will cause the model checker to try all possible valid values during the state space exploration.

Right Expressions (Contd.)

- There is no general way for the ESP compiler to determine the set of values that would be sufficient to cover the entire state space. Therefore, the ESP compiler relies on the programmer to supply the right set of values.
  - For each variable in the program (except boolean variables) for which the abstract model needs a nondeterministic value, a channel is generated in the abstract model. When a value is needed, the model performs a read operation on the channel. The programmer is responsible for supplying values on the channel using a nondeterministic statement.

Example of a non-deterministic statement

- For instance, the following code executes an infinite do loop and nondeterministically supplies either of the three values (5, 6, and 9).

```
do
:: intC ! 5
:: intC ! 6
:: intC ! 9
od
```

Week 2 Outline

- Model checking for device firmware (Contd.)
  - Discussion and limitations
  - Results
- Other papers
  - Model Checking applied to e-commerce

Discussion and Limitations

- Size of the state space
  - In principle, an abstract model generated can have more states than the corresponding detailed model.
  - This is because two different things happen during abstraction.
  - First, some details in the program are discarded. This will reduce the number of states.
  - Second, the compiler uses the abstractions specified conservatively by translating some of the deterministic statements in the program into nondeterministic statements. This can increase the number of states that have to be explored.

Size of the state space (Contd.)

- In practice, the abstract model has significantly fewer states
  - This is because only a small number of nondeterministic statements get introduced.
  - In addition, since the programmer specifies the abstraction in ESP, the programmer can pick the abstraction carefully so as to minimize the number of states.

Limitations (Contd.)

- Bugs
  - By being conservative, ESP ensures that all bugs in the original program are present in the extracted model. However, this does not guarantee that all bugs will be caught during model checking
  - First, the model checker might not be able to check the entire state space because of resource constraints.
  - Second, ESP relies on the programmer to provide test code as well as code that generates values on some channels. A mistake by the programmer can result in bugs being missed during model checking.

Results: Debugging VMMC firmware

- The VMMC firmware was reimplemented using ESP
  - The earlier implementation in C includes about 15600 lines of C code.
  - In contrast, the new implementation using ESP required 500 lines of ESP code together with around 3000 lines of C code.
  - The C code implements simple tasks like initialization, initiating DMA, packet marshalling and unmarshalling and shared data structures with code running on the host processor (in the VMMC library and the VMMC driver).

Results (Contd.)

- Spin was used throughout the development process
  - For instance, the retransmission code, which uses a simple sliding window protocol with piggyback acknowledgement, was developed and debugged entirely using the Spin simulator.
  - Once debugged, the firmware was ported to the network interface card with little effort.

Implementation and Results

- Spin was used to verify memory safety in the VMMC firmware.
  - Instead of supporting safety through garbage collection, ESP provides an explicit malloc/free-style interface to support dynamic memory management.
  - Although this interface is unsafe, the Spin model checker can be used to verify memory safety.
  - To facilitate this, ESP uses a novel scheme that makes memory safety a local property of each process. This allows each process to be verified separately resulting in smaller models. Consequently, memory safety of the VMMC firmware could be checked exhaustively.

Deadlocks in VMMC firmware

- System-wide deadlocks are often a result of complex interactions in the program and can be difficult for programmers to find.
- Therefore, the use of model checking to find these bugs is important.

Deadlocks (Contd.)

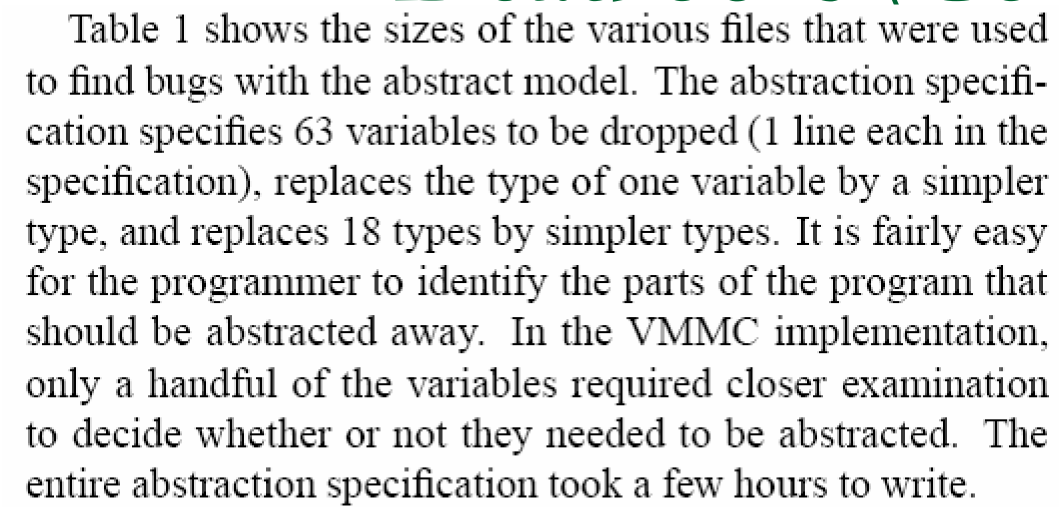
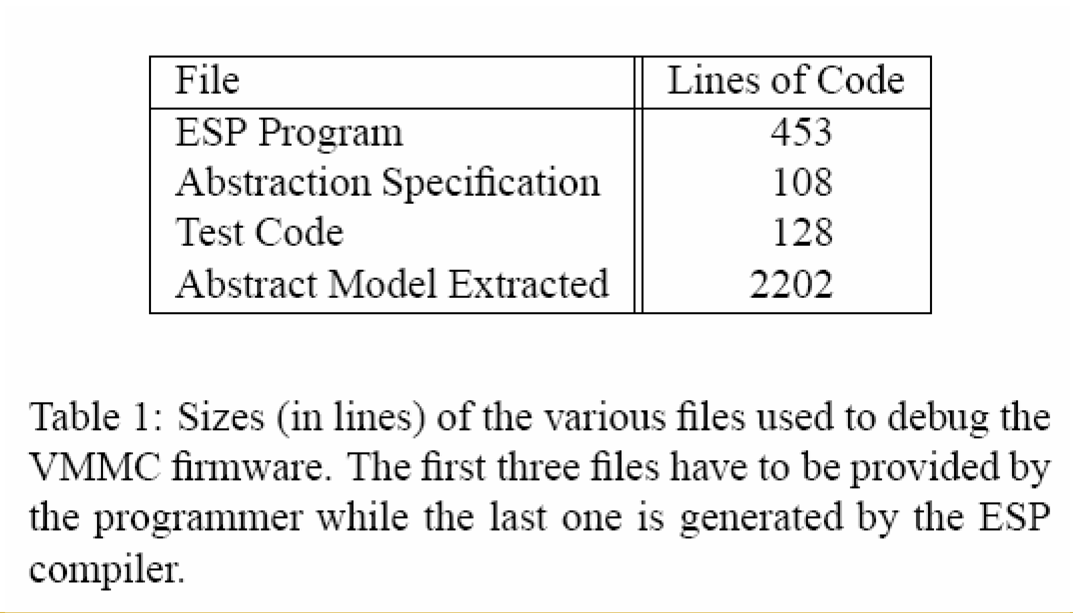


Table 1 shows the sizes of the various files that were used to find bugs with the abstract model. The abstraction specification specifies 63 variables to be dropped (1 line each in the specification), replaces the type of one variable by a simpler type, and replaces 18 types by simpler types. It is fairly easy for the programmer to identify the parts of the program that should be abstracted away. In the VMMC implementation, only a handful of the variables required closer examination to decide whether or not they needed to be abstracted. The entire abstraction specification took a few hours to write.



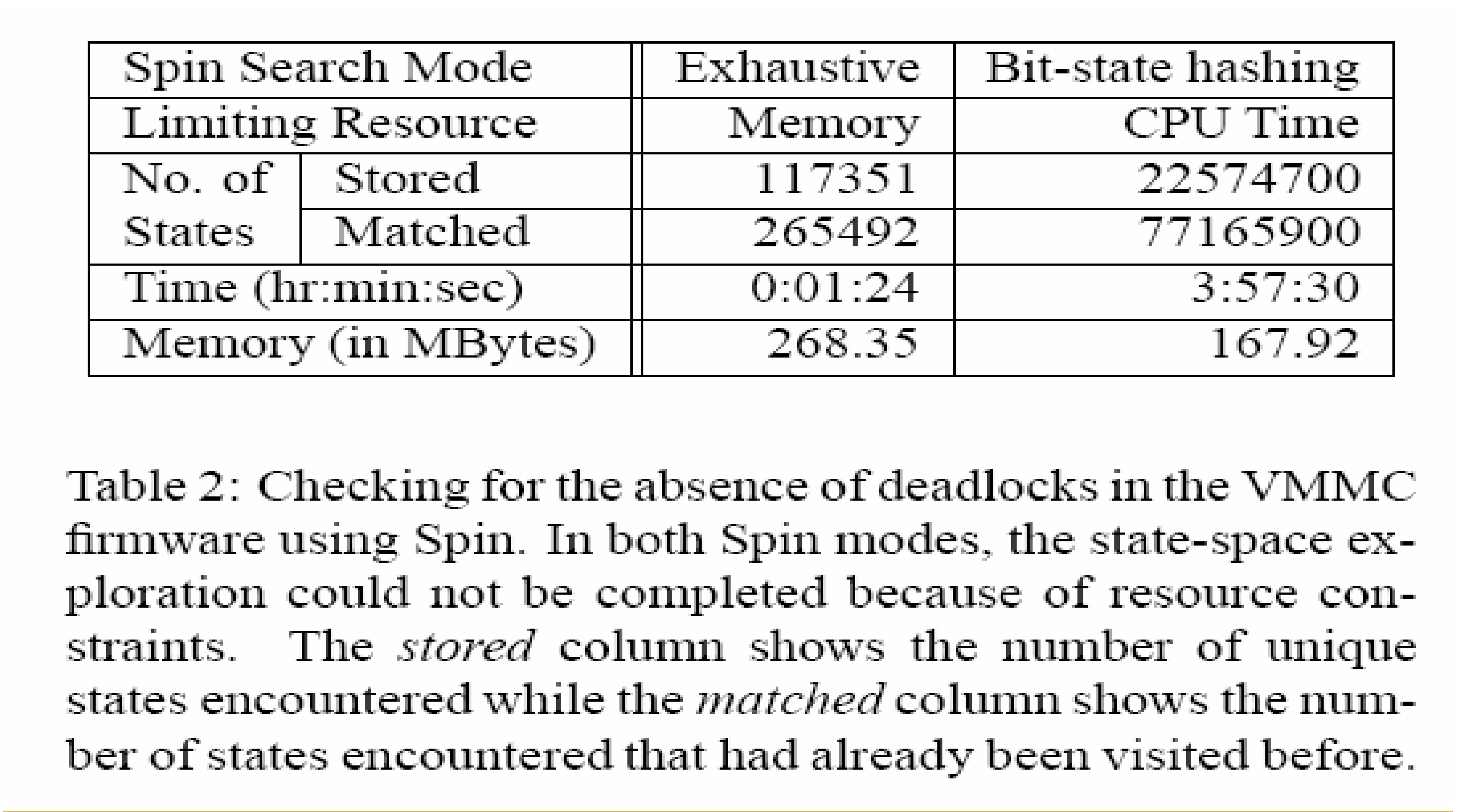
File	Lines of Code
ESP Program	453
Abstraction Specification	108
Test Code	128
Abstract Model Extracted	2202

Table 1: Sizes (in lines) of the various files used to debug the VMMC firmware. The first three files have to be provided by the programmer while the last one is generated by the ESP compiler.

Results (Contd.)

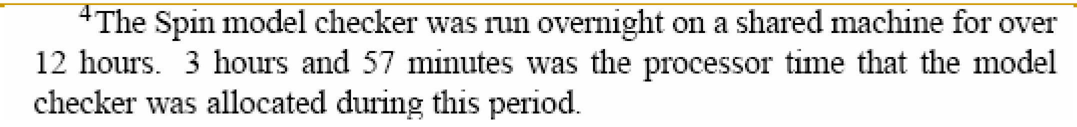
- Using the abstract model, Spin found several subtle bugs in the VMMC firmware that would cause the firmware to deadlock.
- However, even with the abstract model, an exhaustive search of the state space was not possible because of resource constraints.
- Therefore, only partial searches were performed.
- After the bugs that were found were fixed, further state-space exploration using Spin did not uncover any more bugs.

Results (Contd.)



Spin Search Mode		Exhaustive	Bit-state hashing
Limiting Resource		Memory	CPU Time
No. of States	Stored	117351	22574700
	Matched	265492	77165900
Time (hr:min:sec)		0:01:24	3:57:30
Memory (in MBytes)		268.35	167.92

Table 2: Checking for the absence of deadlocks in the VMMC firmware using Spin. In both Spin modes, the state-space exploration could not be completed because of resource constraints. The *stored* column shows the number of unique states encountered while the *matched* column shows the number of states encountered that had already been visited before.

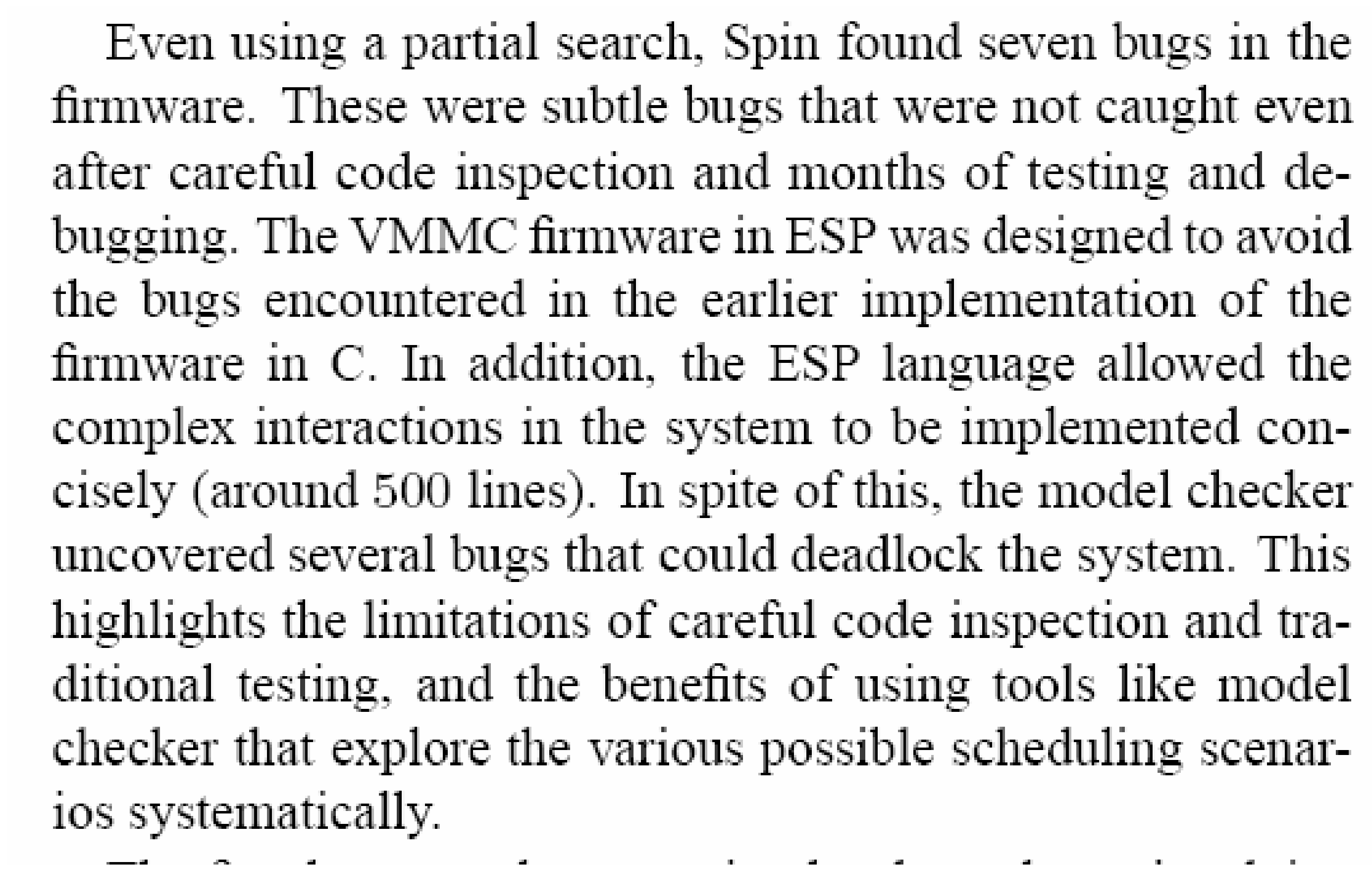


[4] The Spin model checker was run overnight on a shared machine for over 12 hours. 3 hours and 57 minutes was the processor time that the model checker was allocated during this period.

Results (Contd.)

- In the exhaustive mode, Spin had to abort the search after 84 seconds because it ran out of memory.
- In the bit-state hashing mode, Spin ran for 3 hours and 57 minutes before the search was terminated by the user.
- Since Spin only could perform a partial search, we cannot conclude that there are no more bugs in the VMMC firmware. However, the authors have not encountered any bugs while running the firmware on the device that were not caught by Spin

Results (Contd.)



Even using a partial search, Spin found seven bugs in the firmware. These were subtle bugs that were not caught even after careful code inspection and months of testing and debugging. The VMMC firmware in ESP was designed to avoid the bugs encountered in the earlier implementation of the firmware in C. In addition, the ESP language allowed the complex interactions in the system to be implemented concisely (around 500 lines). In spite of this, the model checker uncovered several bugs that could deadlock the system. This highlights the limitations of careful code inspection and traditional testing, and the benefits of using tools like model checker that explore the various possible scheduling scenarios systematically.

Description of bugs

- The first bug was due to a circular dependency involving 3 processes that resulted in a deadlock. Once identified, the deadlock was avoided by eliminating the cycle.
- The second bug involved a situation when the sliding window in the retransmission protocol was full and, therefore, not accepting any new messages to be sent to the network. This eventually led to no new data packets being accepted from the network.
- Since incoming messages were delivered in FIFO order, an explicit acknowledgement message that could unlock the system was trapped behind a data packet resulting in a deadlock.
- To fix this problem, packets have to be dropped occasionally to allow the explicit acknowledgements to get through.

Description of bugs (Contd.)

- The remaining bugs discovered involved receiving unexpected messages or not receiving expected messages.
- The first bug involved receiving acknowledgments with invalid acknowledgement numbers. This was fixed by first checking for the validity of the acknowledgement numbers before using them.
- The second bug involved receiving an unexpected import reply message. These messages are usually received in response to an import request. An unexpected reply would deadlock the system. The problem was fixed by adding code that discarded these unexpected messages.
- The final bug involved not receiving a reply to an import request. This bug can be eliminated using a timeout.

Limitations

- One of the limitations of model checking is that it requires test code to be provided for each property to be verified. The test code is responsible for simulating the environment (external events like network message arrivals) as well as specifies the property to be verified.
- A mistake in the test code can result in the wrong property being checked or, more commonly, failing to explore a part of the state space for bugs. The latter happens when the environment is over constrained.
- For instance, when debugging the retransmission protocol, our initial code simulated a well-behaved environment that only generated network packets with the expected sequence numbers.
- Later, we found bugs when the test code was modified to generate unexpected messages.

Limitations (Contd.)

- Another problem that they encountered when looking for deadlocks was that a deadlock that involved only a few processes in the model could go undetected.
- This is because a deadlock in Spin is a state out of which there are no transitions. When a deadlock involves only a few processes, the remaining processes can sometimes make progress.
- Consequently, there are always transitions out of the current state. Therefore, the state machine is not technically deadlocked. For instance, this happens when a deadlock in one part of the system prevents messages from being sent while another part of the system responsible for receiving messages from the network continues to function correctly.
- To avoid this, the test code can be changed to ensure that no part can inactive for long periods of time (by maintaining activity counters).

Conclusions

- ESP allows abstract models to be extracted from the programs.
- These models are smaller because they omit details that are irrelevant to the property being checked. In ESP, the programmer specifies the abstraction and therefore has control over the abstraction process. The ESP compiler uses the abstractions specified by the programmer conservatively when generating an abstract model. This ensures a bug in the ESP program will be in the generated model even when a programmer makes a mistake in specifying the abstraction.

Conclusions (Contd.)

- The use of model checker is greatly simplified when the models could be automatically extracted from the ESP programs by the compiler.
- A model checker is very effective as a debugging tool. Sometimes, only a partial state-space exploration is possible due to resource constraints. However, this is acceptable because the goal is to identify bugs and not to certify correctness.
- Even a partial systematic search by the model checker results in more extensive testing than traditional testing methods and can be invaluable in debugging concurrent firmware.

Next

- "The application of model checking for securing e-commerce transactions" by Bonnie Brinton Anderson, James V. Hansen, Paul Benjamin Lowry, Scott L. Summers, CACM 2006

Model checking for securing e-commerce transactions

- As firms become progressively more dependent on Internet-based information systems, they are increasingly vulnerable to defects in those systems. These defects can lead to errors, undetected fraud, and malicious intrusion.
- Information system errors can be catastrophic, whether they occur in market transactions, banking, air traffic control, and so forth. Resulting damages can include lost revenue, lost data, lost trust, and increased costs

Motivation

- Consequently, effective design of e-business processes is essential to avoid defects that can otherwise lead to errors, fraud, and intrusion.
- Whereas carefully designed e-business protocols can perform effectively within most expected situations, assuring correct processing under all circumstances is exceptionally complex and difficult.

Assuring correct processing

- Hidden flaws and errors that occur only under unexpected and hard-to-anticipate circumstances can lead to processing errors and potentially ruinous failures.

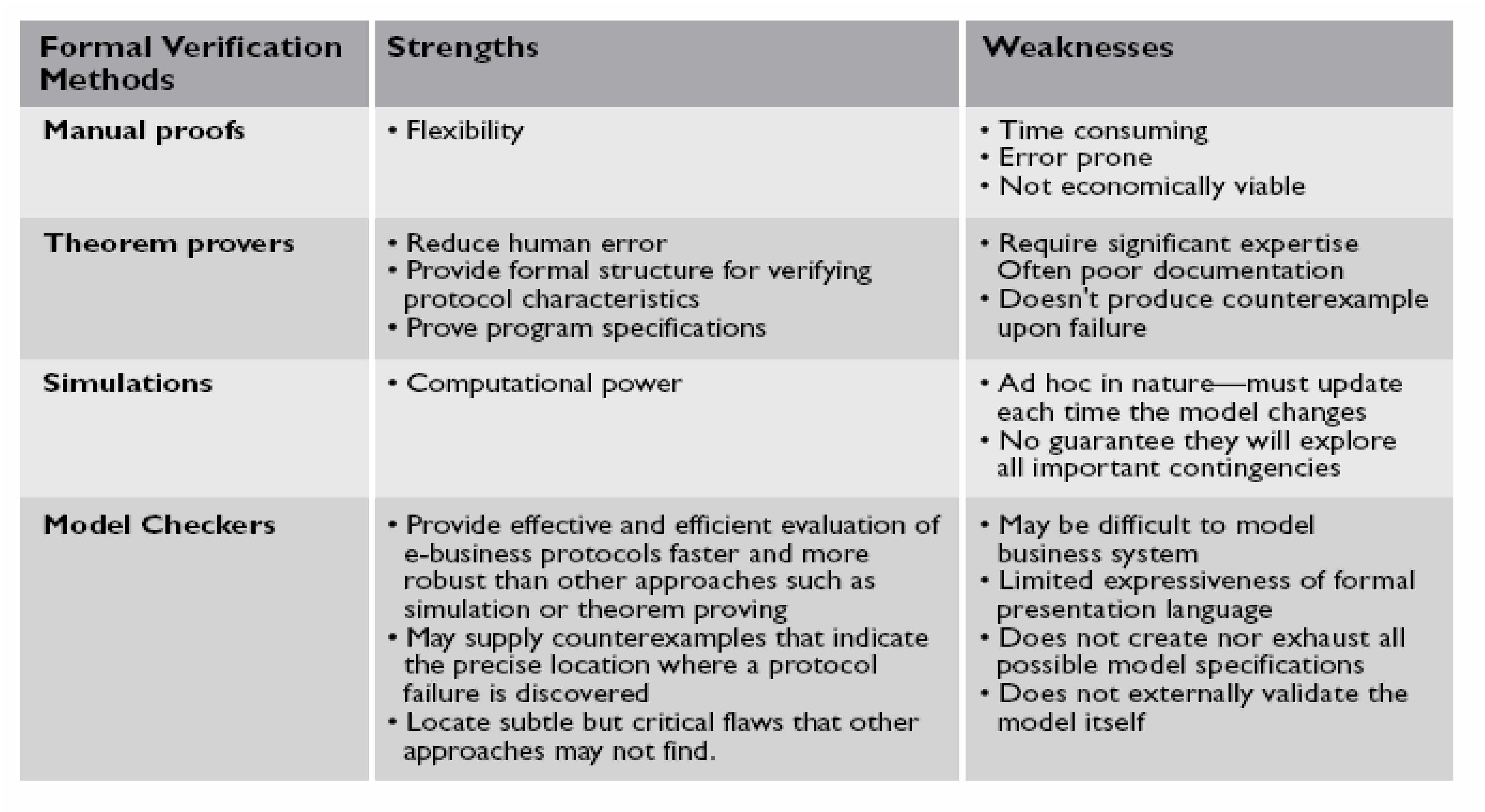
Different Approaches

- Verifying an e-business protocol is robust against hidden flaws and errors is a formidable task.
- Manual methods are slow and error prone. Even theorem provers, which provide a formal structure for verifying protocol characteristics, may require human intervention and can be time consuming. Moreover, theorem provers generally do not provide much help in locating failure sources.
- Finally, simulations offer computational power, but they are ad hoc in nature and there is no guarantee they will explore all important contingencies

Model Checking

- Conversely, model checking is an evolving technology that offers a platform for effective and efficient evaluation of e-business protocols. Current model checking technology is based on automated techniques that are considerably faster and more robust than other approaches such as simulation or theorem proving.
- With today's model checkers, large state spaces can be analyzed in minutes. Additionally, model checkers are able to extend their analysis by supplying counterexamples that identify the precise location where a protocol failure is discovered

Comparison of formal verification methods



Formal Verification Methods	Strengths	Weaknesses
Manual proofs	• Flexibility	• Time consuming • Error prone • Not economically viable
Theorem provers	• Reduce human error • Provide formal structure for verifying protocol characteristics • Prove program specifications	• Require significant expertise Often poor documentation • Doesn't produce counterexample upon failure
Simulations	• Computational power	• Ad hoc in nature—must update each time the model changes • No guarantee they will explore all important contingencies
Model Checkers	• Provide effective and efficient evaluation of e-business protocols faster and more robust than other approaches such as simulation or theorem proving • May supply counterexamples that indicate the precise location where a protocol failure is discovered • Locate subtle but critical flaws that other approaches may not find.	• May be difficult to model business system • Limited expressiveness of formal presentation language • Does not create nor exhaust all possible model specifications • Does not externally validate the model itself

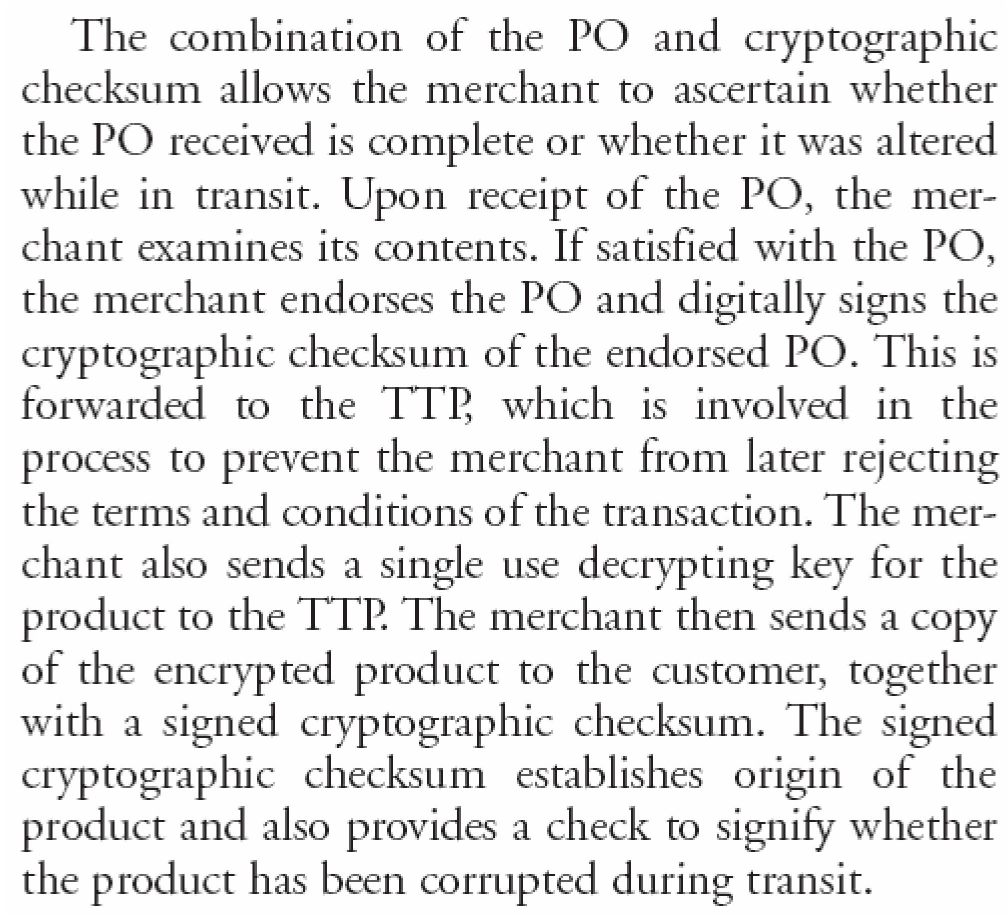
E-business protocol example

- Messages are exchanged between a customer, a merchant, and a trusted third party (TTP). A merchant has several products to sell. The merchant places a description of each digital product in an online catalog service with a TTP along with a copy of the encrypted product.
- When a customer finds a product of interest in the catalog, the customer downloads the encrypted product and sends a purchase order (PO) to the merchant. The customer cannot use the product until it is decrypted, and the merchant does not send the decrypting key unless the merchant receives a payment token through the PO process.
- The customer, in turn, does not pay unless he/she is sure the correct and complete product has been received. The TTP provides support for PO validation, payment token approval, and approval of the overall transaction between the customer and the merchant.

The transaction in details

- The process begins as the customer browses the product catalog located at the TTP and chooses a product. The customer then downloads the encrypted product along with the product identifier— a file that contains product information such as its description and identifier.
- If the identifier of the encrypted product file corresponds to the identifier in the product identifier file, the transaction proceeds. If the identifiers do not match, an advice is sent to the TTP and the customer waits for the correct encrypted product. This process ensures the customer receives the product requested from the catalog.
- Next, the customer prepares a PO containing the customer's identity, the merchant identifier, the product identifier, and the product price. A cryptographic checksum is also prepared. The PO along with the cryptographic checksum is then sent to the merchant.

Transaction (Contd.)



The combination of the PO and cryptographic checksum allows the merchant to ascertain whether the PO received is complete or whether it was altered while in transit. Upon receipt of the PO, the merchant examines its contents. If satisfied with the PO, the merchant endorses the PO and digitally signs the cryptographic checksum of the endorsed PO. This is forwarded to the TTP, which is involved in the process to prevent the merchant from later rejecting the terms and conditions of the transaction. The merchant also sends a single use decrypting key for the product to the TTP. The merchant then sends a copy of the encrypted product to the customer, together with a signed cryptographic checksum. The signed cryptographic checksum establishes origin of the product and also provides a check to signify whether the product has been corrupted during transit.

Transaction (Contd.)

- Upon receipt of the second copy of the encrypted product, the customer verifies the first and second copies of the product are identical. Through this process, customers can be assured they received the product ordered.
- The customer then requests the decrypting key from the TTP. To do this, the customer forwards the PO and a signed payment token to the TTP, together with its cryptographic checksum.
- The payment token contains the customer's identity, the identity of the customer's financial institution, the customer's bank account number and the amount to be debited from the customer's account.

Transaction (Contd.)

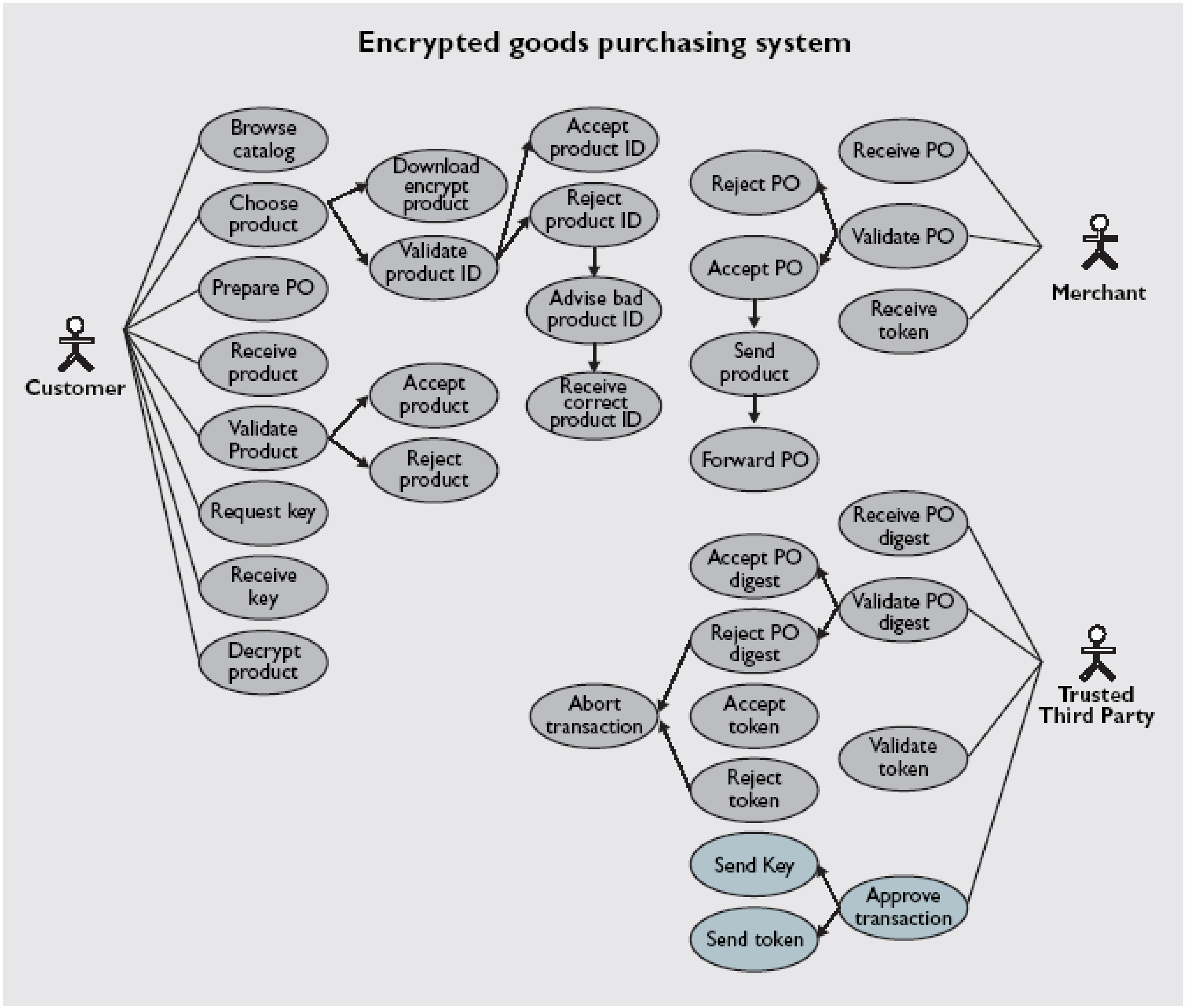
- To verify the transaction, the TTP first compares the digest included in PO from the customer with the digest of the same from the merchant. If the two do not match, the TTP aborts the transaction.
- Otherwise the TTP proceeds by validating the payment token with the customer’s financial institution by presenting the token and the sale price. The financial institution validates the token. If the token is not validated, the TTP aborts the transaction and advises the merchant accordingly.
- If the token is validated, the TTP sends the decrypting key to the customer and the payment token to the merchant, both digitally signed with the TTP’s private key.

Transaction (Contd.)

- Secure channels guarantee the confidentiality of all messages throughout this protocol, which ensures money atomicity if the payment token generated by the customer contains the amount to be debited from the customer’s account and credited to the merchant’s account.
- Consequently, no money is created or destroyed in the system by this protocol.
- Goods atomicity is guaranteed if the TTP submits the payment token only when the customer acknowledges the receipt of the product. The process also ensures the product is actually available to the customer for use.

Transaction (Contd.)

- Delivery verification is guaranteed if the TTP receives a cryptographic checksum of the product from the merchant. Also, the customer independently generates a checksum of the product received and sends it to the TTP.
- Using these two copies of the checksums, which are available at the TTP, both the merchant and the consumer demonstrate proof of the contents of the delivered goods.


Encrypted goods purchasing system
Customer
Browse catalog
Choose product
Prepare PO
Receive product
Validate Product
Request key
Receive key
Decrypt product
Download encrypt product
Validate product ID
Accept product
Reject product
Accept product ID
Reject product ID
Advise bad product ID
Receive correct product ID
Reject PO
Accept PO
Send product
Forward PO
Receive PO
Validate PO
Receive token
Merchant
Accept PO digest
Reject PO digest
Accept token
Reject token
Abort transaction
Send Key
Send token
Receive PO digest
Validate PO digest
Validate token
Approve transaction
Trusted Third Party

Example of Model Checking

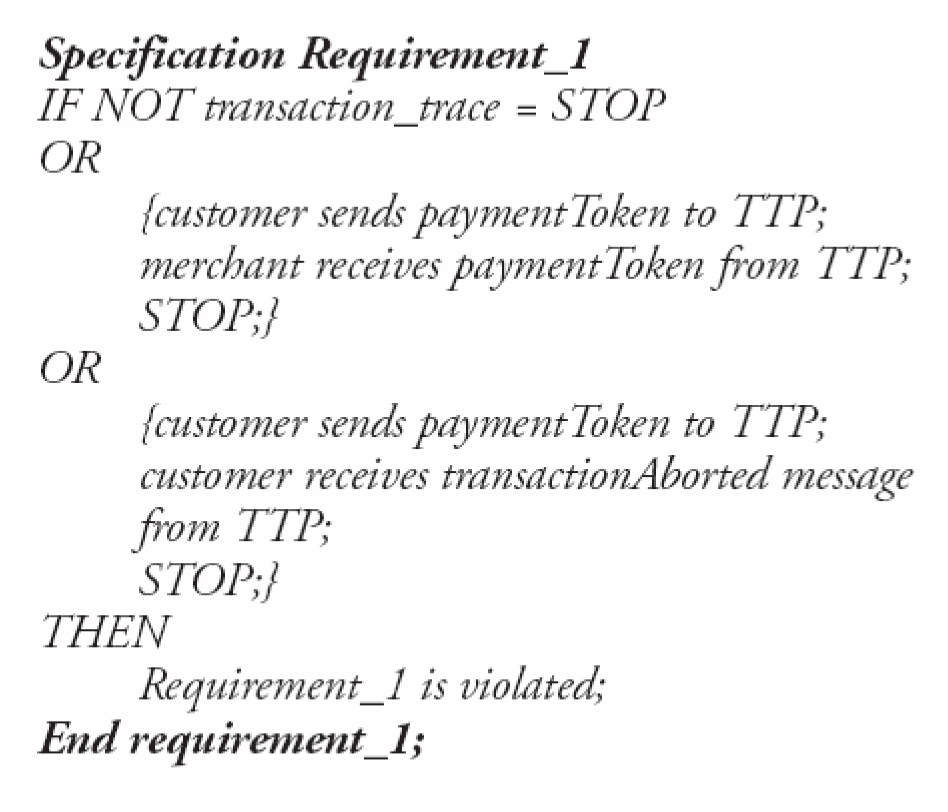
- Three Important requirements:
  - Money is neither created nor destroyed in the course of an e-commerce transaction. A transaction should ensure the transfer of funds from one party to another without the possibility of the creation or destruction of money. No viable e-commerce payment method can exist without supporting this property.
  - Both the customer and merchant should receive evidence that the goods sent (or received) are those to which both parties agreed. Particularly when dealing with goods that can be transferred electronically, the combination of both is essential.

Requirements (Contd.)

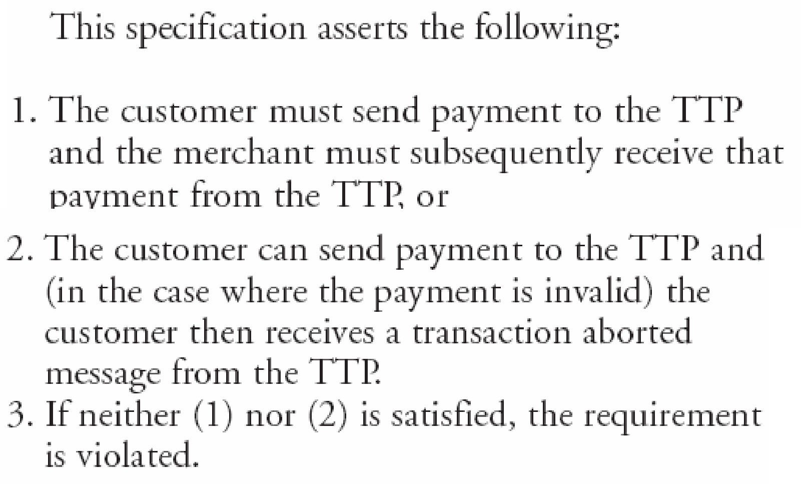
- The customer is able to verify the contents of the product about to be received before making payment. The customer must be able to verify the product about to be received is the same product that was ordered, before the customer pays for the product.

Model Checking

- Used to write specifications



```
Specification Requirement_1
IF NOT transaction_trace = STOP
OR
        {customer sends paymentToken to TTP;
        merchant receives paymentToken from TTP;
        STOP;}
OR
        {customer sends paymentToken to TTP;
        customer receives transactionAborted message
        from TTP;
        STOP;}
THEN
        Requirement_1 is violated;
End requirement_1;
```

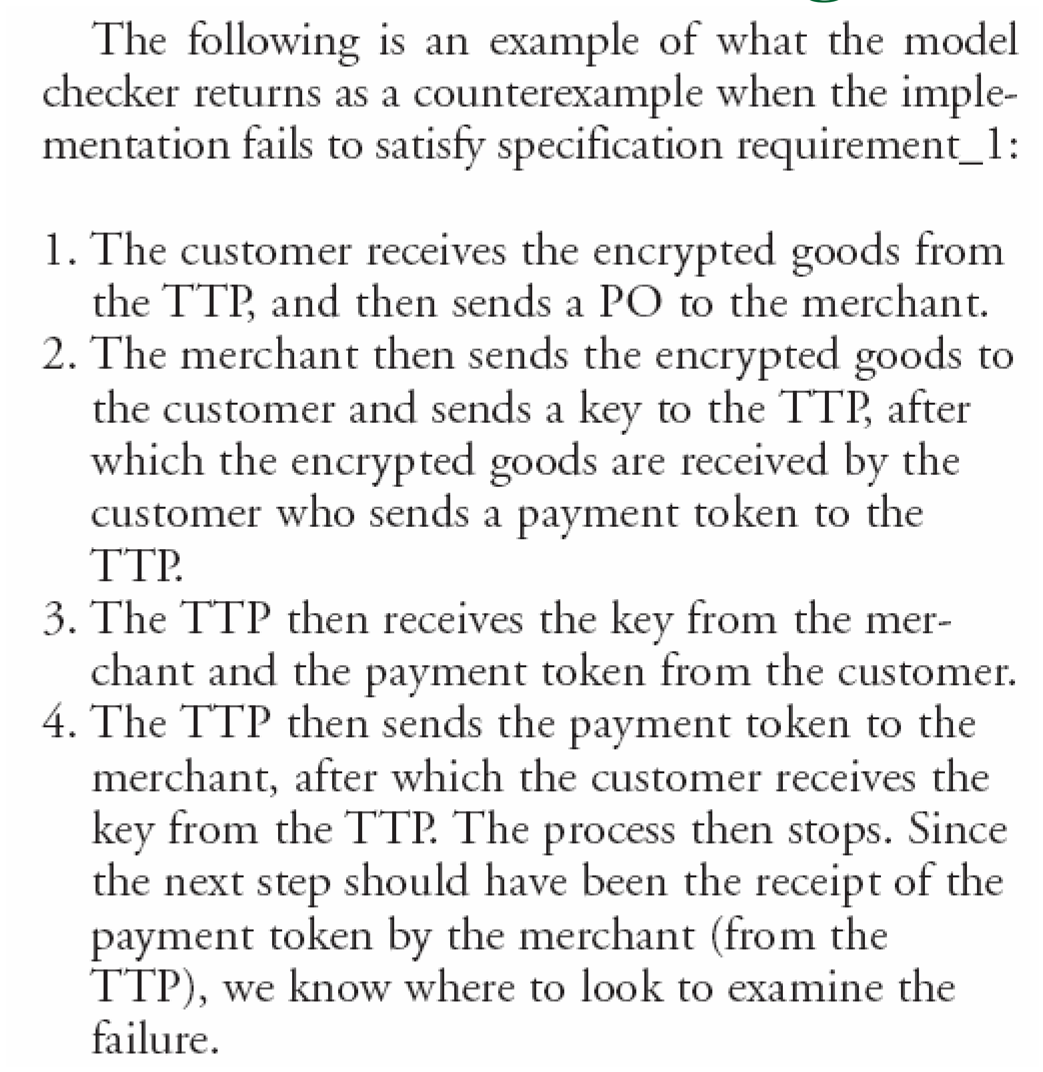


This specification asserts the following:

1. The customer must send payment to the TTP and the merchant must subsequently receive that payment from the TTP, or
2. The customer can send payment to the TTP and (in the case where the payment is invalid) the customer then receives a transaction aborted message from the TTP.
3. If neither (1) nor (2) is satisfied, the requirement is violated.

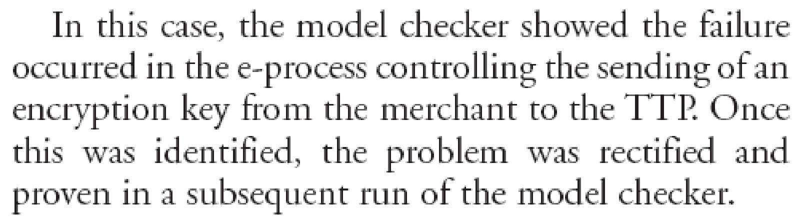
PURDUE UNIVERSITY

Model Checking: A counterexample



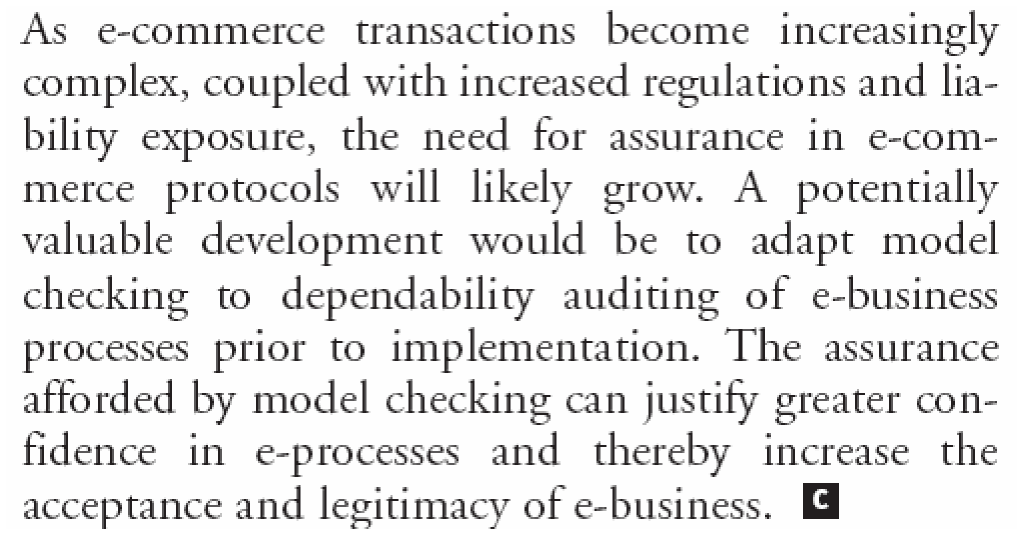
The following is an example of what the model checker returns as a counterexample when the implementation fails to satisfy specification requirement_1:

1. The customer receives the encrypted goods from the TTP, and then sends a PO to the merchant.
2. The merchant then sends the encrypted goods to the customer and sends a key to the TTP, after which the encrypted goods are received by the customer who sends a payment token to the TTP.
3. The TTP then receives the key from the merchant and the payment token from the customer.
4. The TTP then sends the payment token to the merchant, after which the customer receives the key from the TTP. The process then stops. Since the next step should have been the receipt of the payment token by the merchant (from the TTP), we know where to look to examine the failure.



In this case, the model checker showed the failure occurred in the e-process controlling the sending of an encryption key from the merchant to the TTP. Once this was identified, the problem was rectified and proven in a subsequent run of the model checker.

Conclusions


As e-commerce transactions become increasingly complex, coupled with increased regulations and liability exposure, the need for assurance in e-commerce protocols will likely grow. A potentially valuable development would be to adapt model checking to dependability auditing of e-business processes prior to implementation. The assurance afforded by model checking can justify greater confidence in e-processes and thereby increase the acceptance and legitimacy of e-business. C

Next

- Junfeng Yang, Paul Twohey, Dawson Engler, Madanlal Musuvathi, **Using Model Checking to Find Serious File System Errors, OSDI 04**

Motivation

- File systems have two dynamics that make them attractive for the model checking approach
- First, their errors are some of the most serious, since they can destroy persistent data and lead to unrecoverable corruption.
- Second, traditional testing needs an impractical, exponential number of test cases to check that the system will recover if it crashes at any point during execution.
- Model checking employs a variety of state-reducing techniques that allow it to explore such vast state spaces efficiently.

Introduction

- Since almost all deployed file systems reside in the operating system kernel, even a simple error can crash the entire system, most likely in the midst of a mutation to stable state.
- Bugs in file system code can range from those that cause “mere” reboots to those that lead to unrecoverable errors in stable on disk state.

Challenges

- Not only are errors in file systems dangerous, file system code is simultaneously both difficult to reason about and difficult to test.
- The file system must correctly recover to an internally consistent state if the system crashes at *any* point, regardless of what data is being mutated, flushed or not flushed to disk, and what invariants have been violated.
- Anticipating all possible failures and correctly recovering from them is known to be hard;

Testing

- Testing that a file system correctly recovers from a crash requires doing reconstruction and then comparing the reconstructed state to a known legal state.
- The cost of a single crash-reboot-reconstruct cycle (typically a minute or more) makes it impossible to test more than a tiny fraction of the exponential number of crash possibilities.

Model Checking

- Model checkers employ various *state reduction* techniques to efficiently explore the exponential state space.
- The dominant cost of traditional model checking is the effort needed to write an abstract specification of the system (commonly referred to as the “model”).
- This upfront cost has traditionally made model checking completely impractical for large systems.

Model checkers

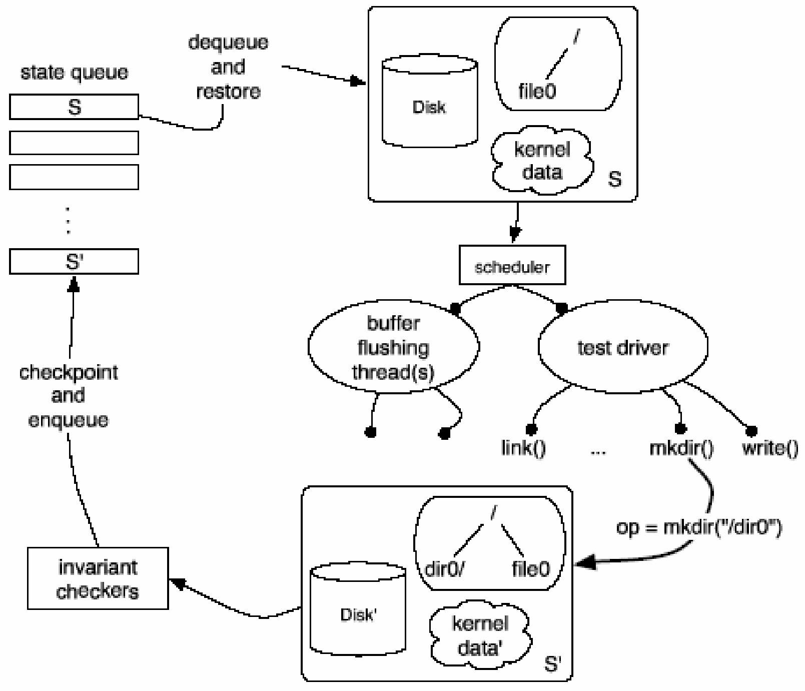
- Recent work has developed *implementation-level* model checkers that check implementation code directly without requiring an abstract specification
- They leverage this approach to create a model checking infrastructure, the File System Checker (FiSC), which lets implementors model-check real, unmodified file systems with relatively little model checking knowledge.

Parts of the system

- Our system is comprised of four parts: (1) CMC, an explicit state model checker running the Linux kernel,
- A file system test driver
- A permutation checker which verifies that a file system can recover no matter what order buffer cache contents are written to disk, and
- A fsck recovery checker.

Execution

- The model checker starts in an initial pristine state (an empty, formatted disk) and recursively generates and checks successive states by systematically executing state transitions.
- Transitions are either test driver operations or FS-specific kernel threads which flush blocks to disk.
- The test driver is conceptually similar to a program run during testing.



Model Checking Loop

```
while S = dequeue()
    for each op in S.operations
        S' =  checkpoint( op(restore(S) )
        check(S')
        if lookup(S'.disk, S'.abstract_fs) is new
            enqueue(S')
```

Figure 1: State exploration and checking overview. FiSC's main loop picks a state S from the state queue and then iteratively generates its successor states by applying each possible operation to a restored copy of S. The generated state S' is checked for validity and, if valid and not explored before, inserted onto the state queue.

Execution

- As each new state is generated, we intercept all disk writes done by the checked file system and forward them to the permutation checker, which checks that the disk is in a state that fsck can repair to produce a valid file system after each subset of all possible disk writes.
- This avoids storing a separate state for each permutation and allows FiSC to choose which permutations to check.

Execution

- We run fsck on the *host* system outside of the model checker and use a small shared library to capture all the disk accesses fsck makes while repairing the file system generated by writing a permutation.
- We feed these fsck generated writes into the crash recovery checker. This checker allows FiSC to recursively check for failures in fsck

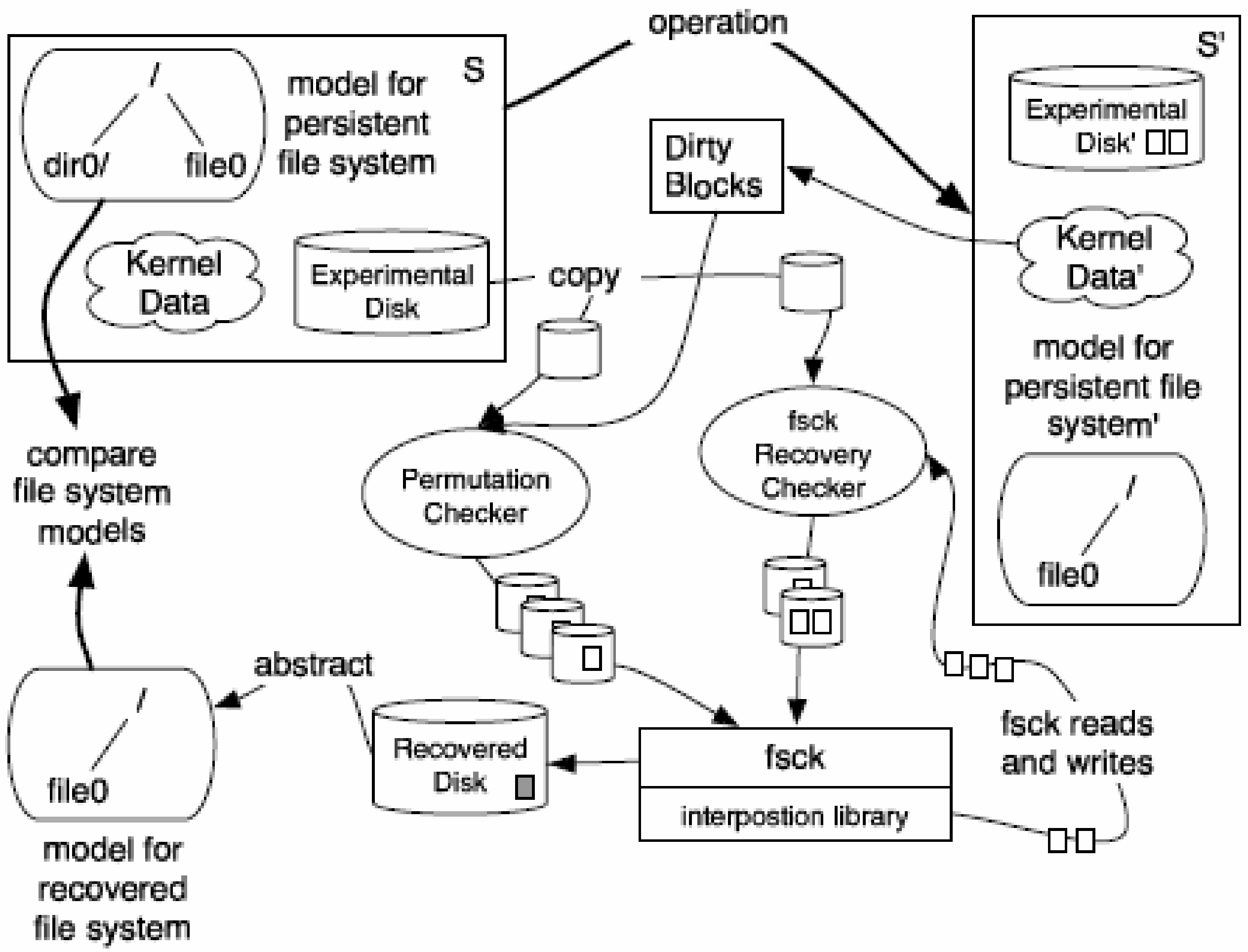


Figure 2: Disk permutation and fsck recovery checkers.

Execution

- Both checkers copy the disk from the starting state of a transition and write onto the copy to avoid perturbing the system.
- After the copied disk is modified the model checker traverses its file system, recording the properties it checks for consistency in a model of the file system.
- Currently these are the name, size, and link count of every file and directory in the system along with the contents of each directory.
- We check that this model matches one of the possible valid file systems. An error is flagged in case of a mismatch.

Execution

- After each new state is generated our system runs a series of invariant checkers looking for file system errors.
- If an error is found FiSC (1) emits an error message, (2) stores a trace for later error diagnosis that records all the nondeterministic choices it made to get to the error, and (3) discards the state.
- If there is no error, FiSC looks at the new state and puts it on the state queue if it has not already visited a similar state. Otherwise, it discards the state.

Execution

- New states are *checkpointed* and added to the state queue for later exploration. Checkpointing the kernel state captures the current execution environment so that it can be put on the state queue and *restored* later when the model checker decides to take it off the state queue and explore its operations.
- This state consists of the kernel heap and data, the disk, an abstract model of the current file system, and additional data necessary for invariant checks.

The Checking Environment

- Model-checking a file system requires selecting two layers at which to cut the checked system.
- One layer defines the external interface that the test driver runs on. In our case we have the driver run atop the system call interface.
- The other layer provides a “fake environment” that the checked system runs on.
- We need this *environment model* because the checked file system does not run on bare hardware. Instead, FiSC provides a virtual block device that models a disk as a collection of sectors that can be written atomically.
- The block device driver layer is a natural place to cut as it is the only relatively well-documented boundary between in-core and persistent data.

Checking a new file system

- Because CMC encases Linux, a file system that already runs within Linux and conforms to FiSC's assumptions of file system behavior will require relatively few modifications before it can be checked.
- FiSC needs to know the minimum disk and memory sizes the file system requires. Ext3 had the smallest requirements: a 2MB disk and 16 pages of memory.
- In addition, it needs the commands to make and recover the file system (usually with mkfs and fsck respectively).
- Ideally, the developer provides three different fsck options for: (1) default recovery, (2) "slow" full recovery, and (3) "fast" recovery that only replays the journal so that the three recovery modes may be checked against each other

Basic Setup

- In addition to providing these facts, an implementor may have to modify their file system to expose dirty blocks. (buffers)
- During crash checking FiSC mounts a copy of the disk used by the checked file system as a second block device that it uses to check the original.
- Thus, the file system must independently manage two disks.

Setup

- Unfortunately, ReiserFS does not do so: it uses a single kernel thread to perform journal writes for all mounted devices, which causes a deadlock when the journal thread writes to the log, FiSC suspends it, creates a copy of the disk, and then remounts the file system.
- Remounting normally replays the journal, but this requires writing to the journal – which deadlocks waiting for the suspended journal thread to run.
- We fixed the problem by modifying ReiserFS to not wake the journal thread when a clean file system is mounted read-only.

Modeling the File System

- After every file system operation, FiSC compares the checked file system against what it believes is the correct *volatile file system* (VolatileFS).
- The VolatileFS reflects the effects of all file system operations done sequentially up through the last one. Because it is defined by various standards rather than being FS-specific, FiSC can construct it as follows.
- After FiSC performs an operation (e.g., mkdir, link) to the checked concrete system, it also emulates the operation's effect on a "fake" abstract file system. It then verifies that the checked and abstract file systems are equivalent using a lossy comparison that discards details such as time.

Modeling

- After every disk write, FiSC compares the checked file system against a model of what it believes to be the current *stable file system* (StableFS).
- The StableFS reflects the state the file system should recover to after a crash.
- At any point, running a file system's fsck repair utility on the current disk should always produce a file system equivalent to this StableFS.

Modeling

- Unlike the VolatileFS, the StableFS is FS-specific.
- Different file systems make wildly different guarantees as to what will be recovered after a crash.
- Determining how the StableFS evolves requires determining two FS-specific facts: (1) when it can legally change and (2) what it changes to.
- FiSC requires that the implementor modify the checked file system to call into the model checker to indicate when the StableFS changes.

Modeling

- For journaling file systems this change typically occurs when a journal commit record is written to disk.
- We were able to identify and annotate the commit records relatively easily for ext3 and ReiserFS. JFS was more difficult. In the end, after a variety of false starts, we gave up trying to determine which journal write represented a commit-point and instead let the StableFS change after *any* journal write.

Modeling

- Once we know that the StableFS changes, we need to know what it changes to. Doing so is difficult since it essentially requires writing a crash recovery specification for each file system.
- While we assume a file system implementor could do so, we check systems we did not build. Thus, we take a shortcut and use fsck to generate the StableFS for us. We copy the experimental disk, run fsck to reconstruct a file system image after the committed operations, and traverse the file system, recording properties of interest.
- This approach can miss errors since we have no guarantee that fsck will produce the correct state. However, it is relatively unlikely that fsck will fail when repairing a perfectly behaving disk.

Checking more Thoroughly

- **Downscale**. Operationally this means making everything as small as plausible. Caches become one or two entries large, file systems just a few “nodes” (where a node is a file or directory).
- There were three main places we downscaled.
- First, making disk small (megabytes rather than gigabytes).
- Second, checking small file system topologies, typically 2-4 nodes.
- Finally, reducing the size of “virtual memory” of the checked Linux system to a small number of pages.

Second Technique

- **Canonicalization**. This technique modifies states so that state hashing will not see “irrelevant” differences.
- In practice, the most common canonicalization is to set as many things as possible to constant values: clearing inode generation numbers, mount counts, time fields; zeroing freed memory and unused disk blocks (especially journal blocks).

Canonicalization

- However, FiSC does do two generic canonicalizations.
- First, it constrains the search space by only writing two different values to data blocks, significantly reducing the number of states while still providing enough resolution to catch data errors.
- Second, before hashing a model of a file system, FiSC transforms the file system to remove superficial differences, by renaming files and directories so that there is always a sequential numbering among file system objects.

Expose choice points

- Making sources of nondeterminism(“choice points”) visible to FiSC lets it search the set of possible file system behaviors more thoroughly. A low-level example is adding code to fail FS-specific allocators.
- More generally, whenever a file system makes a decision based on an arbitrary time constraint or environmental feature, we change it to call into FiSC so that FiSC can choose to explore each possible decision in every state that reaches that point.

Expose Choice Points

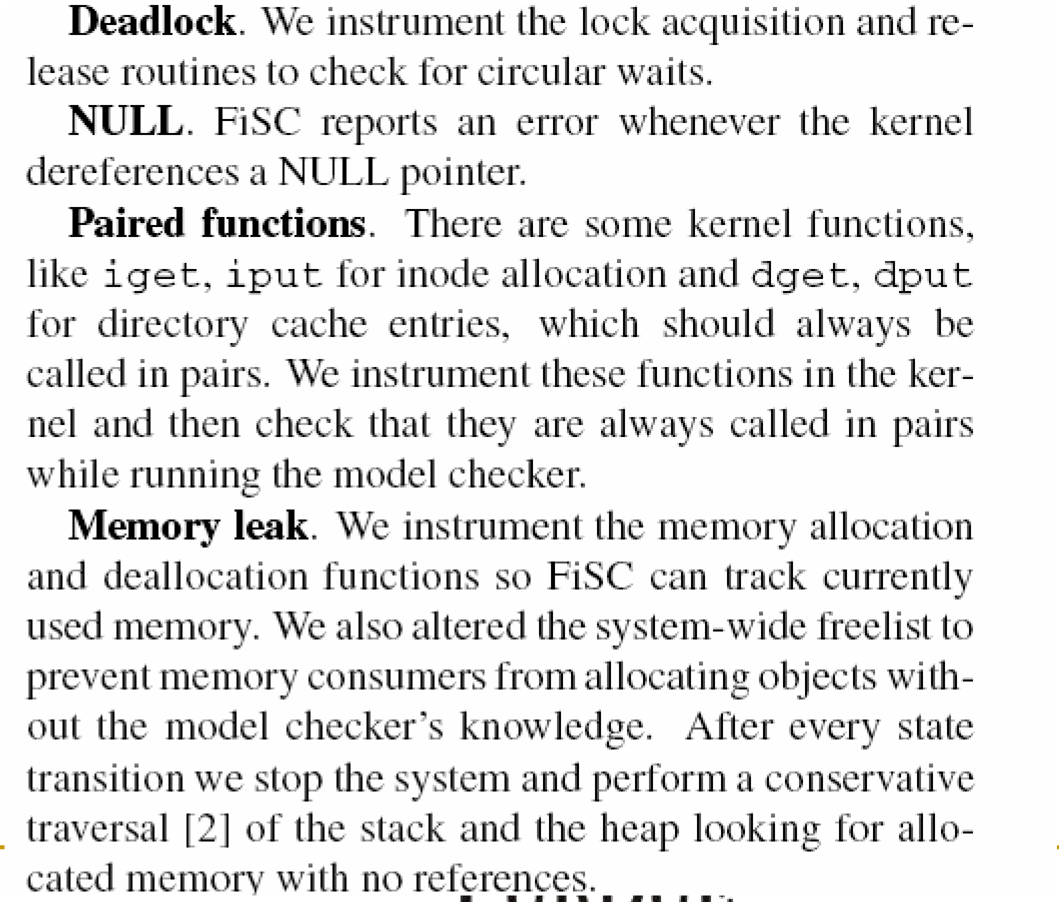
- Mechanically, exposing a choice point reduces to modifying the file system code to call “choose(n)” where n is the number of possible decision alternatives.
- choose will appear to return to this callsite n times, with the return values 0, . . . , (n − 1).
- The caller uses this return value to pick which of the n possible actions to perform.

Expose Choice Points

- When inserting choice points, the implementor can exploit well-defined internal interfaces to increase the set of explored actions.
- Interface specifications typically allow a range of actions, of which an implementation will pick some subset.
- Unfortunately, it is not always easy to expose choice points and may require restructuring parts of the system to remove artificial constraints. The most invasive example of these modifications are the changes to the buffer cache we made so that the permutation checker would be able to see all possible buffer write orderings.

Checks that FiSC performs

- Generic Checks



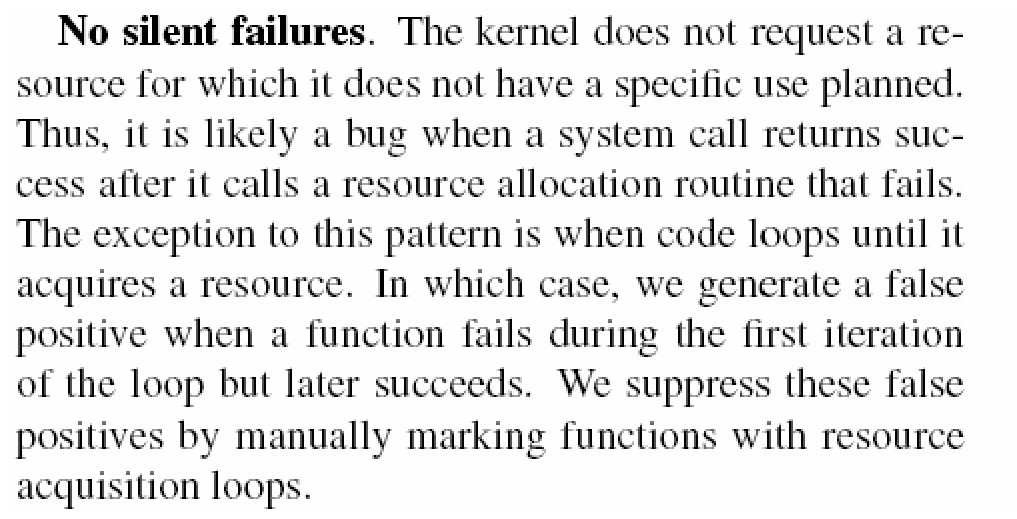
Deadlock. We instrument the lock acquisition and release routines to check for circular waits.

NULL. FiSC reports an error whenever the kernel dereferences a NULL pointer.

Paired functions. There are some kernel functions, like `iget`, `iput` for inode allocation and `dget`, `dput` for directory cache entries, which should always be called in pairs. We instrument these functions in the kernel and then check that they are always called in pairs while running the model checker.

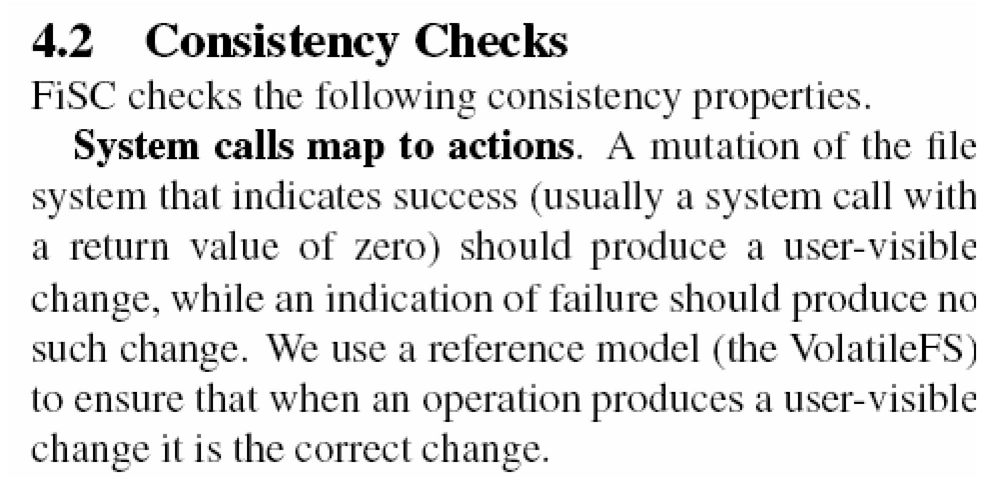
Memory leak. We instrument the memory allocation and deallocation functions so FiSC can track currently used memory. We also altered the system-wide freelist to prevent memory consumers from allocating objects without the model checker's knowledge. After every state transition we stop the system and perform a conservative traversal [2] of the stack and the heap looking for allocated memory with no references.

Generic Checks



No silent failures. The kernel does not request a resource for which it does not have a specific use planned. Thus, it is likely a bug when a system call returns success after it calls a resource allocation routine that fails. The exception to this pattern is when code loops until it acquires a resource. In which case, we generate a false positive when a function fails during the first iteration of the loop but later succeeds. We suppress these false positives by manually marking functions with resource acquisition loops.

Consistency Checks



4.2 Consistency Checks

FiSC checks the following consistency properties.

System calls map to actions. A mutation of the file system that indicates success (usually a system call with a return value of zero) should produce a user-visible change, while an indication of failure should produce no such change. We use a reference model (the VolatileFS) to ensure that when an operation produces a user-visible change it is the correct change.

Consistency Checks

- **Recoverable disk write ordering**:
  - File system recovery code typically requires that disk writes happen in certain stylized orders.
  - Illegal orders may not interfere with normal system operation, but will lead to unrecoverable data loss if a crash occurs at an inopportune moment.
  - Comprehensively checking for these errors requires we (1) have the largest legal set of possible dirty buffers in memory and (2) flush combinations of these blocks to disk at every legal opportunity.

Consistency Checks

- Unfortunately, many file systems (all those we check) thwart these desires by using a background thread to periodically write dirty blocks to disk.
- Further, the vagaries of thread scheduling can hide vulnerabilities — if the thread does not run when the system is in a vulnerable state then the dangerous disk writes will not happen.
- Thus we modified this thread to do nothing and instead have the model checker track all blocks that could be legally written.

Consistency Checks

- Whenever a block is added to this set we write out different permutations of the set, and verify that running fsck produces a valid file system image.
- The set of possible blocks that can be written are (1) all dirty buffers in the buffer cache (dirty buffers may be written in any order) and (2) all requests in the disk queue (disks routinely reorder the disk queue).
- This set is initially empty. Blocks are added whenever a buffer cache entry is marked dirty. Blocks are removed from this set in four ways: (1) they are deleted from the buffer cache, (2) marked clean, (3) the file system explicitly waits for the block to be written or (4) the file system forces a synchronous write of a specific buffer or the entire disk request queue.

Consistency Checks

- **Changed buffers are marked dirty:**
  - When a file system changes a block in the buffer cache it needs to mark it as dirty so the operating system knows it should eventually write the block back to disk.
  - Blocks that are not marked as dirty may be flushed from the cache at any time. Initially we thought we could use the generic dirty bit associated with each buffer to track the "dirtiness" of a buffer, but each file system has a slightly different concept of what it means for a buffer to be dirty.
  - For example, ext3 considers a buffer dirty if one of the following conditions is true: (1) the generic dirty bit is set, (2) the buffer is journaled and the journal dirty bit is set, or (3) the buffer is journaled and it has been revoked and the revocation is valid. Discovering dirty buffer invariants requires intimate knowledge of the file system design; thus we have only run this checker on ext3.

Consistency Checks

- **Buffer consistency:**
  - Each journaling file system associates state with each buffer it uses from the buffer cache and has rules about how that state may change.
  - For example a buffer managed by ext3 may not be marked both dirty and "journal dirty." That is, it should be written first to the journal (journal dirty), and then written to the appropriate location on disk (dirty).

Consistency Checks

- **Double fsck:**
  - By default fsck on a journaled file system simply replays the journal.
  - We compare the file system resulting from recovering in this manner with one generated after running fsck in a comprehensive mode that scans the entire disk checking for consistency. If they differ, at least one is wrong.

Scaling the System

- State Hashing and Search
  - Optimizations in steps: (1) state hashing, where FiSC selectively discards details to make bit-level different states equivalent and (2) searching, when it picks the next state to explore.
  - We initially hashed most things in the checked file system's state, such as the heap, data segment, and the raw disk. In practice this meant it was hard to comprehensively explore "interesting" states since the model checker spent its time re-exploring states that that were not that much different from each other.
  - After iterative experimentation we settled on only hashing the VolatileFS, the StableFS, and the list of currently runnable threads. We ignore the heap, thread stacks, and data segment. Users can optionally hash the actual disk image instead of the more abstract StableFS to check at a higher-level of detail.

Scaling the system

- Despite discarding so much detail we rarely can explore all states. Given the size of each checkpoint (roughly 1-3MB), the state queue holding all “to-beexplored” states consumes all memory long before FiSC can exhaust the search space.
- We stave this exhaustion off by randomly discarding states from the state queue whenever its size exceeds a user-selected threshold.

Scaling the system

- Searching
  - We provide two heuristic search strategies as alternatives to vanilla DFS or BFS.
  - The first heuristic attempts to stress a file system's recovery code by preferentially running states whose disks will likely take the most work to repair after a crash. It crudely does so by tracking how many sectors were written when the state's parent's disk was recovered and sorts states accordingly.

Scaling the system

- The second heuristic tries to quantify how different a given state is from previously explored states using a utility score. A state’s utility score is based on how many times states with the same features have already been explored.
- Features include: the number of dirty blocks a state has, its abstract file system topology, and whether its parent executed new file system statements. A state’s score is an exponentially-weighted weighted sum of the number of times each feature has been seen.

Systematically Failing Functions

- When a transition (e.g., mkdir, creat) is executed,it may perform many different calls to functions that can fail such as memory allocation or permission checks
- Blindly causing all combinations of these functions to fail risks having FiSC explore an exponential number of uninteresting, redundant transitions for each state. Additionally, in many cases FS-implementors are relatively uninterested in “unlikely” failures, for example, those only triggered when both memory allocation failures and a disk read error occurs.
- Instead, we use an iterative approach—FiSC will first run a transition with no failures, it will then run it failing only a single callsite until all callsites have been failed, it will then similarly fail two callsites, etc.
- Users can specify the maximum number of failures that FiSC will explore up to. The default is one failure. This approach will find the smallest possible number of failures needed to trigger an error.

Efficiently Modeling Large Disks

- As Figure 3 shows, naively modeling reasonable-sized disks with one contiguous memory allocation severely limits the number of states our model checker can explore as we quickly exhaust available memory.

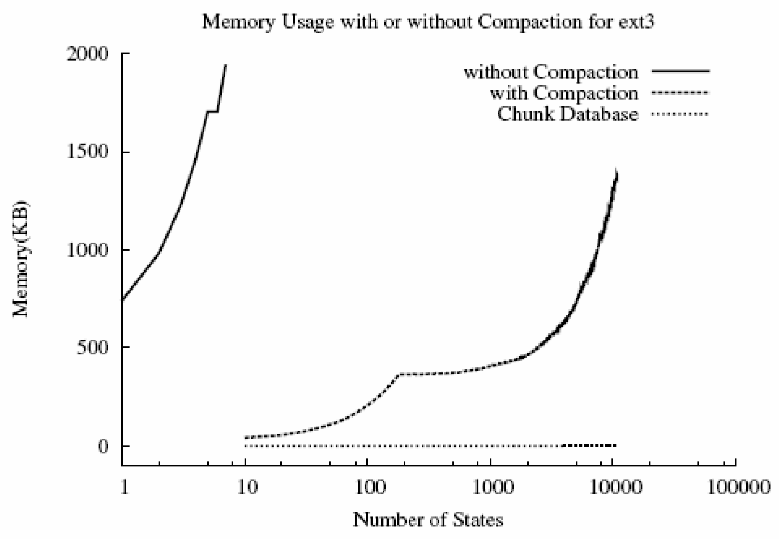


Figure 3: Memory usage when model checking ext3 on a 40MB disk with and without disk compaction. Without compaction the model checker quickly exhausts all the physical memory and dies before it reaches 20 states.

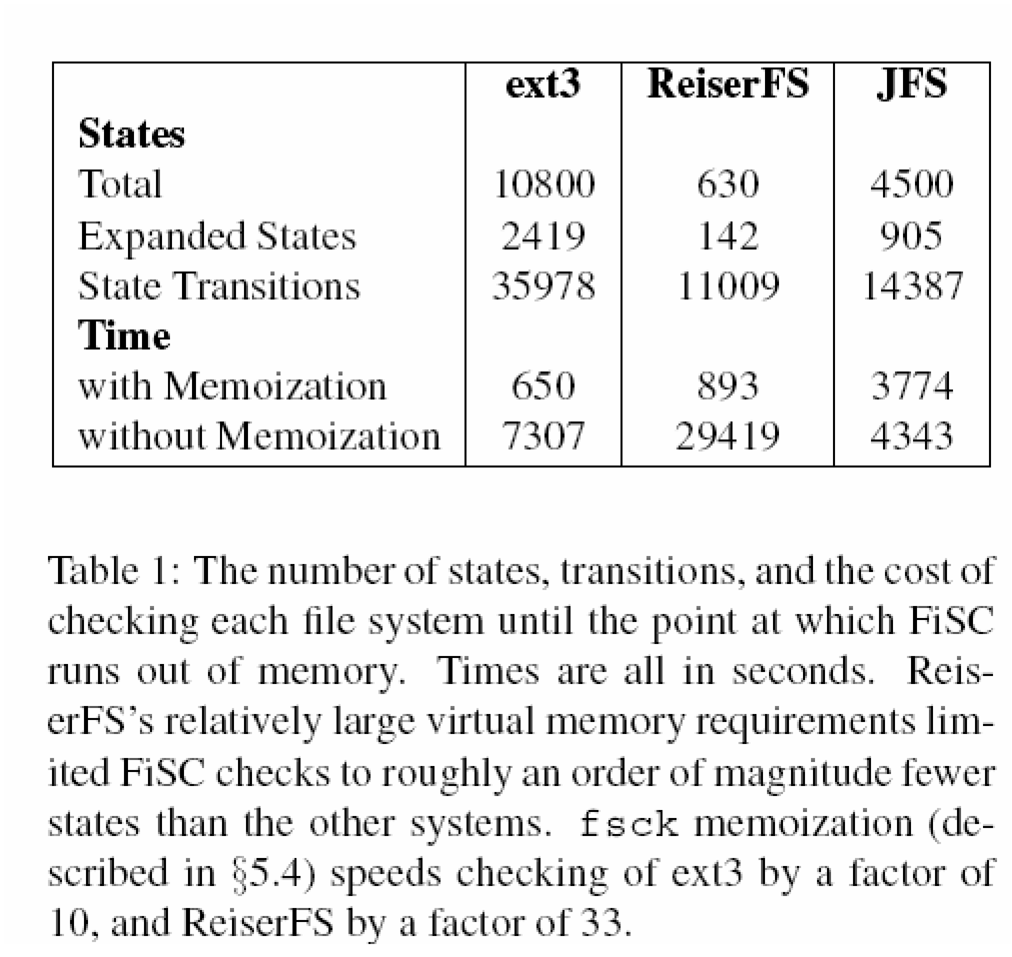
Efficiently Modeling Large Disks

- Changing file system code so that it works with a smaller disk is non-trivial and error prone as the code contains mutually dependent macros, structures, and functions that all rely on offsets in intricate ways. Instead we efficiently model large disks using hash compaction
- We keep a database of disk *chunks*, collections of disk sectors, and their hashes. The disk is thus an array of references to hashed chunks. When a write alters a chunk we hash the new value, inserting it into the database if necessary, and have the chunk reference the hash.

fsck Memoization

- Repairing a file system is expensive. It takes about five times as long to run fsck as it does to restore a state and generate a new state by executing an operation. If we are not careful, the time to run fsck dominates checking.
- Fortunately, for all practical purposes, recovery code is deterministic: given the same input disk image it should always produce the same output image. This determinism allows us to memoize the result of recovering a specific disk. Before running fsck we check if the current disk is in a hash table and, if so, return the already computed result. Otherwise we run fsck and add the entry to the table.
- While memoization is trivial, it gives a huge performance win as seen in Table 1.

Memoization



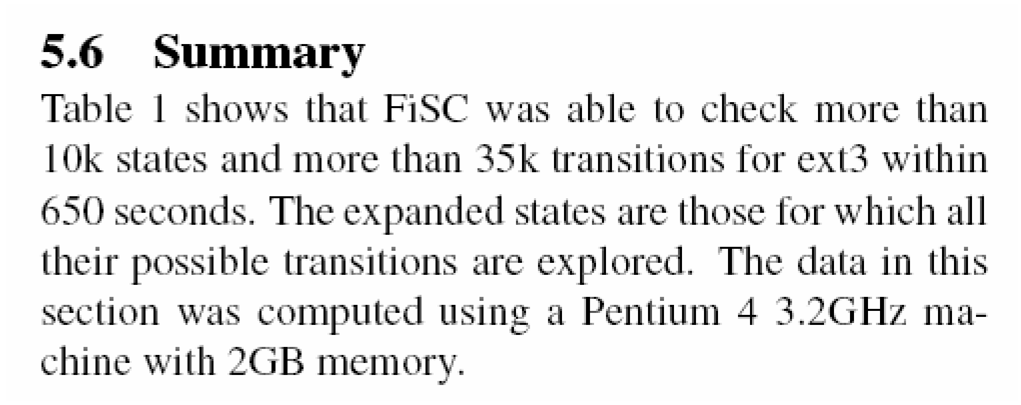
	ext3	ReiserFS	JFS
States			
Total	10800	630	4500
Expanded States	2419	142	905
State Transitions	35978	11009	14387
Time			
with Memoization	650	893	3774
without Memoization	7307	29419	4343

Table 1: The number of states, transitions, and the cost of checking each file system until the point at which FiSC runs out of memory. Times are all in seconds. ReiserFS's relatively large virtual memory requirements limited FiSC checks to roughly an order of magnitude fewer states than the other systems. `fsck` memoization (described in §5.4) speeds checking of ext3 by a factor of 10, and ReiserFS by a factor of 33.

Cluster-based Model Checking

- A model checking run makes a set of configuration choices: the number of files and directories to allow, what operations can fail, whether crashes happen during recovery, etc.
- Exploring different values is expensive, but not doing so can miss bugs. Fortunately, exploration is easily parallelizable. We wrote a script that given a set of configuration settings and remote machines, generates all configurations and remotely executes them.

Summary



5.6 Summary

Table 1 shows that FiSC was able to check more than 10k states and more than 35k transitions for ext3 within 650 seconds. The expanded states are those for which all their possible transitions are explored. The data in this section was computed using a Pentium 4 3.2GHz machine with 2GB memory.

PURDUE UNIVERSITY

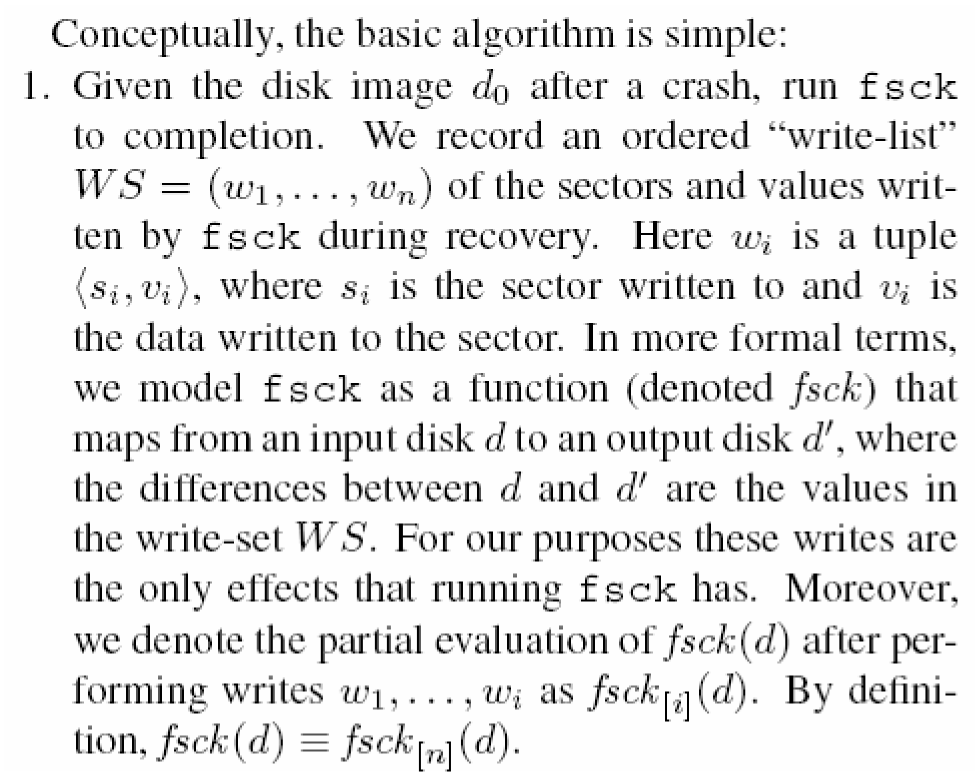
Crashes During Recovery

- A classic recovery mistake is to incorrectly handle a crash that happens during recovery. The number of potential failure scenarios caused by one failure is unwieldy, the number of scenarios caused by a second failure is combinatorially huge.
- Unfortunately, since many failures are correlated, such crashes are not uncommon.
- For example, after a power outage, one might run fsck only to have the power go out again while it runs.
- Similarly, a bad memory board will cause a system crash and then promptly cause another one during recovery.

Crashes During Recovery

- Next we describe how we check that a file system's recovery logic can handle a single crash during recovery.
- We check that if fsck crashes during its first recovery attempt, the final file system (the StableFS) obtained after running fsck a second time (on a disk possibly already modified by the previous run) should be the same as if the first attempt succeeded.
- We do not consider the case where fsck crashes repeatedly during recovery. While repeated failure is intellectually interesting, the difficulty in reasoning about errors caused by a single crash is such that implementors have shown a marked disinterest in more elaborate combinations.

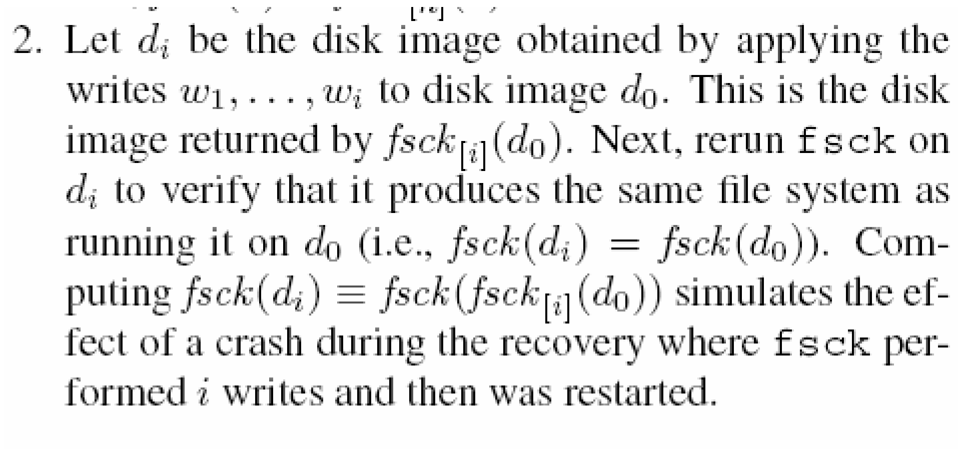
Algorithm



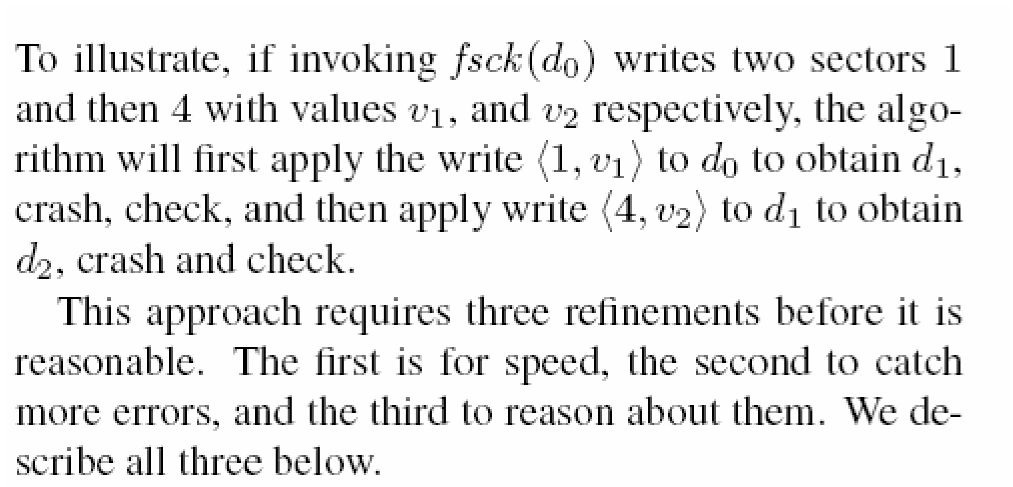
Conceptually, the basic algorithm is simple:

1. Given the disk image d_0 after a crash, run `fsck` to completion. We record an ordered "write-list" $WS = (w_1, \ldots, w_n)$ of the sectors and values written by `fsck` during recovery. Here w_i is a tuple $\langle s_i, v_i \rangle$, where s_i is the sector written to and v_i is the data written to the sector. In more formal terms, we model `fsck` as a function (denoted *fsck*) that maps from an input disk d to an output disk d', where the differences between d and d' are the values in the write-set WS. For our purposes these writes are the only effects that running `fsck` has. Moreover, we denote the partial evaluation of $\mathit{fsck}(d)$ after performing writes $w_1, \ldots, w_i$ as $\mathit{fsck}_{[i]}(d)$. By definition, $\mathit{fsck}(d) \equiv \mathit{fsck}_{[n]}(d)$.

Algorithm



2. Let d_i be the disk image obtained by applying the writes $w_1, \ldots, w_i$ to disk image d_0. This is the disk image returned by $fsck_{[i]}(d_0)$. Next, rerun `fsck` on d_i to verify that it produces the same file system as running it on d_0 (i.e., $fsck(d_i) = fsck(d_0)$). Computing $fsck(d_i) \equiv fsck(fsck_{[i]}(d_0))$ simulates the effect of a crash during the recovery where `fsck` performed i writes and then was restarted.



To illustrate, if invoking $fsck(d_0)$ writes two sectors 1 and then 4 with values v_1, and v_2 respectively, the algorithm will first apply the write $\langle 1, v_1 \rangle$ to d_0 to obtain d_1, crash, check, and then apply write $\langle 4, v_2 \rangle$ to d_1 to obtain d_2, crash and check.

This approach requires three refinements before it is reasonable. The first is for speed, the second to catch more errors, and the third to reason about them. We describe all three below.

Refinements

- **Speed From Determinism**
  - The naive algorithm checks many more cases than it needs to. We can dramatically reduce this number by exploiting two facts.
  - First, for all practical purposes we can regard fsck as a deterministic procedure. Determinism implies a useful property: if two invocations of a deterministic function read the same input values, then they must compute the same result. Thus, if a given write by fsck does *not change* any value it previously read then there is no need to crash and rerun it — it will always get back to the current state.
  - Second, fsck rarely writes data it reads. As a result, most writes do not require that we crash and recover: they will not intersect the set of sectors fsck reads and thus, by determinism, cannot influence the disk it would produce.

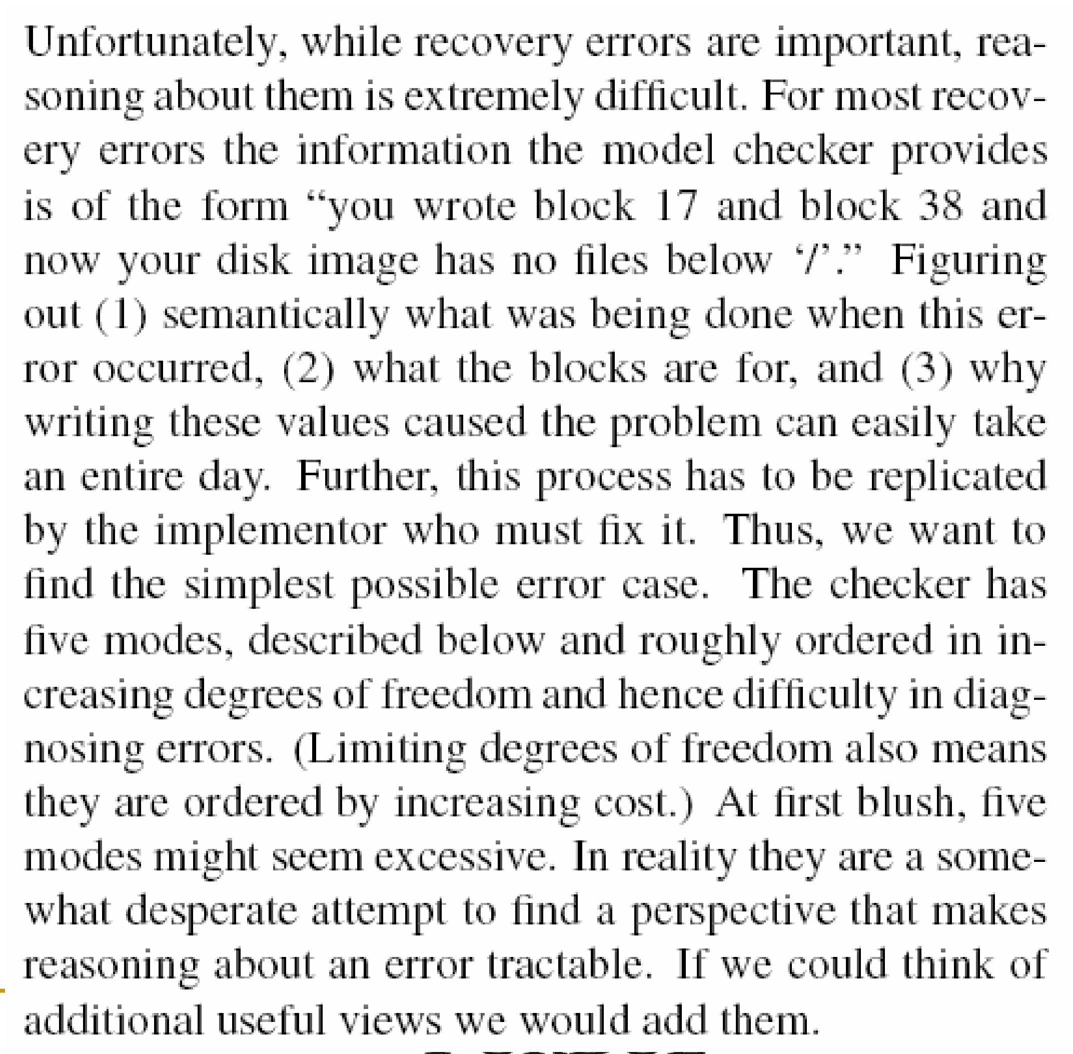
Checking All Write Orderings

- Sector writes can complete in any order unless (1) they are explicitly done synchronously or (2) they are blocked by a "sync barrier," such as the sync system call on Unix which (is believed to) only return when all dirty blocks have been written to disk.
- Thus, generating all disk images possible after crashing fsck at any point requires partitioning the writes into "sync groups" and checking that fsck would work correctly if it was rerun after performing any subset of the writes in a sync group (i.e., the power set of the writes contained in the sync group).
- For example, if we write sectors 1 and 2, call sync, then write sector 4, we will have two sync groups S0 = {1, 2} and S1 = {4}.
- The first, S0, generates three different write schedules: {(1), (2), (1, 2)}. A write schedule is the sectors that were written before fsck crashed (note that the schedule (2, 1) is equivalent to (1, 2) since we rerun fsck after both writes complete).

Checking All Write Orderings

- Given a sync group Si our checker does one of two things.
  - If the size of Si is less than or equal to a user-defined threshold, t, the checker will exhaustively verify all different interleavings.
  - If the size is larger than t the checker will do a series of trials, where it picks a random subset of random size within Si. These trials can be deterministically replayed later because we seed the pseudo-random number generator with a hash of the sectors involved.
  - We typically set t = 5 and the number of random trials to 7. Without this reordering we found no bugs in fsck recovery; with it we have found them in all three checked file systems.

Finding the Right Perspective



Unfortunately, while recovery errors are important, reasoning about them is extremely difficult. For most recovery errors the information the model checker provides is of the form "you wrote block 17 and block 38 and now your disk image has no files below '/'." Figuring out (1) semantically what was being done when this error occurred, (2) what the blocks are for, and (3) why writing these values caused the problem can easily take an entire day. Further, this process has to be replicated by the implementor who must fix it. Thus, we want to find the simplest possible error case. The checker has five modes, described below and roughly ordered in increasing degrees of freedom and hence difficulty in diagnosing errors. (Limiting degrees of freedom also means they are ordered by increasing cost.) At first blush, five modes might seem excessive. In reality they are a somewhat desperate attempt to find a perspective that makes reasoning about an error tractable. If we could think of additional useful views we would add them.

Finding the Right Perspective

- **Synchronous, atomic, logical writes.**
  - The first, simplest view is to group all sector writes into "logical writes" and do these synchronously (i.e., in the order that they occur in program execution). Here, logical writes means we group all blocks written by the same system call invocation as one group. I.e., if there are two calls to the write system call, the first writing sectors l0 = (1, 2, 3) and the second writing sectors l1 = (7, 8) we have two logical operations, l0 and l1.
  - We apply all the writes in l0, crash, check, apply the writes in l1 crash, and check.
  - This is the strongest logical view one can have of disk: all operations complete in the order they were issued and all the writes in a single logical operation occur atomically.

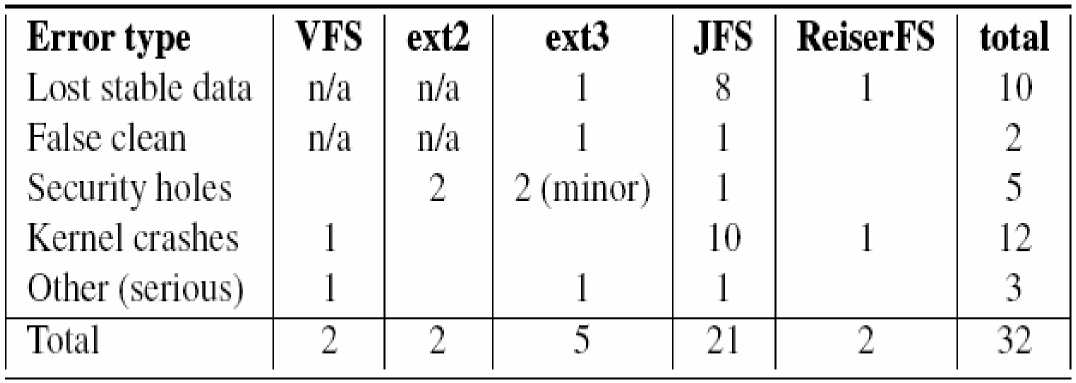
Finding the Right Perspective

- **Synchronous, non-atomic, left-to-right logicalwrites.**
  - Here we still treat logical writes as synchronous, but write their contained sectors non-atomically, left-to right. I.e., write the first sector in a logical group, crash, check, then write the next, etc.
  - These errors are also relatively easy to reason about and tend to be localized to a single invocation of a system call where a data structure that was assumed to be internally consistent straddled a sector boundary.
- **Reordered, atomic logical writes.**
  - This mode reorders all logical writes within the same sync group, checking each permutation. These errors can often be fixed by inserting a single sync call.

Finding the Right Perspective

- **Synchronous, non-atomic logical writes.**
  - This mode writes the sectors within a logical operation in any order, crashing after each possible schedule. These errors are modular, but can have interesting effects.
- **Reordered sector writes.**
  - This view is the hardest to reason about, but the sternest test of file system recovery: reorder all sectors within a sync group arbitrarily.
  - We do not like these errors, and if we hit them will make every attempt to find them with one of the prior modes.

Results



Error type	VFS	ext2	ext3	JFS	ReiserFS	total
Lost stable data	n/a	n/a	1	8	1	10
False clean	n/a	n/a	1	1		2
Security holes		2	2 (minor)	1		5
Kernel crashes	1			10	1	12
Other (serious)	1		1	1		3
Total	2	2	5	21	2	32

Table 2: We found 32 errors, 10 of which could cause permanent data loss. There are 3 intended errors, where programmers decided to sacrifice consistency for availability. They are not shown in this table.

Results

- **Unrecoverable Data Loss**
  - The most serious errors we found caused the irrevocable loss of committed, stable data. There were 10 such errors where an execution sequence would lead to the complete loss of metadata (and its associated data) that was committed to the on-disk journal.
  - In several cases, all or large parts of long-lived directories, including the root directory "/", were obliterated. Data loss had two main causes: (1) invalid write ordering of the journal and data during recovery and (2) buggy implementations of transaction abort and fsck.

Results

- **Invalid recovery write ordering.**
  - There were three bugs of this type. During normal operation of a journaling file system the journal must be flushed to disk before the data it describes. The file systems we check seem to get this right.
  - However, they all get the inverse of this ordering constraint wrong: during recovery, when the journal is being replayed, all data modified by this roll forward must be flushed to disk before the journal is persistently cleared. Otherwise, if a crash occurs, the file system will be corrupt or missing data, but the journal will be empty and hence unable to repair the file system.

Results

- **Buggy transaction abort and fsck.**
  - There were five bugs of this type, all in JFS. Their causes were threefold.
  - First, JFS immediately applies all journaled operations to its in-memory metadata pages. Unfortunately, doing so makes it hard to roll back aborted transactions since their modifications may be interleaved with the writes of many other ongoing or committed transactions. As a result, when JFS aborts a transaction, it relies on custom code to carefully extricate the side-effects of the aborted transactions from non-aborted ones. If the writer of this code forgets to reverse a modification, it can be flushed to disk, interlacing many directories with invalid entries from aborted transactions.

Results

- Second, JFS's fsck makes no attempts to recover any valid entries in such directories. Instead its recovery policy is that if a directory contains a single invalid entry it will remove all the entries of the directory and attempt to reconnect subdirectories and files into "lost+found."
- This opens a huge vulnerability: any file system mistake that results in persistently writing an invalid entry to disk will cause fsck to deliberately destroy the violated directory.

Results

- Third, JFS fsck has an incorrect optimization that allows the loss of committed subdirectories and files. JFS dynamically allocates and places inodes for better performance, tracking their location using an “inode map.”
- For speed, incremental modifications to this map are written to the on-disk journal rather than flushing the map to disk on every inode allocation or deletion. During reconstruction, the fsck code can cause the loss of inodes because while it correctly applies these incremental modifications to its copy of the inode map, it deliberately does not overwrite the out-of-date, on-disk inode map with its (correct) reconstructed copy.

Results

- **Security Holes**
  - While we did not target security, FiSC found five security holes, three of which appear readily exploitable.
  - The easiest exploit we found was a storage leak in the JFS routine jfs link used to create hard links. It calls the routine get UCSname, which allocates up to 255 bytes of memory. jfs link must (but does not) free this storage before returning. This leak occurs each time jfs link is called, allowing a user to trivially do a denial of service attack by repeatedly creating hard links.
  - Even ignoring malice, leaking storage on each hard link creation is generally bad.

Results: Security Holes

- The two other seemingly exploitable errors both occurred in ext2 and were both caused by lookup routines that did not distinguish between lookups that failed because (1) no entry existed or (2) memory allocation failed. The first bug allows an attacker to create files or directories with the same name as a preexisting file or directory, hijacking all reads and writes intended for the original file.
- The second allows a user to delete nonempty directories to which they do not have write access.

Experience

- **FiSC-Assisted Development**
  - We checked preexisting file systems, and so could not comprehensively study how well model checking helps the development process.
  - However, the responsiveness of the JFS developers allowed us to do a micro-case study of FiSC-assisted software development by following the evolution of a series of mistaken fixes

Experience

- While there are many caveats that one must keep in mind, model checking has some nice properties. First, it makes it trivial to verify that the original error is fixed.
- Second, it allows more comprehensive testing of patches than appears to be done in commercial software houses.
- Third, it finds the corner-case implications of seemingly local changes in seconds and demonstrates that they violate important consistency invariants.

False Positives

- The false positives we found fell into two groups. Most were bugs in the model checking harness or in our understanding of the underlying file system and not in the checked code itself. The latter would hopefully be a minor problem for file system implementors using our system (though it would be replaced by problems arising from their imperfect understanding of the underlying model checker).
- The other group of false positives stemmed from implementors intentionally ignoring or violating the properties we check. For example, ReiserFS causes a kernel panic when disk read fails in certain circumstances. Fortunately, such false positives are easily handled by disabling the check.

False Negatives

- **Exploring thresholds.**
  - We do a poor job of triggering system behavior that only occurs after crossing a threshold value. The most glaring example: because we only test a small number of files and directories (<15) we miss bugs that happen when directories undergo reorganization or change representations only after they contain a "sufficient" number of entries.
  - Real examples include the re-balancing of directory tree structures in JFS or using a hashed directory structure in ext3. With that said, FiSC does check a mixture of large and small files (to get different inode representations) and file names or directories that span sector boundaries (for crash recovery).

False Negatives

- **Multi-threading support.**
  - The model checker is single-threaded both above and below the system call interface. Above, because only a single user process does file system operations. Below, because each state transition runs atomically to completion. This means many interfering state modifications never occur in the checked system. In particular, in terms of high-level errors, file system operations never interleave and, consequently, neither do partially completed transactions (either in memory or on disk). We expect both to be a fruitful source of bugs.

False Negatives

- **White-box model checking.**
  - FiSC can only flag errors that it sees. Because it does not instrument code it can miss low-level errors, such as memory corruption, use of freed memory, or a race condition unless they cause a crash or invariant violation.
  - Fortunately, because we model-check implementation code we can simultaneously run dynamic tools on it.

False Negatives

- **Unchecked guarantees.**
  - File systems provide guarantees that are not handled by our current framework. These include versioning, undelete operations, disk quotas, access control list support, and journaling of data or, in fact, any reasonable guarantees of data block contents across crashes.
  - The latter is the one we would like to fix the most. Unfortunately, because of the lack of agreed upon guarantees for non-sync'd data across crashes we currently only check metadata consistency across crashes— data blocks that do not precede a "sync" point can be corrupted and lost without complaint.

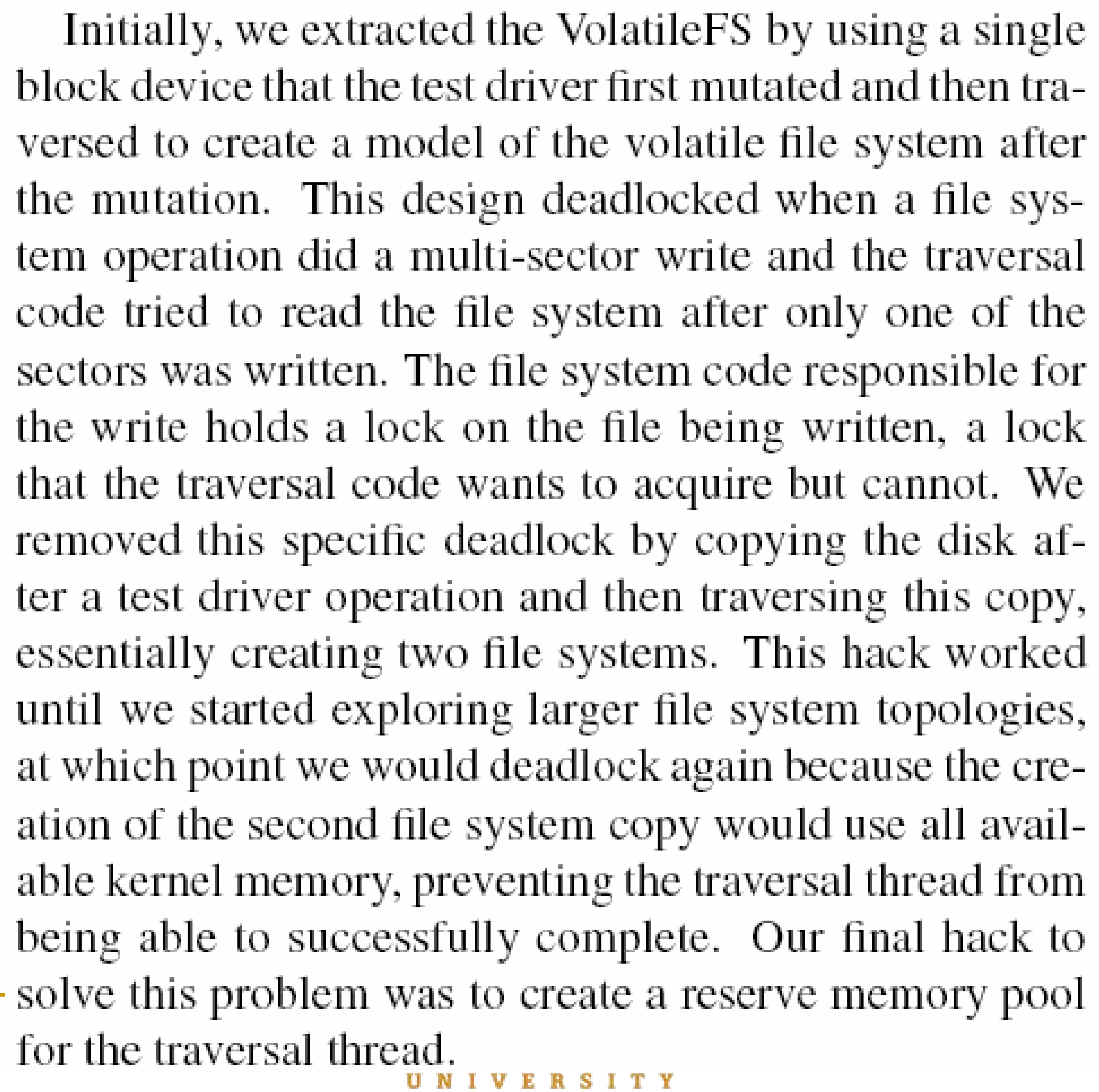
False Negatives

- **Missed states.**
  - While our state hashing can potentially discard too much detail, we do not currently discard enough of the right details, possibly missing real errors. Using FS-specific knowledge opens up a host of additional state optimizations.
  - One profitable example would be if we knew which interleavings of buffer cache blocks and fsck written blocks are independent (e.g., those for different files), which would dramatically reduce the number of permutations needed for checking the effects of a crash.

Design Lessons

- One hard lesson we learned was a sort of “Heisenberg” principle of checking: make sure the inspection done by your checking code does not perturb the state of the checked system.
- Violating this principle leads to mysterious bugs. A brief history of the code for traversing a mounted file system and building a model drives this point home.

Design Lessons



Initially, we extracted the VolatileFS by using a single block device that the test driver first mutated and then traversed to create a model of the volatile file system after the mutation. This design deadlocked when a file system operation did a multi-sector write and the traversal code tried to read the file system after only one of the sectors was written. The file system code responsible for the write holds a lock on the file being written, a lock that the traversal code wants to acquire but cannot. We removed this specific deadlock by copying the disk after a test driver operation and then traversing this copy, essentially creating two file systems. This hack worked until we started exploring larger file system topologies, at which point we would deadlock again because the creation of the second file system copy would use all available kernel memory, preventing the traversal thread from being able to successfully complete. Our final hack to solve this problem was to create a reserve memory pool for the traversal thread.

Design Lessons

- In retrospect, the right solution is to run two kernels side by side: one dedicated to mutating the disk, the other to inspecting the mutated disk.
- Such isolation would straightforwardly remove all perturbations to the checked system.

Design Lessons

- A similar lesson is that the system being checked should be instrumented instead of modified unless absolutely necessary.
- Code always contains hidden assumptions, easily violated by changing code. For example, the kernel we used had had its kernel memory allocators re-implemented in previous work as part of doing leak checking.
- While this replacement worked fine in the original context of checking TCP, it caused the checked file systems to crash. It turned out they were deliberately mangling the address of the returned memory in ways that intimately depended on how the original allocator (page alloc) worked. We promptly restored the original kernel allocators.

Conclusion

- This paper has shown how model checking can find interesting errors in real file systems. We found 32 serious errors, 10 of which resulted in the loss of crucial metadata, including the file system root directory "/". The majority of these bugs have resulted in immediate patches.
- Given how heavily-tested the file systems we model checked were and the severity of the errors found, it appears that model checking works well in the context of file systems.
- The underlying reason for its effectiveness in this context seems to be because file systems must do so many complex things right. The single worst source of complexity is that they must be in a recoverable state in the face of crashes (e.g., power loss) at every single program point. We hope that model checking will show similar effectiveness in other domains that must reason about a vast array of failure cases, such as database recovery protocols and optimized consensus algorithms.

Model checking Tools

- SPIN
  - For asynchronous distributed algorithms: Distributed System Design
  - Ref:
    - *The Model Checker Spin*, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.
  - Examples of applications:
    - Process scheduling in a distributed operating system
    - Flow Control: Safety and liveness properties
      - E.g. Absence of deadlock, unreachable code etc.

Other Model Checking Tools

- CBMC
  - A Bounded Model Checker for ANSI-C Programs
  - Checks safety properties like correctness of pointer constructs, array bounds and user specified assertions
  - http://www.cs.cmu.edu/~modelcheck/cbmc/

Other Model Checking Tools (Contd.)

- BLAST: Berkeley Lazy Abstraction Software Verification Tool
  - Checks temporal safety properties of C programs
  - Checks memory safety too
  - “The BLAST query language for software verification”, Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Proceedings of the 11th International Static Analysis Symposium (SAS 2004), LNCS 3148, pages 2-18, Springer-Verlag, 2004.

References

- Introduction to Model Checking, Online at http://www.embedded.com/columns/showArticle.jhtml?articleID=17603352
- S. Kumar and K. Li. Using model checking to debug device firmware. In 5th Symp on Operating Systems Design and Impl. 2002.
- Bonnie Brinton Anderson, James V. Hansen, Paul Benjamin Lowry, Scott L. Summers, “The application of model checking for securing e-commerce transactions” CACM 2006

References

- Junfeng Yang, Paul Twohey, Dawson Engler, Madanlal Musuvathi, “**Using Model Checking to Find Serious File System Errors”,** OSDI 04